Avoka Transact

How-to articles

Useful articles on using Transact

By Subject Area:

- Transact Platform Adminstration
- Transact Manager (TM)
- Integration
- Maestro
- Composer
- Collaboration Jobs and Bundling
- Groovy Services
- Application Development Patterns
- TransactField Mobile App
- Transact SDK

By Topic:

A

abandonment
abn
accessibility
adaptive
address
administration
advisory
analytics
announcement
anonymous
apis
approval
assignment
attachments
avoka

B

birt
blocks
brand
browser

C

cascading
change
cloud
collaboration
compatibility
component
composer
confirmation
copy
csp
customer-care-portal

D

data
data-driven
data-security
datepicker
debugging
delivery
deployment
description
design
detail
details
development
dropdown
dynamic

E

email
endpoint
environments
error
errors
exceptions
export

F-G

file-list
font-awesome
fonts
form
form-chaining
form-space
forminformation
forms
general
google
googletagmanager
groovy

H-K

help
hidden
hosting
import
integration
ios
javascript
jquery
kb-how-to-article
kb-how-to-video
kb-troubleshooting-article

L-M

layout
library
logging
login
maestro
maguire
maguire-template
manager
mastersoft
mismatch
mobile
mobile app
mobile-app

N-Q

native
navigation
on-premise
operations
parameter
password
payment
pdf
perfect
pixel
portal
prefill
prepopulation
properties

R

receipt
recolor
reconciliation
repeating-blocks
repeats
responsive
retrospective
review
rules

S

salesforce
saving
scripting
search
security
service-connection
sharepoint
sharing
signatures
soap-ui
space
static
styling
submission
submit
support

T

task
task-assignment
templates
testing
tfield
tm
tracking
transact
transact-field
transaction
transaction manager
transaction-life-cycle
troubleshooting

U-Z

uat
upgrade
url
validation
veda
velocity
version
versions
watermark
web
widget
widgets
windows
xml

Transact Platform Adminstration

Configuration

- Composer: How to communicate the right form for development or support
- Configuring Apache for On Premise Installations
- Deploying and customising a new Form Space (Portal)
- How to add a "favicon" to a Form Space (Portal)
- How to Change Transact Manager Database Password
- Managing Content Security Policy (CSP) Settings
- On-premise Composer Upgrades
- So you’ve made a mess of your development environment? Here’s how to clean up!
- Transact Manager on-premise deployment notes/check list
- Upgrading a Form Space (Portal)

Composer: How to communicate the right form for development or support

Unknown macro: 'redirect'

Once your system is in place, there will come a time when you would like to update forms, or request some mentoring or support on a form design. Throughout the life cycle of composer environment, you may rearrange folders or projects, take copies and backups of forms or change form names, it is really important to ensure that you are aware of which form is the current production form, and subsequently, communicate this to your developers or support team when requesting assistance.

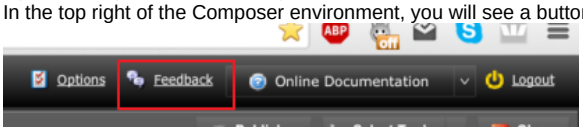
See this article on recommendations on how to ensure you keep your environment clean, and how to clearly identify the current production form in Composer.

This short guide will show you how to get the details of the form you wish to request support or development on.

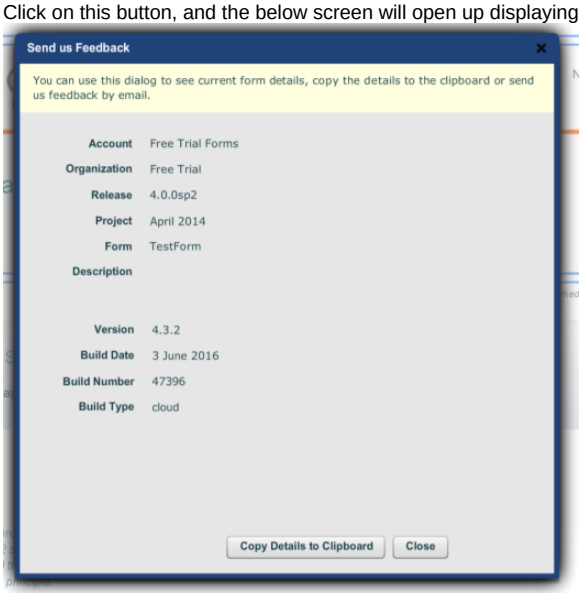
Step-by-step guide

Before you begin, ensure you have identified for yourself the current production form in the Composer environment. If you have followed the instructions in the linked article above, then you may have written in the description of the form "Current PROD form".

1. Open the required form in Composer
2. In the top right of the Composer environment, you will see a button labelled "Feedback"



3. Click on this button, and the below screen will open up displaying details about your form.



4. The idea behind this screen is to enable you to collect information about the form, the organisation and the composer environment you are using, to make it clear and simple for your development or support teams to identify which form you need assistance on.
5. To communicate these details, you can simply click on "Copy Details to Clipboard" and then paste the clipboard into an email, or into your support ticket.

Here is a sample of the output of clicking "Copy Details to Clipboard". You can see that the resultant information is very comprehensive and will greatly assist your support and development teams to identify your form clearly.

-- Form Details --
Account: Test Forms
Organization: Test Forms

Project: April 2014
Form: TestForm
Template Name: Template-Maguire

-- Build Details --
Version: 4.3.2
Build Date: 3 June 2016
Build Number: 47396
Build Type: cloud

-- Search Path Details --
Release Path: Release v4.0.0
Service Pack path: v4.0 Service Pack 2
Custom Path: Free Trial Forms__Updated Maguire Pack 4 Current

-- Browser Details --
Chrome/51.0.2704.103
https://transact.composer.avoka.com.au/composer/secure/composer.htm

Related articles

- Rendering a newly imported Form generates an IllegalArgumentException error
- Composer: How to communicate the right form for development or support

Configuring Apache for On Premise Installations

Unknown macro: 'redirect'

Avoka provides (and recommends) an Apache server to sit in front of the Transact Manager server. The Apache server provides various services such as reverse-proxy, caching, SSL termination, and security. Avoka provides the Apache server and a set of configuration files for this server as part of our installer.

Our customers sometimes ask whether the Apache server configuration can be modified, or whether the Apache server can be re-used for other purposes.

The short answer is yes, but with care.

- In an on-premise environment, you (the customer or your hosting provider) are responsible for the environment. This includes configuring the Apache server, applying patches, monitoring, etc. You are welcome to configure this any way you want, as long as you are comfortable that it meets your security and other requirements, since this is your responsibility.
- The Apache server that Avoka provide is pre-configured with what we consider to be our best practices for security, reverse proxy, SSL termination, etc. You can use our configuration as-is, or you are welcome to use our configuration as a starting point, and modify it. You are also welcome to configure the Apache server from the ground up. And you are welcome to use the Apache servers for other purposes. However, we recommend only staff with strong Apache experience do this, that you do this with care, and with the knowledge that this is your responsibility.
- If you would like to understand exactly what our recommended Apache configuration looks like, you can review the Apache configuration files. After you install the Transact Installer, the apache config template files can be found in the “avoka-tm-setup” folder.
  Below are screenshots from Transact Manager 17.10.1, which supports both Apache 2.2 and Apache 2.4.

```
~]$ ls -al /data/avoka-tm-setup
total 40
drwxrwxr-x. 12 tmuser tmuser  4096 Dec  1 15:06 .
drwxr-xr-x.  9 tmuser tmuser  4096 Dec  1 15:03 ..
drwxrwxr-x.  5 tmuser tmuser    48 Dec  1 15:05 apache-2.2
drwxrwxr-x.  5 tmuser tmuser    48 Dec  1 15:05 apache-2.4
drwxrwxr-x.  3 tmuser tmuser    82 Dec  1 15:05 db-updater
drwxrwxr-x.  2 tmuser tmuser    45 Dec  1 15:05 examples
drwxrwxr-x.  2 tmuser tmuser    74 Dec  1 15:05 linux
drwxrwxr-x.  2 tmuser tmuser    64 Dec  1 15:05 livecycle
drwxrwxr-x.  9 tmuser tmuser    82 Dec  1 15:05 modules
drwxrwxr-x.  4 tmuser tmuser    32 Dec  1 15:05 reports
drwxrwxr-x.  2 tmuser tmuser    64 Dec  1 15:05 sql
-rw-rw-r--.  1 tmuser tmuser 26162 Dec  1 15:05 standalone.xml
drwxrwxr-x.  2 tmuser tmuser  4096 Dec  1 15:05 war
~]$ ls -al /data/avoka-tm-setup/apache-2.4
total 8
drwxrwxr-x.  5 tmuser tmuser   48 Dec  1 15:05 .
drwxrwxr-x. 12 tmuser tmuser 4096 Dec  1 15:06 ..
drwxrwxr-x.  3 tmuser tmuser   95 Dec  1 15:05 linux
drwxrwxr-x.  4 tmuser tmuser   32 Dec  1 15:05 modsecurity
drwxrwxr-x.  3 tmuser tmuser 4096 Dec  1 15:05 win64
~]$ ls -al /data/avoka-tm-setup/apache-2.4/linux
total 12
drwxrwxr-x. 3 tmuser tmuser   95 Dec  1 15:05 .
drwxrwxr-x. 5 tmuser tmuser   48 Dec  1 15:05 ..
drwxrwxr-x. 2 tmuser tmuser   38 Dec  1 15:05 additional-config
-rw-rw-r--. 1 tmuser tmuser  421 Dec  1 15:05 mod_cache.conf
-rw-rw-r--. 1 tmuser tmuser 1492 Dec  1 15:05 mod_deflate.conf
-rw-rw-r--. 1 tmuser tmuser 1489 Dec  1 15:05 mod_proxy.conf
```

```
~]$ ls -al /data/avoka-tm-setup/apache-2.4/linux/additional-config
total 16
drwxrwxr-x. 2 tmuser tmuser   38 Dec  1 15:05 .
drwxrwxr-x. 3 tmuser tmuser   95 Dec  1 15:05 ..
-rw-rw-r--. 1 tmuser tmuser 9479 Dec  1 15:05 httpd.conf
-rw-rw-r--. 1 tmuser tmuser 3017 Dec  1 15:05 ssl.conf
```

Deploying and customising a new Form Space (Portal)

Unknown macro: 'redirect'

Compatibility

Since	4.0
Deprecated	

Introduction

This article describes how to add a new space to Transact.

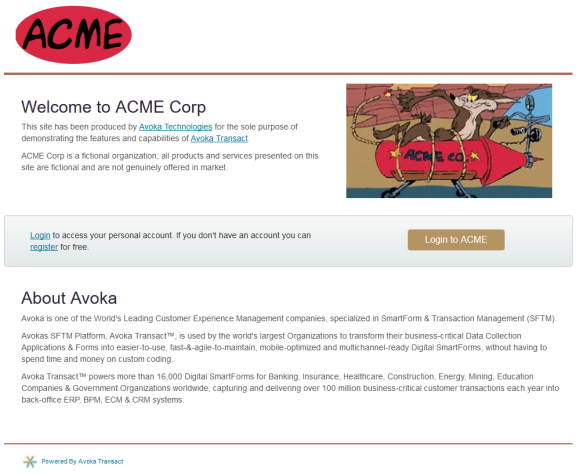
Spaces can be used to provide branded access to a set of forms and to a group of users securely or anonymously (unsecured). You can deploy multiple spaces to provide unique entry points for customers, partners, helpdesk employees or employee self service; you can whitelist by IP, or authenticate against different directories. Forms can be deployed to multiple Spaces or to a single space to provide access to a single group of users.

At least one Space is required to provide access to forms (Transact 3.x and 4.0). At a minimum, the Space styling is used for the Confirmation Page and Attachment Page (Deprecated with FTX in 4.1). When using a fresh deployment of Transact, you will need to customize a Space to reflect your brand. When operating shared services, you will need to deploy a Space for each department/brand which you support. When managing different classes of users (employees, contractors, partners, public) who perform different data collection functions (multi-solution platform), you will want to deploy a new Space for each class of user.

Avoka staff often follow this guide to build branded demos, or to instantiate cost-effective POC environments on shared infrastructure. Clients can use this guide to manage their own platform.

This article is written for beginners, however it assumes that you have a running version of Transaction Manager 4.0, and that you have a *very* basic knowledge of HTML & CSS (we will still walk you through it).

The outcome of this guide will be something that looks like this: https://tm.demo.avoka.com/acme/



Preparation

Before getting started, you will need:

- Admin access to Transaction Manager
- Access to Apache configuration (or if hosted by Avoka, access to customer support)
- A copy of the Transaction Manager installer "Avoka-TM-Setup.jar" (ensure that you have the correct version). If you do not have it, you can get it here.
- Optionally a form, ready to deploy, waiting in Composer for publishing.

You will also need to decide a few things:

- The **name** of the Space (EG: ACME Partner Space)
- The **base URL context path** (IE: the bit of the URL immediately after the domain name https://client.transactcentral.com/***XXXX***, where ***XXXX*** is the base URL context path). Note that your domain name will differ from this specific example.
- Which existing site to use as inspiration for theming.

Demo scenario

This guide will make some assumptions for the purposes of demonstration. You will need to make the obvious substitutions for the following:

- Transaction Manager Server: "tm.demo.avoka.com"
- Company name (brand): ACME

Method

1. Configure Apache

Configure Apache on your server to add a new context path "/acme". Traditionally URLs are lowercase; you should follow this convention.

If your Edition of Transaction Manager is hosted by Avoka, please contact support to request this configuration, and clearly identify which environment(s) should be updated (prod, test, UAT etc).

If your Edition of Transaction Manager is on premise, you may need to request a change to be made to your Web Server via your IT department, depending on your specific deployment configuration.

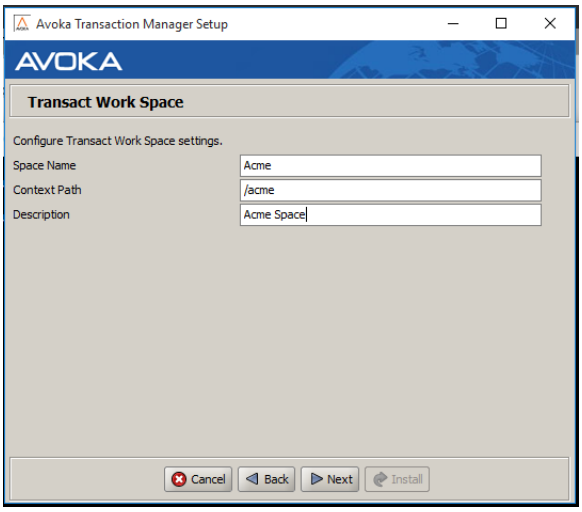
2. Run the java Transaction Manager Installer

This will not install software onto your computer, it just writes out some files. Just run it locally on your personal computer.

You should be able to just double-click the file "Avoka-TM-Setup.jar", however sometimes your computer's security settings will not allow double clicking jar files. In that case you can type "java -jar Avoka-TM-Setup.jar".

In the wizard:

- Make sure that you select the option to "**Create a New Work Space**".
- You will need fill in some details (see image below): The Space Name and Description fields are used only in Transaction Manager to help catalogue the space, but the Context Path option **must** match the context path defined in Apache.
- Ignore any additional options until you get to the "installation directory" page - you may want to choose something unique, if you are managing more than one Space or more than one version of Transaction Manager:
- Complete the wizard. Hitting "Install" will just produce a bunch of files in a directory, which we will use in the next step.

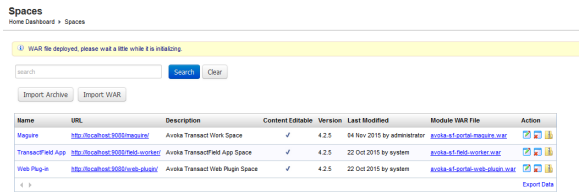


3. Deploy the Space

Log into Transaction Manager, and navigate to "Forms" -> "Form Spaces"

Press "Import WAR", and browse to the files you just created with the installer. You will need to find the "**Deploy**" directory, which will contain **a single .war file**.

A successfully deployed WAR file will look like this:

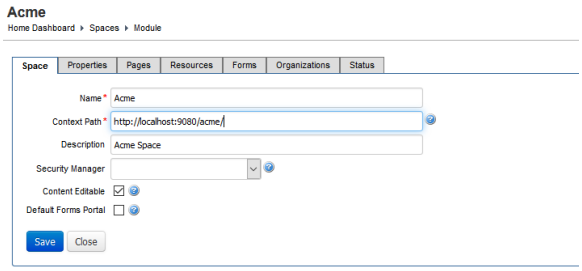


...but it still won't work, until you configure the portal...

4. Configure the Space

Select the Space Name "ACME" to edit it.

You must edit the "Context Path" (which is set to "http://localhost:9080/acme/") to match the Internet Domain Name which matches your server. The easiest way to do this is to copy from your browser address bar:



You now have a deployed (but not configured) portal. You can test it by copying and pasting the entire Context Path into a new browser tab.

Before we get into branding, it is worthwhile to configure a few things at this point. This guide does not go into detail, however you will need to think about:

1. Creating & configuring an Organisation and associating it with the Space
2. Provisioning access for yourself and any colleagues. Alternatively you might associate & configure a security manager (external directory)
3. Deploy a form or two. At least for testing.

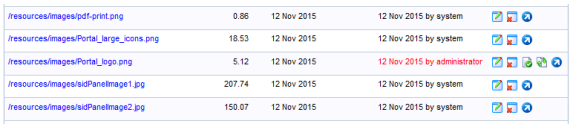
5. Brand the Space

To keep things simple, we will look at performing a very basic branding (logo and colours):

5.1 Replace the logo & hero image

In the Space settings, navigate to the "Resources" tab, and find the "Portal_logo.png" object. Replace this with a file of your choice, by clicking the name of the resource. If you want to keep things simple, you should use a logo which is about 300x50px (+/-), has no border or bleed area and has a transparent or white background. (Feel free to get complex if you have time).

A successfully replaced resource will have a red modified date, but will retain the original resource name despite the filename which you have uploaded:



Do the same for the hero image (which is used by the index.html page). To replace this, look for the resource called "Fincorp_lady_370.jpg", and replace it. To keep things simple, use an image which is exactly 370 x 205 px.

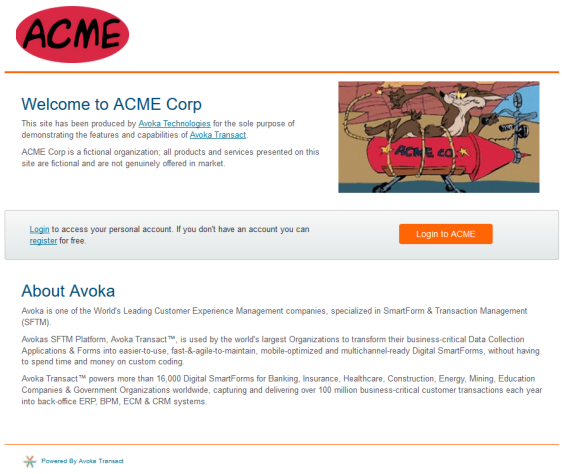
Next, you need to set the size of the logo. Look for the "Style.css" resource, and edit it. Scroll right to the bottom of the page to see the logo class, to edit the size of the logo:

```
.jobItemPadding {
    margin-left: 30px;
}
#logo {
    background: url('../images/Portal_logo.png');
    background-size: 200px 100px;
    width: 200px;
    height: 100px;
    margin: 10px 20px;
    display: inline-block;
}
.mobileOnly {
    display: none !important;
}
.taskTileMargin {
    margin-left: 10px;
}
.taskSectionMargin {
    margin-top: 10px;
}
.jobTitle {
    padding-left: 0px;
    margin-bottom: 5px;
    margin-top: 15px;
}
```

5.2 Edit the index.html page

In the Space settings, navigate to the "Pages" tab, and find the "index.html" object. You can edit this page in the same way as above, to remove any reference to Maguire. You can keep things simple as displayed below, or you can replace the entire landing page with security information or alternate links to enrolment forms which do not require authentication. Some clients will delete all content in this page, so that staff can only get to resources via a link on another system (EG: Extranet or Intranet site).

By now, you should have something which has the following customisations:

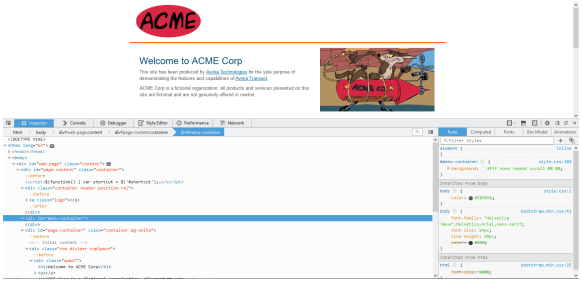


5.3 Edit the portal CSS to reflect the brand

Edit the "Style.css" resource to change colours. You can use websites such as Stylify Me to choose new colors from an existing site. To start with look for the following:

1. Any reference to Avoka's bright orange colour #FF6405.Replace this with a bright corporate "callout" colour. It's used for important things like reference numbers.
2. .h1 class color (from #005280)
3. .header border-bottom & .footer border-top (The page header & footer dividers)
4. .btn-primary background-colour and .btn-primary:hover background-colour. Use primary call to action colours here.

You can go much further than this. If, whilst testing, you find an element which you would like to recolour, you can easily find the class which describes it's settings by using your browser debugger. Just right-click on the object, and hit "inspect element" to get something like this:



If you are unfamiliar with your browser debugger

Keep in mind one technique: you may need to explore the HTML (bottom left pane), to find the right element, and you will need to scroll through the CSS pane (bottom right) to find the reference to the colour. Most debuggers will display a colour swatch (in this case orange), which can be used to find the right class quickly.

Once you have found the name of the class (in this case .btn-index), you will be able to search the Style.css file in Transaction Manager to edit the right property (in this example: background-color)

Next steps....

- If you've built your Space in a non-production environment, it's easy to export it, and then import it onto a different server. All your Space settings will be retained. In this case, you will also need to think about the organisation, and forms and user configuration which can be exported separately if required.
- You can perform much more complex customisation of the Space, either by editing HTML pages & resources, or by editing the contents of the WAR file produced by the installer. This is not described in this document.
- Share the URL to some of your Space resources including forms, the base index page, or links directly to any page in the Space (EG: Account page, main menu, To Do list etc). These can be used by other systems to provide direct navigation to Transact resources.

Related articles

- Upgrading a Form Space (Portal)
- How to add a "favicon" to a Form Space (Portal)
- Deploying and customising a new Form Space (Portal)

How to add a "favicon" to a Form Space (Portal)

Unknown macro: 'redirect'

1. Upload a new fav icon file under spaces > resources, give it a different file name from the standard one as browsers cache these things very aggressively, and they are hard to clear out. Make sure it is in the correct format.

Maguire

Home Dashboard ▸ Spaces ▸ Module

Space | Properties | Pages | **Resources** | Forms | Organizations | Status

search Type PNG Search Clear

Resource Path	Size KB	Base Content Date	Merged	Last Modified	Action
/resources/images/arrow_orange.png	0.51	22 Oct 2015		22 Oct 2015 by system	
/resources/images/arrow_orange_down.png	0.44	22 Oct 2015		22 Oct 2015 by system	
/resources/images/Avoka_Icon_Set_16.png	16.04	22 Oct 2015		22 Oct 2015 by system	
/resources/images/Avoka_Icon_Set_24.png	1.55	22 Oct 2015		22 Oct 2015 by system	
/resources/images/Avoka_Icon_Set_32.png	10.91	22 Oct 2015		22 Oct 2015 by system	
/resources/images/Avoka_Icon_Set_blue.png	9.68	22 Oct 2015		22 Oct 2015 by system	
/resources/images/corner.png	0.22	22 Oct 2015		22 Oct 2015 by system	
/resources/images/cvv-amex.png	64.40	22 Oct 2015		22 Oct 2015 by system	
/resources/images/cvv-visa.png	45.99	22 Oct 2015		22 Oct 2015 by system	
/resources/images/get_adobe_reader.png	4.71	22 Oct 2015		22 Oct 2015 by system	
/resources/images/govassist_logo.png	12.80	22 Oct 2015		22 Oct 2015 by system	
/resources/images/icon-person.png	0.39	22 Oct 2015		22 Oct 2015 by system	
/resources/images/icons-16x16/help.png	0.45	22 Oct 2015		22 Oct 2015 by system	
/resources/images/login_icon.png	0.71	22 Oct 2015		22 Oct 2015 by system	
/resources/images/nav-icon.png	0.14	22 Oct 2015		22 Oct 2015 by system	
/resources/images/pdf-print.png	0.86	22 Oct 2015		22 Oct 2015 by system	
/resources/images/Portal_large_icons.png	18.53	22 Oct 2015		22 Oct 2015 by system	
/resources/images/Portal_logo.png	9.91	22 Oct 2015		22 Oct 2015 by system	
/resources/images/tick.png	0.52	22 Oct 2015		22 Oct 2015 by system	
/resources/images/transact.png	8.19	22 Oct 2015		22 Oct 2015 by system	

New | Restore Base Content | Close

The Portal Resource has been successfully saved.

Portal Resource | Resource History

Path *

/resources/faviconSmile.ico

Content

1

File Upload

Browse... No file selected.

Save | Restore | Close

2. Edit the Space Page “Border Template” and set the path to the new fav icon image:

Note: this only affects the space, and it will not work if you want to directly render a form.

Related articles

- Upgrading a Form Space (Portal)
- How to add a "favicon" to a Form Space (Portal)
- Deploying and customising a new Form Space (Portal)

How to Change Transact Manager Database Password

Unknown macro: 'redirect'

This how to article provides the means to change the password for a Transact Manager database.

For Windows Systems

Generating a new database(DB) password

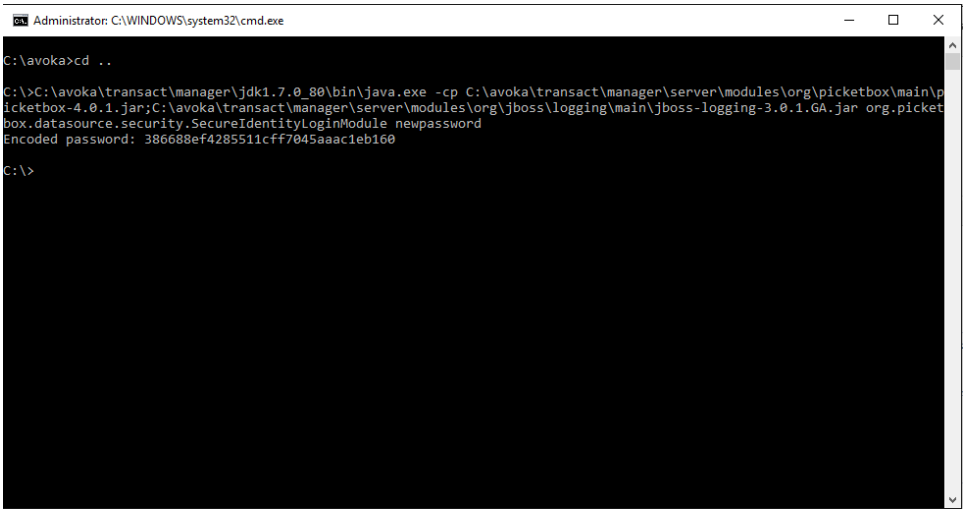
For security reasons Transact manager stores the DB password in it's configuration files as a hash. If you wish to change this password you will need to first generate a hash out of the new password.

1. Open Command Prompt
2. Next run the following command
   C:\avoka\transact\manager\jdk1.7.0_80\bin\java.exe -cp C:\avoka\transact\manager\server\modules\org\picketbox\main\picketbox-4.0.1.jar;C:\avoka\transact\manager\server\modules\org\jboss\logging\main\jboss-logging-3.0.1.GA.jar org.picketbox.datasource.security.SecureIdentityLoginModule newpassword

> Check the file path for the command as this may be different for your system. The path is the installation location for Transaction Manager for your system. For this example the TM file path is **C:\avoka\transact\manager**.
>
> Also check the jdk version as this may also be different for your version and may alter the file paths. For this example the jdk version is **jdk1.7.0_80**

> **For Linux**
>
> For Linux system the command line may look more like this:
>
> java -cp modules/org/picketbox/main/picketbox-4.0.1.jar:modules/org/jboss/logging/main/jboss-logging-3.0.1.GA.jar:org.picketbox.datasource.security.SecureIdentityLoginModule '*<newDBpassword>*'

3. Running this command with your **'newpassword'** will return a hash value, like this: 386688ef4285511cff7045aaac1eb160



4. **Keep note of this hash as you will need to copy it into the configuration files for TM.**

Updating the Transaction Manager (TM)

Now that you have a your new password as a hash, you can update the TM configuration files so that it will use this new password to access the TM database.

1. Navigate to your TM installation location and then find the file **standalone.xml**, located **C:\avoka\transact\manager\server\standalone\configuration**
2. Edit this file and search for the '<security-domain name=' section
3. By default there are two places that use the DB password, one for each data source (FormCenterDS and TransactionHistoryDS)

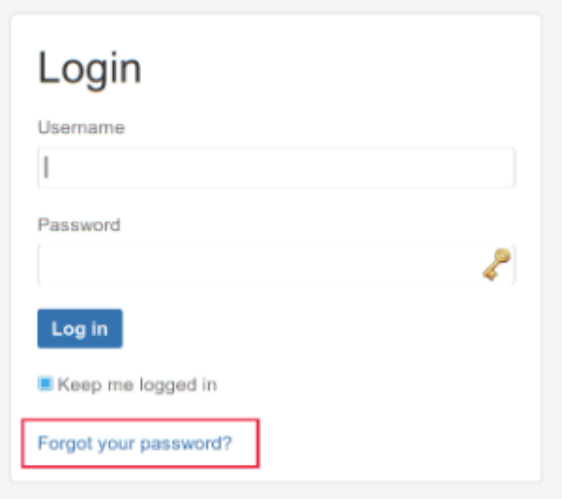
```
        </modules>
    </subsystem>
    <subsystem xmlns="urn:jboss:domain:remoting:1.0"/>
    <subsystem xmlns="urn:jboss:domain:resource-adapters:1.0"/>
    <subsystem xmlns="urn:jboss:domain:sar:1.0"/>
    <subsystem xmlns="urn:jboss:domain:security:1.0">
        <security-domains>
            <security-domain name="other" cache-type="default">
                <authentication>
                    <login-module code="Disabled" flag="required"/>
                </authentication>
            </security-domain>
            <security-domain name="sfm-login" cache-type="default">
                <authentication>
                    <login-module code="org.picketbox.datasource.security.SecureIdentityLoginModule" flag="required">
                        <module-option name="username" value="txmanager"/>
                        <module-option name="password" value="5dfc52b51bd355539bd9a0b76da9b8e7"/>
                    </login-module>
                </authentication>
            </security-domain>
            <security-domain name="history-login" cache-type="default">
                <authentication>
                    <login-module code="org.picketbox.datasource.security.SecureIdentityLoginModule" flag="required">
                        <module-option name="username" value="txmanager"/>
                        <module-option name="password" value="5dfc52b51bd355539bd9a0b76da9b8e7"/>
                    </login-module>
                </authentication>
            </security-domain>
        </security-domains>
    </subsystem>
    <subsystem xmlns="urn:jboss:domain:threads:1.0"/>
    <subsystem xmlns="urn:jboss:domain:transactions:1.0">
        <core-environment>
            <process-id>
```

4. The name of the domains is **sfm-login** and **history-login**. You will need to replace **BOTH** these passwords with the new encrypted one as each of the domains use the same credentials.

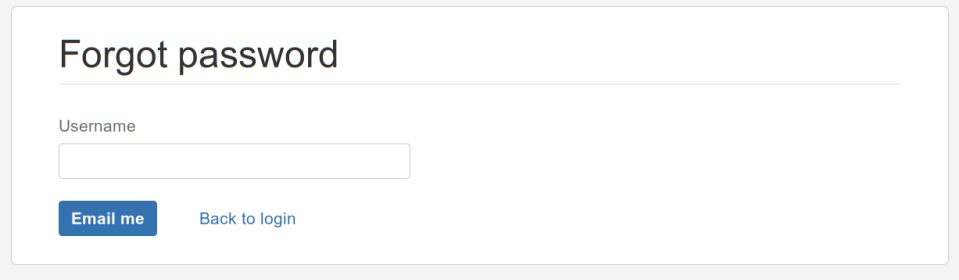
How to reset Customer Care Portal Password

Unknown macro: 'redirect'

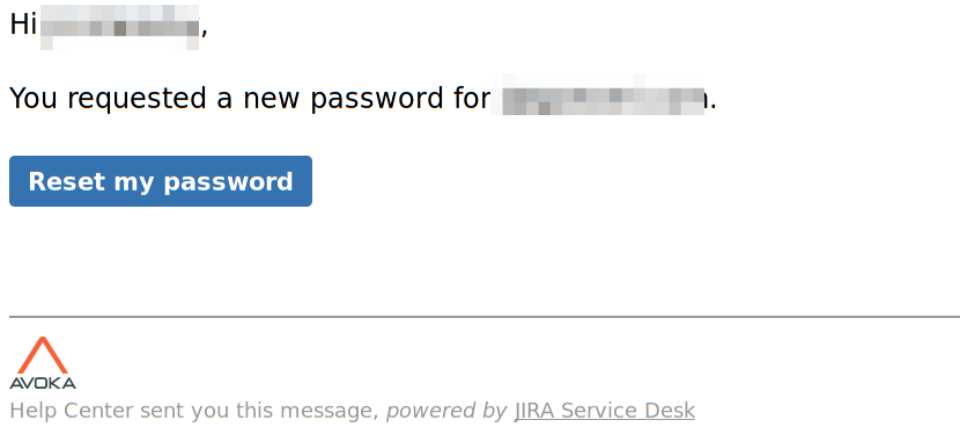
1. Go to Customer Care Portal
2. Click on **Forgot your passport?.**



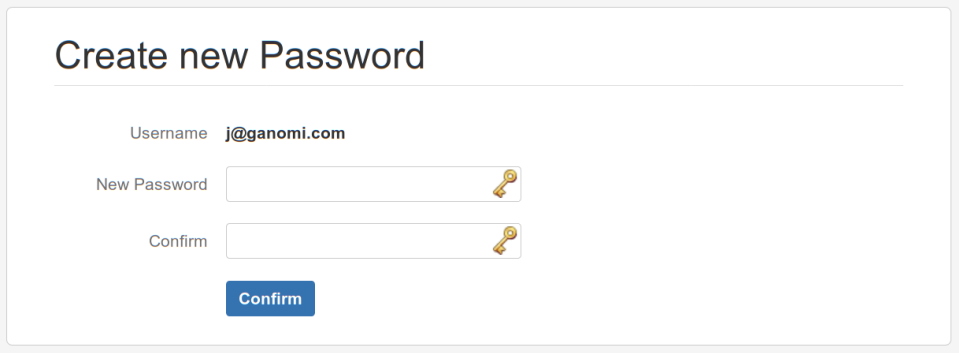
3. Enter your username (email address) and press **Email me** button.



4. A link to reset your password will be send to your email address. Click on **Reset my password button** in the email.



5. Enter your new password.



Note: If you have issues seeing these fields, either clear your browser cached data or use Private (Incognito) window.

6. You password has been reset and you should be logged into the Customer Care Portal automatically.

Managing Content Security Policy (CSP) Settings

Unknown macro: 'redirect'

The Content Security Policy (CSP) response header helps you reduce XSS risks on modern browsers by declaring what dynamic resources are allowed to load via a HTTP Header, thereby protecting your solution from a range of vulnerability attacks.

By default, Avoka Transact has strict CSP settings configured. Occasionally you need to include a capability or 3rd party service in your forms that require changes to the default CSP settings. This article describes how to manage these settings in Transaction Manager in order to allow your solution to operate as expected yet still maintain a strong CSP configuration.

Compatibility

Since	
Deprecated	

Step-by-step guide

CSP configurations can be modified at a system wide level, and/or at the organization level. The following sections describe how to achieve each of these configuration options.

System Wide Settings

1. As a global administrator, in Transaction Manager go to the **Services** menu and select **All Services**
2. Filter the list to display only services of type **Form Submission Access Controller** - there should only be one item of this type
3. Open the service and on the **Parameters Edit** page modify the **Form CSP Header Value** as required and Save

Organization Level Settings

1. As an organization administrator, in Transaction Manager go to the **Forms** menu and select **Org anizations**
2. Open your organization and go to the Security tab
3. Under **Forms Content Security Policy (CSP)**, select the **Override System Defaults** checkbox then **Save**
4. Modify the value of **CSP Security Header** as required and **Save**

LinkedIn Example

The LinkedIn SDK requires a JavaScript library to be loaded from their remote server. The default CSP settings are usually something like

```
script-src 'self'; object-src 'none'
```

These settings permit scripts to be loaded from the same server, but no other. To permit the remote LinkedIn JavaScript file to load we must modify these setting to allow JavaScript sources from the relevant LinkedIn domains as follows:

```
script-src 'self' https://platform.linkedin.com https://www.linkedin.com
'unsafe-inline'; object-src 'none';
```

Due to the way the LinkedIn integration uses inline JavaScript we need to add the 'unsafe-inline' directive.

Note that when we relax these settings, we are only adding known and trusted domains to the configuration, as opposed to using wildcards which could compromise this security layer.

References

- https://content-security-policy.com/
- Transaction Manager Administration Guide - Organization Configuration

Related Articles

- Adding a security question in Maguire forms

- Composer Security V4.2
- How to setup a new Role in Transaction Manager
- Managing Content Security Policy (CSP) Settings

On-premise Composer Upgrades

Unknown macro: 'redirect'

On-premise Composer installations will require regular upgrades and updates to take advantage of new features and defect fixes. This KB article describes the process to upgrade the Composer installation on Windows.

Step-by-step guide

Composer upgrades require the deployment of a new WAR file in the JBoss application server. The WAR file can be downloaded from https:/files.avoka.com, you can contact the Avoka support team for get access. The upgrade will automatically upgrade the database if required, it is therefore important to create a full back of the database before a new WAR file is deployed. Please consult the Composer installation manual for backup instructions.

For updating composer war file:

1. Backup composer database
2. Backup old the avoka-formcomposer.war from [JBOSS_HOME]\server\default\deploy\,
3. Shutdown JBoss for Composer
4. Copy new avoka-formcomposer.war to [JBOSS_HOME]\server\default\deploy\, replacing the old avoka-formcomposer.war
5. Delete [JBOSS_HOME]\server\default\tmp folder
6. Start JBoss for Composer
7. The database upgrade process will be kicked off once the JBoss application server has been started.
8. You will now be able to access Composer as per usual

To rollback the upgrade please follow the following steps:

1. Shutdown JBoss for Composer
2. Restore the database
3. Copy the backed up avoka-formcomposer.war to [JBOSS_HOME]\server\default\deploy\
4. Delete [JBOSS_HOME]\server\default\tmp folder
5. Start JBoss for Composer
6. You will now be able to access Composer as per usual
7. Contact the Avoka support team and provide them with the relevant JBoss log files for further analysis

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Addresses
- Address Block Samples

So you've made a mess of your development environment? Here's how to clean up!

Unknown macro: 'redirect'

At Avoka, we actively encourage all customers to put controls around development and promotion of forms to a live environment, through the use of multiple development, testing and staging (as well as the production) servers. Having separate environments means you can adequately test your work, and promote to live without affecting other services. Unfortunately, this doesn't always happen, and below is an example of a situation where Avoka have assisted in solving a confusing and non structured UAT and Production environment mess.

Imagine this situation, you've just purchased Transact and have built some forms. You didn't worry about purchasing a non production environment just now, because you don't have anything live, so you want to just go live with the whole system and your new forms.

Avoka recommends purchasing your test and production systems together, so that a clean go-live can occur, however in this scenario, we were not able to take advantage of a test server.

Later, you decide it would be good practice to get a UAT machine and start developing forms and services on that. More developers come in, publishing forms to UAT from composer, creating copies of forms, publishing more things to UAT. Before you know it, your forms list looks like this:

"Credit Card Application"

"Credit Card Application (form)"

"Credit Card Application 2"

"Credit Card Application test"

"Home Loan Application"

"Home Loan Application (UAT)"

"HL Application"

etc etc etc

So which one is the UAT proper version that you should export to production when you are ready?? What if you have 100s of forms and it looks like this?

Don't let it get to this!

Firstly, don't let it get to this stage. We can do this by:

1. In Composer, Implementing a naming or versioning standard
    a. I.E – all revisions of forms that are the current working copy should be denoted using "Current Version" for the Revision Name
    b. Use the versioning tool within Composer and create your own processes to identify forms
2. Set up your composer security correctly
    i. Developer
    ii. Account Administrator – able to unlock forms and add users/change passwords.
3. Setting up the composer development environment
    i. Have one folder per developer to be used as a sandbox area
    ii. Create a "Backup" folder – This folder will contain a list of forms, denoted by the production form name, with the date appended that it was moved to the backups folder
    iii. Production – production forms only
    a. Process for transferring forms:
        i. If changes are required, copy the current production form to the relevant sandbox project
        ii. Make form changes
        iii. Test changes
        iv. Backup the production form to Backup project using the "Save As" functionality within Composer
        v. Then update the "Production" project with the newly updated form
4. Publish to TM.
5. Ensuring that no one can publish directly to the Production TM environment from the Composer environment.
6. In TM, Implementing a Form Code standard. For example, for a credit card application you may use the syntax – <BRAND CODE><FORMCODE> which could look like MAGCCA (where the brand is Maguire, and the form code for credit card application is CCA). Each form should follow this standard and this standard will depend on the makeup of your business. You may have geographical jurisdictions instead of brands so your code may be <GEOGRAPHY><FORMCODE> for example, AUSCCA (Australia), or UKCCA (United Kingdom).
7. Ensuring any debug or sandbox forms in UAT have the "Test Mode" flag set.

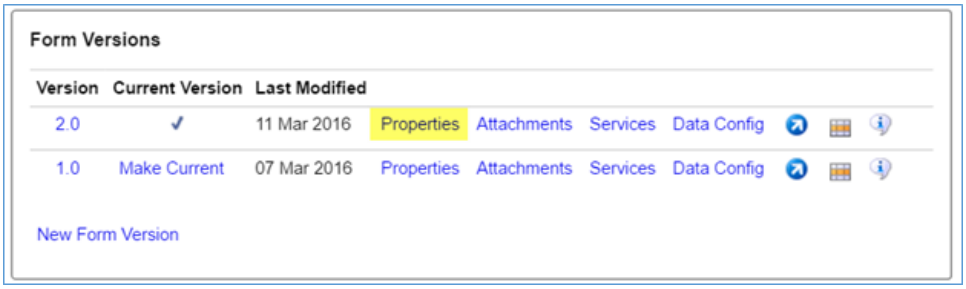
8. Lock your permissions so only those people who need to publish forms to UAT or Prod have the access to (Avoka have a set of recommendations on this).

To correct this issue - Step-by-step guide

1. You will need to set up some type of matrix using a spreadsheet in order to take note of the form name and code, and the corresponding MD5 value, and which server this exists on.
2. Remove connectivity between Composer and the Production Environment to ensure no forms can be publish to production directly from Composer.
3. Match the Form Archive MD5 value of Prod forms to UAT forms (these forms are evidence that forms have been correctly exported from UAT, and imported to production

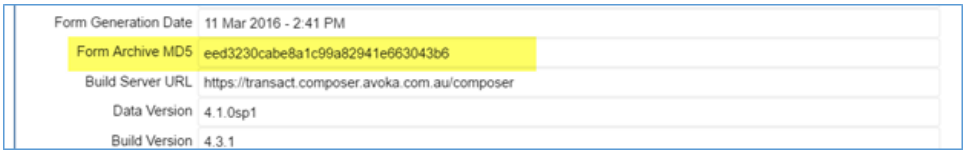
   Go to the Transaction Manager Administration Console

   Go to Forms, Forms, and find the form in question
   Click on “Properties” for the form version in the form dashboard



   Choose the “Form Archive Info” tab

   Take a note of the “Form Archive MD5” Value



   Match this value to the suspected corresponding form in production
4. Where forms are in UAT and also appear to be in Prod, but the Forms Archive MD5 does not match, this is evidence that the form has been published directly to production.
5. An option in this case is to export the form from production, and import it into UAT. Then you can publish from Composer to that form in UAT
6. Once satisfied that the forms in prod are correctly reflected in UAT and Composer, forms in UAT can be made “inactive”. Likewise superfluous forms in Prod can be made inactive
7. After a period of time, forms in prod that are inactive can be deleted.

If these steps do not work for your particular problem, please give your Avoka Assist contact a call and they can assist you further.

Related articles

- Managing multiple service endpoints and credentials for external service calls
- So you’ve made a mess of your development environment? Here’s how to clean up!

Transact Manager on-premise deployment notes/check list

Unknown macro: 'redirect'

Transact Manager Web Server Requirements

Security

It is recommended that at least a SHA256 2048 bit ssl certificate is used and the web server configured to only supports TLS v1.0 or later and avoid the use of weak ciphers.

It is recommenced that a security module (such as mod_security for Apache) is used to implement an additional security layer. Avoka suggest the following settings

- Set the HTTP Request Body size and file upload size allowed to minimum of 52428800 bytes (50MB);
- Disable TRACE method
- Enable the Strict Transport Security Header with "max age=16070400; includeSubDomains"
- Hiding the Web Server info (on Apache set ServerSignature OFF; ServerTokens PROD)

Caching and compression

It is recommended that the Web Server is setup to cache static content and also to enable compression of uncompressed files.

Reverse Proxy

Avoka Transact Manager required to have a reverse proxy configured.

TM Behind a Proxy Server

When TM is deployed behind a proxy server , that need to be used for outbound connections, you need to add the proxy details to the following sections with in the TM Standalone.xml file:

```
<system-properties>
      <property name="org.apache.catalina.connector.URI_ENCODING" value="UTF-8"/>
      <property name="org.apache.catalina.connector.USE_BODY_ENCODING_FOR_QUERY_STRING" value="true"/>
      <property name="http.proxyHost" value="1.2.3.4"/>
      <property name="http.proxyPort" value="8080"/>
      <property name="https.proxyHost" value="1.2.3.4"/>
      <property name="https.proxyPort" value="8080"/>
      <property name="http.nonProxyHosts" value="localhost"/>
  </system-properties>
```

TM Behind a LoadBalancer

When TM is deployed behind a load balance the following may need to be setup:

- Enable sticky sessions for JSESSIONID (the JSESSIONID is generated by Avoka-TM)
- Use the following pages for any TM node health checks from the load balancer => https://[tmservernode]/[portalname]; or http://[tmservernode]/[portalname]
- Ensure the LoadBalancer forwards the X-FORWARDED-FOR header info which contains the original source IP address to the back-end servers. This can be verified by enabling the Apache to output the X-FORWARDED-FOR header info into its access_log file => please use below in your Apache config file "httpd.conf" & "ssl.conf":
    - On the httpd.conf:

```
...

LogFormat "%h %{X-Forwarded-For}i %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" proxy

SetEnvIf X-Forwarded-For "^.*\..*\..*\..*" forwarded

CustomLog "c:/apache2.2.29/logs/access.log" proxy env=forwarded

 ...
```

    - On the ssl.conf:

```
...
<VirtualHost ... >
...
LogFormat "%h %{X-Forwarded-For}i %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" proxy
SetEnvIf X-Forwarded-For "^.*\..*\..*\..*" forwarded
CustomLog "c:/apache2.2.29/logs/ssl_access.log" proxy env=forwarded
 ...
</VirtualHost>
```

Troubleshooting Tips

TWeb Attachments upload or TField Synchronization Issues

Check the HTTP Request Body size settings on all paths to the TM Server. Attachments and TField uploads are large and if they are blocked this will typically cause problems with attachments or TField Synchronization.

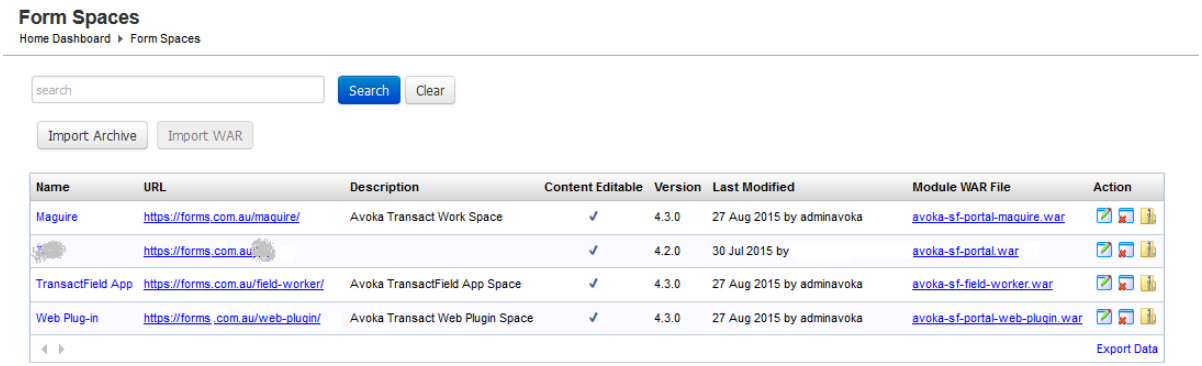
F5's have also been seen to block TField uploadSubmission, uploadDraft and uploadAttachment operations but return a HTTP 200 result. This results in TField sync errors.

Upgrading a Form Space (Portal)

Unknown macro: 'redirect'

When a Transact Manager system is upgraded is it important to make sure that the Spaces get rebuilt. For a system that has content only Spaces (not custom Java Spaces) this is a relatively simple but necessary step as the Space contains the base page mark-up which is used in TM by the page and resource merging facilities. If the war files are not rebuilt to represent the TM version you are upgrading to then you will not see any changes to the mark-up that have been made as part of the product. These may include new features/bug fixes or security patches.

After a TM upgrade each Space in the Space/Space list should have a version number which matches the version of TM you are upgrading to. The following is an example screen shot that shows a system that has been upgraded from 4.2 to 4.3 but the main Space has not been rebuilt.



We have seen instances where security scans have raised issues because the Space mark-up has not been merged in with the new mark-up for the later TM version. Please see the 'Styling and Mark-up' section below.

Please see Managing Form Spaces in the product documentation for more comprehensive information relating to building portals.

Rebuilding Spaces

Content Only Spaces

TM will automatically upgrade the standard TWeb and TField Spaces (Maguire, TransactField App, Web Plug-in), any other Space needs to be rebuilt manually. Content only TWeb Spaces can easily be built using the TM Installer. At the moment TField Spaces need to be manually prepared and there are some instructions in the TM install guide to help with this.

Custom Java Spaces

Custom Java Spaces are more complex and the Spaces need to be manually prepared and the code level customisation need to be merged into a clean Java Space for the version of TM you are upgrading to. This can be complex and time consuming and is beyond the scope of this article. You will need to refer to the original builders of the Space for help with this aspect of the re-build.

Important Points when re-building and deploying a Space :

One of the key points when upgrading a Transact Manager Space is that you want to replace the old Space with a new Space and maintain the forms and user relationships with in TM. The following are a few important points to follow to make sure this happens:

1. Make sure the Space war file name stays the same.
2. Make sure the Space context stays the same.
3. Space Name needs to be the same. This is set in this TM installer when creating a TWeb Space or directly in the db-config1.xml file's portal element (<portal name="testPortal">) if you are manually building a Space .
4. Replace the Space in the deployment directory. On a single node system you can deploy the war file through the TM admin console.
5. Merge in the styling and mark-up changes.

Styling and Mark-up

Merging in styling and mark-up is an important step and this can be time consuming if the styling or mark-up as moved far away from the standard product set.

1. If the Space war includes all the styled pages then hit the restore base content button on the resources and pages tabs within the Space config in TM. Beware - This will replace all styling cached in the TM database.
2. If the styling is in TM then you have to merge in the pages and resources manually within the TM console.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Interacting with TransactField using Custom URLs
- How to rescue attachments from your iOS device
- How to pre-fill static map images into TransactField Tasks
- Determining the IP address that a form is requested from

Operations

- Hosting Other Documents in Transact
- How to Request a new login to Customer Support
- How to rescue transactions that cannot be delivered
- How to setup a new Role in Transaction Manager
- Managing TM Alerts

Hosting Other Documents in Transact

Unknown macro: 'redirect'

Let's say you've got 400 existing forms you're migrating across to Transact.

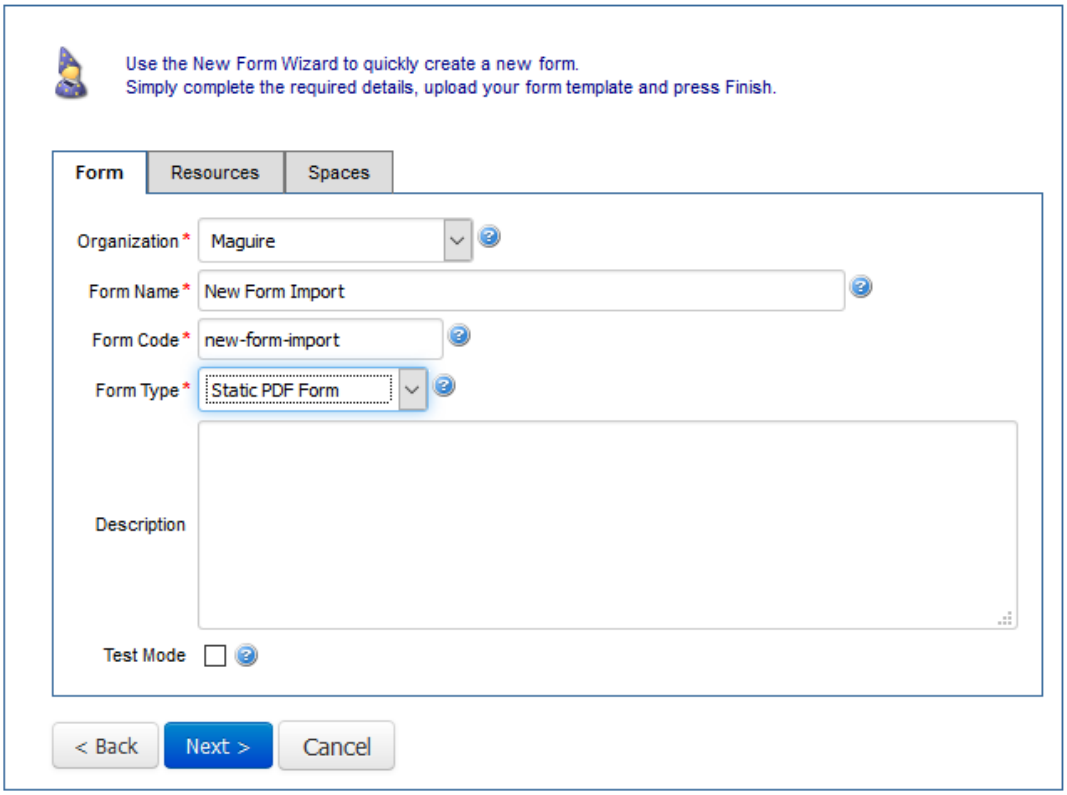
Those 400 forms include Word Documents, Excel Spreadsheets or PDFs created in InDesign or Adobe LiveCycle Designer.

While you're working on migrating your high-value forms, there's no need to manage all your forms in multiple places. You can actually host all of your existing and new forms in Transaction Manager!

Adding an existing form to Transaction Manager

You can add a form to Transaction Manager using either Forms -> Forms -> New or Forms -> New Form Wizard.

In either case, you will be prompted to select a Form Type.



Forms published from Composer are called Composer SmartForms.

But Transact can also serve up to users Dynamic PDF Forms from LiveCycle Designer, Static PDF Forms from InDesign or Acrobat, and Other Documents (Word, Excel, etc.)

More configuration and setup is required to allow users to submit Dynamic PDF Forms through Transact, but at the very least each of these Form Types can be hosted in transact and displayed to the user.

Related articles

- Task Assignment
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer

- How to create a Library in V4

How to Request a new login to Customer Support

Unknown macro: 'redirect'

If someone in your organization needs a new login to the Customer Center:

Step-by-step guide

Send an email to customercare@avoka.com, with the new users name, email address, and phone number.

Note there is a limit of 4 users per organization, so you may need to remove one in order to add

Related articles

- How to Request a new login to Customer Support

How to rescue transactions that cannot be delivered

Unknown macro: 'redirect'

Sometimes things don't go as planned, and you might recognise the following scenario:

Earlier this week you deployed the form change that was requested, it all tested and you were quite happy that it all went smoothly. Until two days later, when you find out that your form change caused a downstream system to fail and all submissions for these two days can't be processed anymore. Quickly restoring the previous form version will at least stop the bleeding, but what to do with the submissions that are "stuck"? You know how to fix them up, a change to the delivery process would be simple. Unfortunately this change would also impact the current submissions, and a change would require regression testing and all kinds of approvals.

Step-by-step guide

1. The easiest solution at this point is to create a new delivery channel within the same organisation, let's call this one "fixer upper delivery channel",
2. For this new delivery channel you can write the Groovy code that's necessary to "massage" the data for the stuck transactions so that the downstream process can handle it.
3. We assume that you have a list of tracking codes of the stuck transactions
4. From the Groovy console you can run the following code that will change the delivery channel on these transactions and make them ready for delivery

```
import com.avoka.fc.core.dao.SubmissionDao
import com.avoka.fc.core.dao.DeliveryDetailsDao
import com.avoka.fc.core.dao.ClientDao
import com.avoka.fc.core.service.DeliveryResult.Status
import com.avoka.fc.core.service.ServiceFactory
import com.avoka.fc.core.entity.Submission

// create your collection of tracking codes
String[] trackingCodes = ["TC-1", "TC-2", ...]

// Find the client and then create a new DeliveryDetails object for the delivery channel for that client
client = new ClientDao().getClientByCode("Your Organisation code");
deliveryDetails = new DeliveryDetailsDao().getDeliveryDetailForClientAndName(client, "fixer upper delivery
channel")

// we need to get the submissions using the SubmissionDa
sDao = new SubmissionDao()

for (String trackCode in trackingCodes) {
    submission =  sDao.getSubmissionByTrackingNumber(trackCode)
    if (submission) {
                println submission.formName + " found";
                submission.setDeliveryDetails(deliveryDetails)
                submission.setDeliveryStatus(Submission.STATUS_Ready)

                    println submission.trackingNumber  + " changed to new delivery process"
          }
}
ServiceFactory.getSubmissionService().commitChanges()
```

Related articles

- How to mix standard delivery functions with custom groovy processing
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating new form tasks in a Groovy delivery service
- Creating anonymous tasks via Groovy service

How to setup a new Role in Transaction Manager

Unknown macro: 'redirect'

Transaction Manager comes with a super administrator role with all permissions related to the Transaction Manager management console. You can create new roles and modify the existing ones to suit your organisation needs.

Standard Roles
When Transaction Manager starts up for the first time (or whenever there are no roles defined in the Transaction Manager database), it will automatically create a set of default roles: The root administrator role with all permissions, a form developer role, an operations monitoring role, an organization user manager role and a system support role. You can customize these roles to suit your specific needs, or set up new roles entirely.

Root Administrator
This is the most powerful role in Transaction Manager. Root administrators have all permissions related to the Transaction Manager management console assigned to them and can access all restricted content. Critical operations such as deleting organizations and editing deployment properties should be accessible to these users only.

Form Developer
Form developer users have a range of permissions related to their need to create, configure, test and debug forms. They can edit organizations, forms and all related entities and view requests, submissions and associated data (included submitted XML data).
Form developers cannot access some areas of the system such as deployment properties and system information. They can also be restricted to a single organization.

Operations
Organization operation administrators can view submissions (included submitted XML data) and reports as well as various logs so they can handle support requests submitted by portal users.
However, they cannot access forms and associated configuration. They cannot access the details for user accounts stored in Transaction Manager.

Organization User Manager
Organization user administrators can create and configure users for their organization. This is a role that might be used in addition to other roles to give existing administrators access to user accounts, though it can also be used as a stand-alone role.

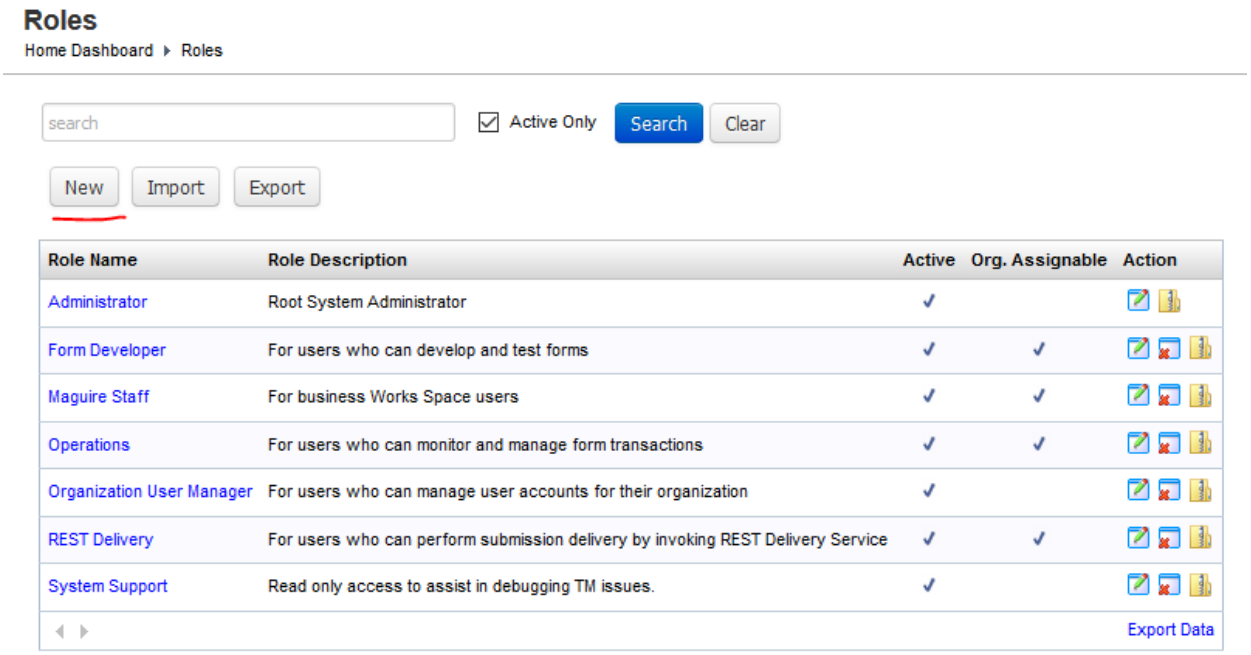
System Support
The system support role can be used by Avoka staff for debugging purposes. By default, the role has extensive read-only permissions to aid in debugging. However, XML data submitted by users cannot be viewed.

To create a new Role in Transact Manager

1. Login to Transact Manager

2. Go to "Security > Roles".

3. Click the New button

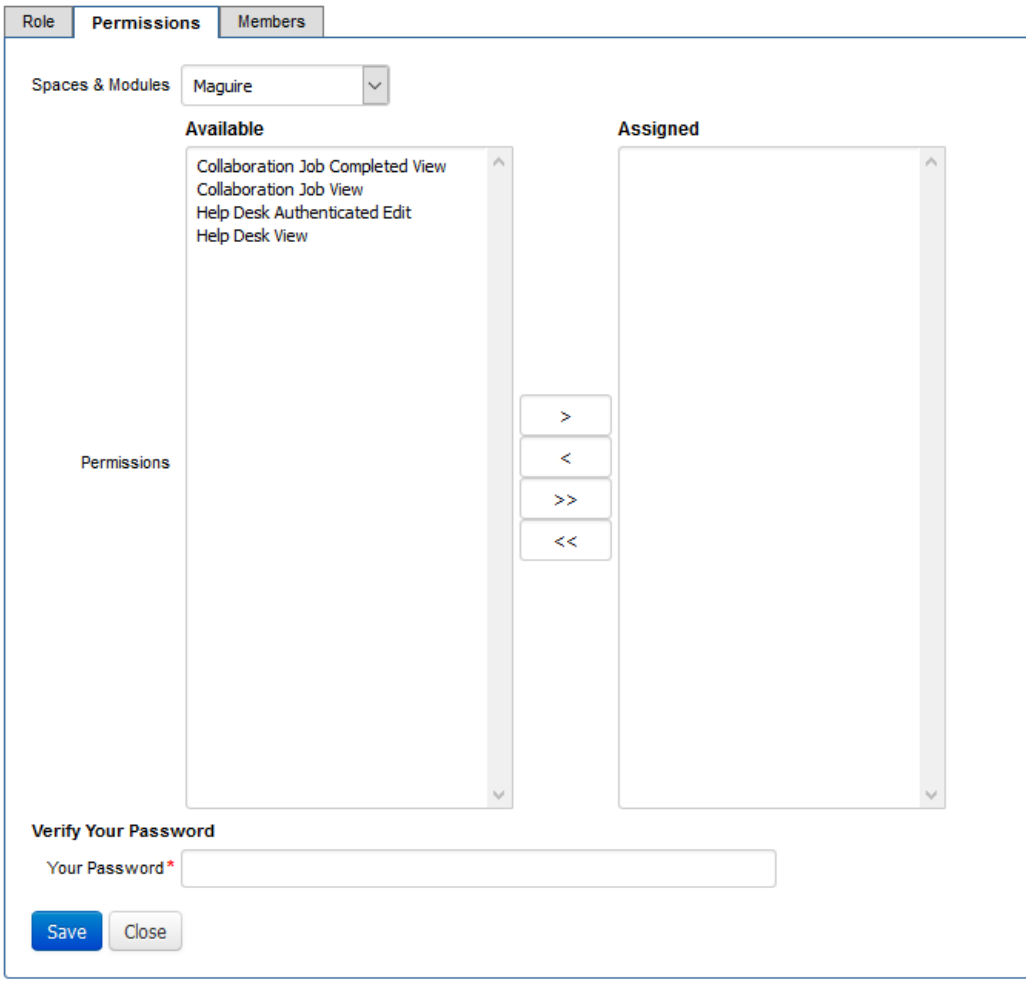


4. Fill in a unique role name and an optional description.

5. Ensure the "Active" checkbox is ticked. Inactive roles cannot be assigned to users.

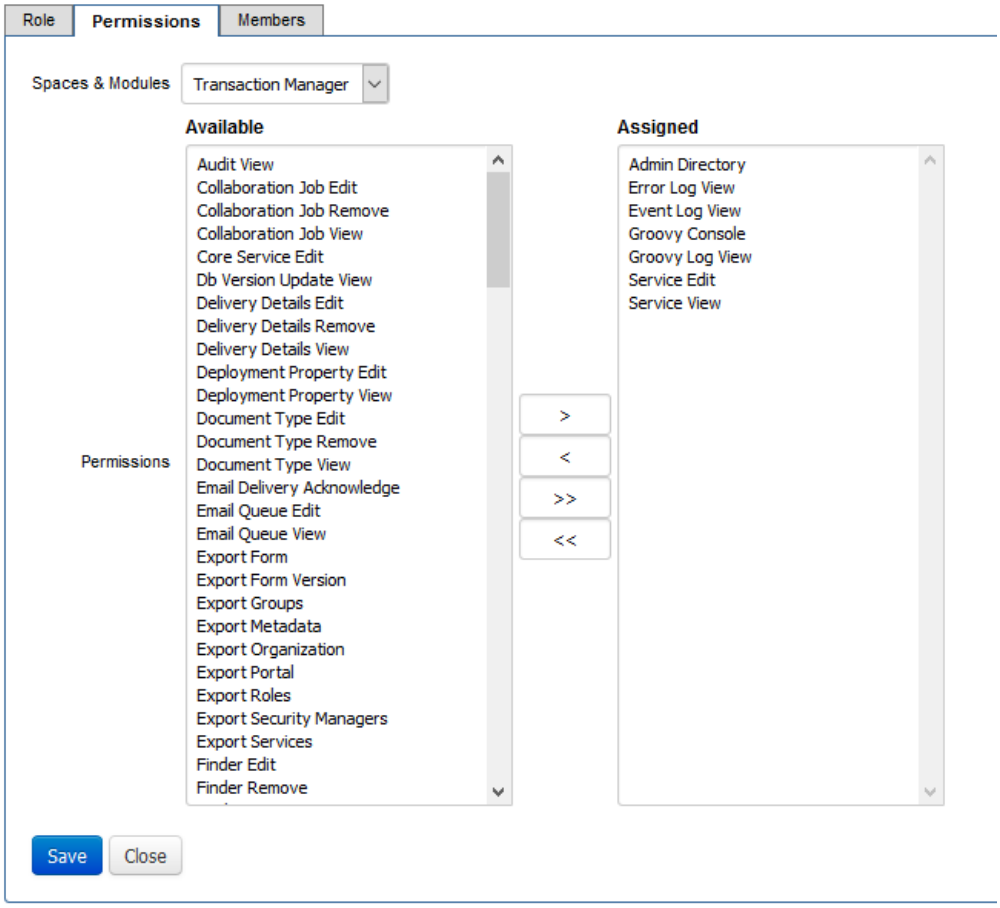
6. Tick the "Organization Assignable" flag only if you want organization administrators to be able to assign that role to other administrators belonging to the same organization. If you leave it unticked, even organization administrators with the permission to assign roles will be unable to see or choose the role. For example, a global administrator role with all privileges will most likely not be organization assignable.

7. Save the role. Now you can assign permissions to the new role.



Setting Up Permissions for a Role

To view or change the permissions assigned to a role, go to "Security > Roles", edit the role and switch to the "Permissions" tab

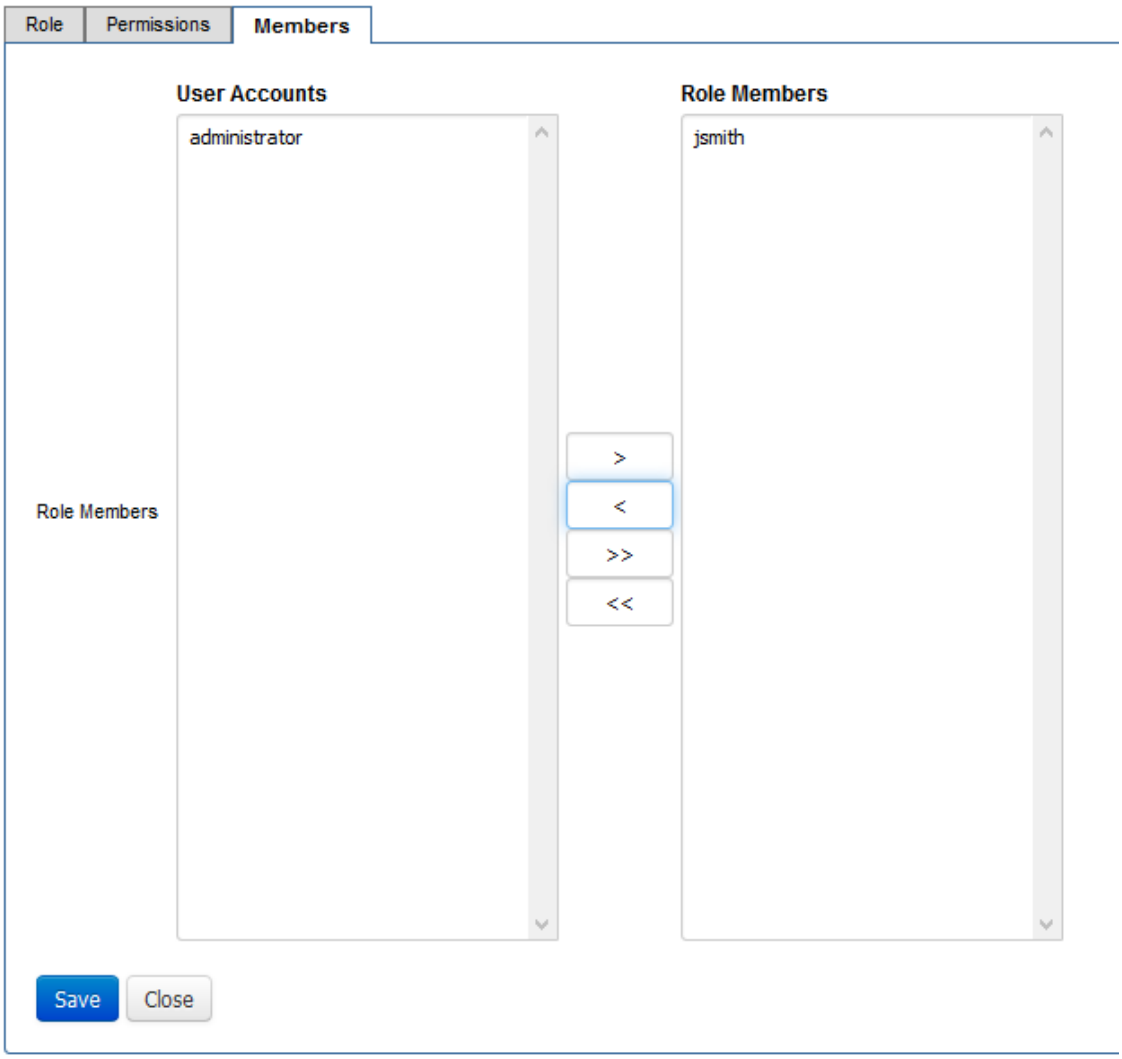


8. Choose a portal from the "Portal" dropdown. The "Permissions" list will then be updated with permissions belonging to the selected portal.
To change the permissions for a role, select permissions (on Windows, use the Shift and Ctrl keys to select multiple permissions) and use the ">" and "<" buttons to assign and revoke permissions.
You can also add all available permissions to a role by using the ">>" button; likewise, the "<<" button will remove all permissions.

9. Once all permissions have been assigned appropriately, save the role.
Note: The permissions assigned to a role are loaded when the user logs in, so changes will be visible to users after logging out of the portal and logging in again

Managing the List of Role Members

You can view and modify the list of role members (users who have been assigned the role) by going to the "Members" tab.



Related articles

- Adding a security question in Maguire forms
- Composer Security V4.2
- How to setup a new Role in Transaction Manager
- Managing Content Security Policy (CSP) Settings

Managing TM Alerts

Unknown macro: 'redirect'

This article describes how the manage the list of users who will receive operational email alerts that are send out from Transaction Manager to report various operation errors.

TM Alert Groups

1 - Promotion Alerts

Event = Form promoted, Context = Form Organization Users and Global Users

User to notify when a form belonging to their organization is promoted (For more information see TM Administration Guide Section 4.12.1).

2 - Delivery Escalation Alerts

Scheduled Job = Delivery Escalation, Context = Transaction Organization Users and Global Users

Users who receive alerts when a transaction delivered by secure email has not been acknowledge yet (For more information see TM Administration Guide Section 7.2.2).

3 - Submission Updates

Event = WebService Call = FormsServerWebService::UpdateProcessingStatus, Context = Transaction User

Users who will be notified when the processing status for a submission they have made has been updated. This group is relevant only to space users, not administrators.

4 - System Alerts

Event = Internal Exception Raised, Context = Global

Users who will be notified when the System Monitoring job finds that important system jobs are impacted (For more information see TM Administration Guide Section 22.3.14).

5 - System Alerts - Delivery

*Event = Exception during delivery, Context = Transaction Organization Users but **NOT** Global User who are not part of the Organization*

Users who will be notified when submission delivery fails.

6 - System Alerts - Receipts

*Event = Exception during receipt generation, Context = Transaction Organization Users but **NOT** Global User who are not part of the Organization*

Users who will be notified when submission receipt generation fails.

7 - User Account Creation Alerts

Event = Account Creation, Context = All Users in Group but NOT Organization Specific

Users who will be notified when a user has applied for an account on a user space (For more information see TM Administration Guide Section 22.9.3.5).

8 - Outstanding Payment Alert

Event = Scheduled Job = Payment Reminder, Context = Users in group and their scope including global users

Event = Fraud & Duplicate Payment Detection, Context = Transaction

Users who receive email alerts reporting transactions that appear to have been abandoned at the payment stage.

Related articles

- Transaction Delivery
- Transaction Reporting Feed
- TField and Locations
- Collaboration Jobs - Rolling Back To a Previous Step (State)
- test

Transact Manager (TM)

- Could not set property 'formDataEncryptFlag' on entity TemplateVersion
- Creating and Presenting Custom Reports
- Distinguish Between "Save on Page Change" and "Timed Save"
- Handling Transact Form Function Exceptions in Maestro Applications
- How to find the Data Model for Form Space (Velocity) Pages
- How to Log Out of Transact Manager from the Client
- PDF Receipts
- Prefill

Could not set property 'formDataEncryptFlag' on entity TemplateVersion

Unknown macro: 'redirect'

Problem

When exporting a Form from a more recent version of Transact Manager to an older version, you may receive the error message *Could not set property 'formDataEncryptFlag' on entity TemplateVersion*. See video below for steps that replicate this issue which was experienced when exporting a Form from version 5.0.0 of TM to a version 4.3.4 of TM.

Solution

As we're exporting between two different versions of TM - the original form has this flag set on it but it doesn't exist in the new environment so it can't be set. This should not stop the form from functioning

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Addresses
- Address Block Samples

Rendering a newly imported Form generates an IllegalArgumentException error

Unknown macro: 'redirect'

Problem

Per the video below, when importing a Form that has come from a more recent version of TM (in this example, we are importing a Form created in TM 5.0.0 into TM 4.3.4) you may get an IllegalArgumentException error.

Solution

In Transact Manager of the system you have imported the Form into, navigate to **Forms > Forms** then click on the **Current Version** number of the Form. Change the **Form Type** to **Composer Smart Form** and this should allow you to now render the Form.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaboration Jobs - Separate out emails from Task Assigns using multiple actions
- Collaborative form completion with anonymous users (form sharing)
- Addresses

Creating and Presenting Custom Reports

Unknown macro: 'redirect'

A quite common requirement at the end stages of a project is to create "on-demand" reports on the data within TM. Reports help management and operational users clearly see data that could even be PII data. One benefit of creating reports that are presented within a formspace is that we are not responsible for the delivery of PII data. The results are displayed on screen or within a CSV file and it's up to the end user to safely and securely download and store the file. Reports can be heavily customized to show data from submission objects, data extracts, or form xml. Remember that the form xml will be purged based on data retention settings. Reports can also be quite processor heavy so special care should come with load testing.

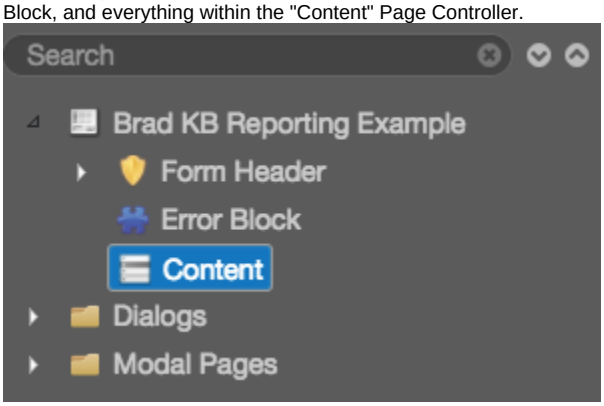
Let's walk though a very simple report that uses the Core API to query submissions. Typically I use the Core API because the Fluent API doesn't have a start and end date as part of the querying criteria. Start and end dates are a standard for most reporting criteria. Most reports are based on submissions and the all DAOs query off of submission date except for the TransactionHistoryDAO which queries off of form start/opened date.

The following example only shows how to display 1 report but the architecture is setup in a way to easily add more reports.

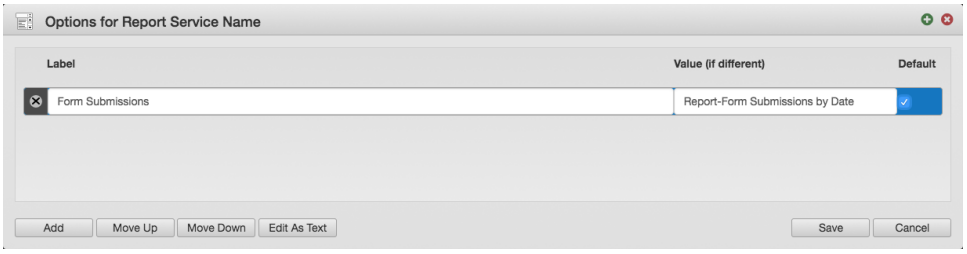
Form Development Step-by-step guide

After creating a new blank form with your customer's branding:

1. Delete all non-essential components, especially the submit and save buttons. We don't want any transactions getting submitted to TM or the customer will incur a fee.
    a. On the Maguire form I deleted: Wizard Nav Bar, Form Footer, Function Bar, Navigation Second Level, Navigation Chevrons, Job Details Block, and everything within the "Content" Page Controller.



2. Add a Page named "Reporting" to the Content Page Controller.
3. Add a Section named "Report Criteria" to your "Reporting" Page.
4. Add a dropdown named "Report Service Name" to your "Report Criteria" Section. We will only have one report but this is where you can add multiple reports if required.
    a. Add one option to the dropdown. Make the label "Form Submissions" and the value has to be the name of the Groovy Service that creates the report data. We'll call our service "Report-Form Submissions by Date".



5. Add a block called "Criteria1" to the "Reporting Criteria" Section.
    a. The reason for the block is to add visibility rules if you have different reports with different criteria.
6. Add a 2 date fields called "Start Date" and "End Date".
    a. Prefill the "Start Date" and "End Date" with today's date.
    b. Add Validation that they must be Past Dates and Include Today.
    c. Make both fields Mandatory.
    d. Add the following validation rule to "End Date" (optional):

ValidationRule

```
if (Calc.daysBetween(data.startDate, data.endDate) > 7)
        return "Date range cannot exceed 7 days.";
```

7. Add a Text Display field beside the dates that states the valid date range. i.e. "Date range cannot exceed 7 days."

8. Add a dropdown called "Output Format" with 2 options: CSV and HTML to the "Reporting" Section.
9. Add a "Run Report" button.
10. Add a Block called "OutputBlk" to house all the hidden data and output fields.
11. Add a Text Display field with "Output records have been returned." to the "OutputBlk".
12. Add a "recordsReturned", "rawData", and "outputReturned" Data Fields to the "OutputBlk".
13. Add a "Download CSV" Button to the "OutputBlk" with the following Visibility rule:

    data.outputreturned === "CSV"

14. Add a "Show HTML" Button to the "OutputBlk" with the following Visibility rule:

    data.outputreturned === "HTML"

15. At this point, all the necessary fields have been added. Now it's time to insert the logic to the fields.
16. Add the following "Click" script to the "Run Report" button:

Run Report Click

```
var runReport = function()
{
    data.rawdata = null;
    data.recordsreturned = 0;
    data.outputreturned = data.outputFormat;

    Form.showProgress("Your report is being generated...");

   var inputData = {
       startDate: data.startDate,
       endDate: data.endDate,
       reportServiceName: data.reportServiceName,
       outputType: data.outputFormat
   }
   DynamicData.call("GenerateDynamicDataReport", inputData).then(
       function(response) {
           Form.showProgress("");
           console.log(response);
           data.rawdata = response.data;
           data.recordsreturned = response.recordsReturned;
   }).catch(
       function(err) {
           Form.showProgress("");
           console.log(err);
       }
   );
}

Form.validate("data.reportCriteria").then(
   function(response) {
       if (response.valid == true) {
           runReport();
       } else {
           return;
       }
   }
).catch(
    function(err) {
        console.log(err);
    }
);
```

17. Add the following Click script to the "Download CSV" Button:

Download CSV Click

```
function requiredFeaturesSupported()
{
    return ( BlobConstructor() && msSaveOrOpenBlobSupported() );
}
function BlobConstructor()
{
```

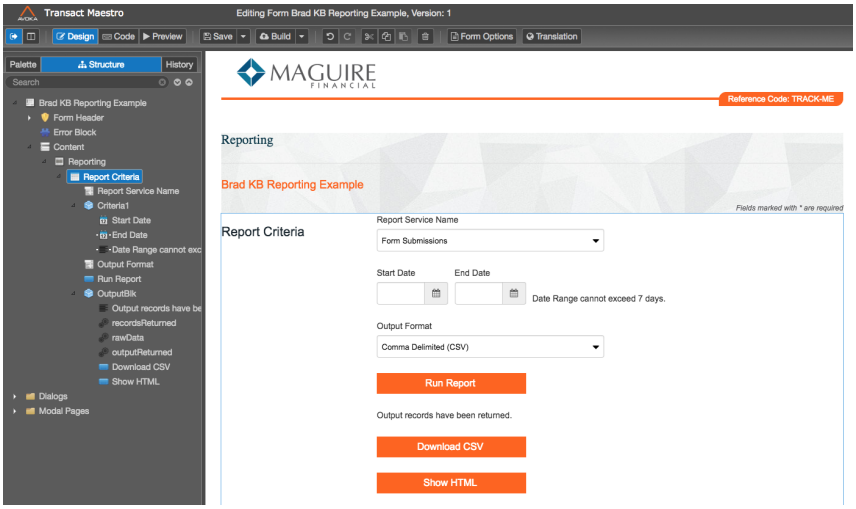
```
    return (window.Blob)
}
function msSaveOrOpenBlobSupported()
{
    return (window.navigator.msSaveOrOpenBlob)
}
function b64toBlob(b64Data, contentType, sliceSize)
{
    contentType = contentType || '';
    sliceSize = sliceSize || 512;
    var byteCharacters = atob(b64Data);
    var byteArrays = [];
    for (var offset = 0; offset < byteCharacters.length; offset += sliceSize) {
        var slice = byteCharacters.slice(offset, offset + sliceSize);
        var byteNumbers = new Array(slice.length);
        for (var i = 0; i < slice.length; i++) {
            byteNumbers[i] = slice.charCodeAt(i);
        }
        var byteArray = new Uint8Array(byteNumbers);
        byteArrays.push(byteArray);
    }
    var blob = new Blob(byteArrays, {type: contentType});
    return blob;
}
if (requiredFeaturesSupported()) {
    blobObject = b64toBlob((data.rawdata), "text/csv");
    window.navigator.msSaveOrOpenBlob(blobObject, 'report.csv');
} else {
    var a = document.body.appendChild(document.createElement("a"));
    a.download = "report.csv";
    a.href = "data:text/csv;charset=utf-8;base64," + data.rawdata;
    a.innerHTML = "download report csv";
    a.hidden = true;
    a.click();
}
```

18. Add the following code to the "Show HTML" Button:

Show HTML Click

```
if (data.rawdata) {
    var win = window.open("", "Report", "toolbar=no, location=no, directories=no, status=no, menubar=no,
scrollbars=yes, resizable=yes, width=780, height=200, top="+(screen.height-400)+", left="+(screen.width-
840));
    win.document.body.innerHTML = data.rawdata;
}
```

All done... not it's time to write the services! Here's what it should look like:



Groovy Script Development Step-by-step guide

The form is designed to call one generic DDS with the name of the report service as a parameter. There are 3 services which sort of represent a typical 3 layer java framework of "Service Layer", "Manager Layer", and "DAO Layer". Only the "Manager Layer" has to change in most situations since the Submission object that gets returned gives you access to everything you'll need.

1. GenericDynamicDataReport - This is the main DDS service that the form calls. It's just passes on the parameters to the right Groovy Service. In a perfect situation you would also handle any bubbled exceptions here too.

GenericDynamicDataReport

```
import com.avoka.core.groovy.GroovyLogger as logger
import com.avoka.fc.core.service.ServiceLocator
import com.avoka.fc.core.dao.*
import com.avoka.fc.core.entity.*
import groovy.json.*

String reportServiceName = request.getParameter('reportServiceName')
String startDate = request.getParameter('startDate')
String endDate = request.getParameter('endDate')
String outputType = request.getParameter('outputType')

def reportService = ServiceLocator.getServiceForName(reportServiceName)
Map<String,Object> result = reportService.invokeService([
    startDate:startDate,
    endDate:endDate,
    outputType:outputType
])

return JsonOutput.toJson(result)
```

2. Report-Form Submissions by Date - This is the name of the service that you configured in your Maestro form. It has to match perfectly with the value of the dropdown in "Report Service Name" and the parameters have to match as well. This service assembles the data returned from the next DAO call. This is where you can get really creative and return back submission properties, data extracts, form XML, etc... But this is where you also have to be careful with performance. The more introspecting and filtering you do the slower your report will be. The "filterData" function

Report-Form Submissions by Date

```
import com.avoka.core.groovy.GroovyLogger as logger
import com.avoka.fc.core.dao.*;
import com.avoka.fc.core.entity.*;
import com.avoka.fc.core.service.ServiceLocator
import groovy.json.*

DATEFORMAT = 'yyyy-MM-dd'

String formCode = serviceParameters.formCode  // You'll have to add a service parameter or hardcode your
formcode here.

String startDate = parameters.startDate
String endDate = parameters.endDate
String outputType = parameters.outputType
String encodeCsv = parameters.encodeCsv ?: 'false'

//logger.info "-------------------------------------------------"
//logger.info serviceDefinition.getServiceName()
//parameters.each { logger.info "$it.key: $it.value" }
//logger.info "-------------------------------------------------"

def reportHeaders = createReportHeaders()
def reportData = retrieveData(formCode, startDate, endDate)
reportData = filterData(reportData)

def output = null

if (outputType.toUpperCase() == "CSV") {
    String csvString = createCsvReport(reportHeaders, reportData)
    if (encodeCsv == 'true')
```

```
        output = Base64.getEncoder().encodeToString(csvString.getBytes())
    else
        output = csvString
} else if (outputType.toUpperCase() == "HTML") {
    StringBuilder reportHtmlStr = new StringBuilder();
    reportHtmlStr.append startHtmlBody()
    reportHtmlStr.append createHtmlReport(formCode, reportHeaders, reportData, startDate, endDate)
    reportHtmlStr.append endHtmlBody()
    output = reportHtmlStr.toString()
}

return ["data":output, "recordsReturned": reportData.size]


/******************
 * HELPER FUNCTIONS
 ******************/
def createReportHeaders()
{
    def reportHeaders = ["ApplicationId"    // 1
                         ,"Submit Date"     // 2
                         ,"Submit Time"     // 3
                         ,"Current Status"  // 4
                         ,"Last Updated"    // 5
                         ]

    return reportHeaders
}


def retrieveData(formCode, startDate, endDate)
{
    def getSubmissionDeliveryListService = ServiceLocator.getServiceForName("GetSubmissionDeliveryList")
    def results = getSubmissionDeliveryListService.invokeService(["formCode":formCode, "startDate":
startDate, "endDate":endDate])
    return results
}


def filterData(results)
{
    StringBuilder dataStr = new StringBuilder();
    def dataTable = [][]

    def filteredResults = results.sort { it.timeSubmission }.reverse()

    filteredResults.eachWithIndex { it, i ->
        dataTable[i] = [
            it.trackingNumber,                      // 1
            it.timeSubmission.format(DATEFORMAT),   // 2
            it.timeSubmission.format(TIMEFORMAT),   // 3
            it.job?.currentStep?.name, // 4
            it.job?.currentStep?.timeCreated.format(DATETIMEFORMAT) // 5
        ]
    }

    return dataTable
}


def createHtmlReport(formCode, reportHeaders, dataTable, startDate, endDate)
{
    StringBuilder dataStr = new StringBuilder();

    dataStr.append "<p>Applications submitted for ${formCode} from ${startDate} to ${endDate}</p>"
    dataStr.append "<table>"

    dataStr.append "<tr>"
    reportHeaders.each {
        dataStr.append "<th>${it}</th>"
    }
```

```
    dataStr.append "</tr>"

    dataTable.each{ it1 ->
        dataStr.append "<tr>"
        it1.each {
            dataStr.append "<td>${it ?: ""}</td>"
        }
        dataStr.append "</tr>"
    }

    dataStr.append "</table>"

    return dataStr.toString()
}


def startHtmlBody()
{
    StringBuilder dataStr = new StringBuilder();
    dataStr.append "<!DOCTYPE html><html>"
    dataStr.append '''<head>
                    <style type="text/css">
                    body {
                        font-family: Verdana, Arial, Helvetica, sans-serif;
                        font-size: 10px;
                    }
                    table {
                        border-collapse: collapse;
                    }

                    table, th, td {
                        font-size: 10px;
                        padding: 3;
                        border: 1px solid black;
                    }
                    td {
                        color: #898989;
                        font-size: 10px;
                        padding: 3;
                        text-align: right;
                    }

                    </style>
                </head>'''
    dataStr.append "<body>"
    return dataStr.toString()
}


def endHtmlBody()
{
    return "</table> </body> </html>"
}


def createCsvReport(reportHeaders, dataTable)
{
    StringBuilder dataStr = new StringBuilder();

    reportHeaders.each {
        dataStr.append "\"${it}\","
    }

    dataStr.append "\n"

    dataTable.each{ it1 ->
        it1.each {
            dataStr.append "\"${it ?: ""}\","
        }
        dataStr.append "\n"
    }
```

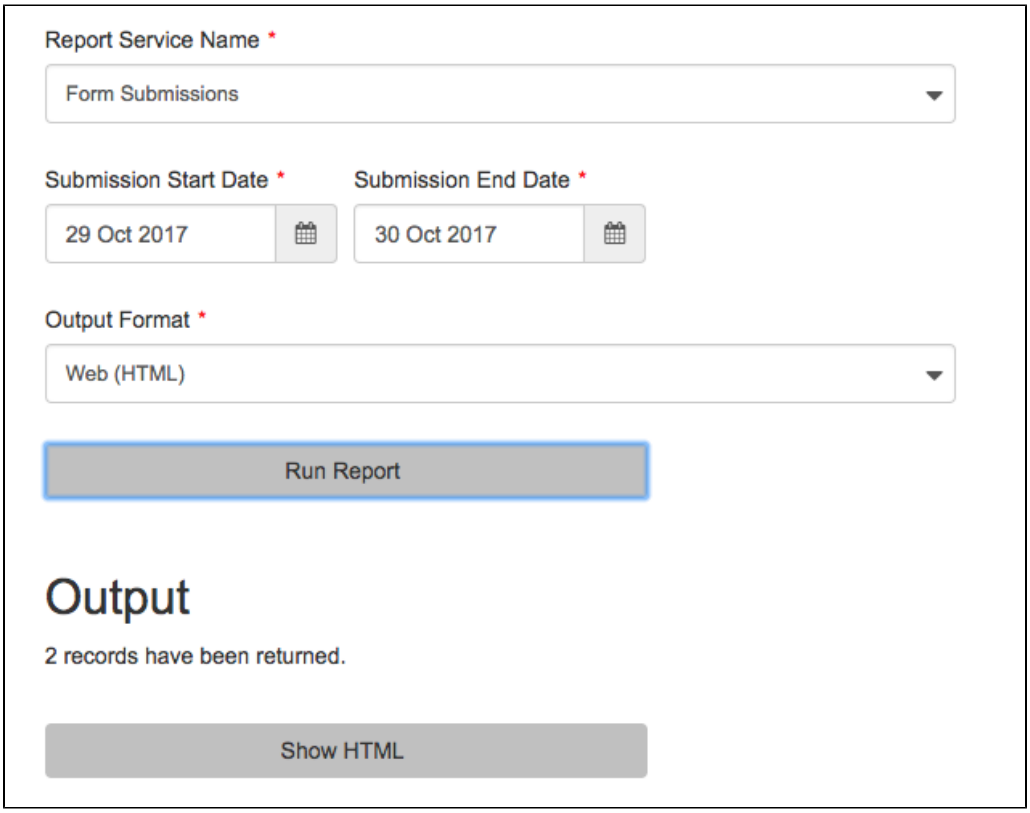
```
    return dataStr.toString()
}
```

3. GetSubmissionDeliveryList - This "DAO Layer" extracts submission objects for a given date range for a specific formcode. I've played with many DAOs and maybe one suits your purpose more than this but what I've found is that many of the DAOs cut off after a certain limit regardless of what you specify. The requirements I've had in many reports goes beyond these limitations so I've found this script/DAO to be the best and surprisingly performs very well. You can also pass a parameter to include Saved transactions.

GetSubmissionDeliveryList

```
import com.avoka.core.groovy.GroovyLogger as logger
import com.avoka.fc.core.service.ServiceLocator
import com.avoka.fc.core.dao.*
import com.avoka.fc.core.entity.*
import groovy.json.*

String formCode = parameters.formCode
Date startDate = Date.parse('yyyy-MM-dd', parameters.startDate)
startDate.set(hourOfDay : 0)
startDate.set(minute : 0)
startDate.set(second : 0)
Date endDate = Date.parse('yyyy-MM-dd', parameters.endDate)
endDate.set(hourOfDay : 23)
endDate.set(minute : 59)
endDate.set(second : 59)

String includeSaved = parameters.includeSaved ?: 'false'

//logger.info "-------------------------------------------------"
//logger.info serviceDefinition.getServiceName()
//parameters.each { logger.info "$it.key: $it.value" }
//logger.info "-------------------------------------------------"

List deliveryStates = ["Not Ready", "Ready", "Sent Email", "In Progress", "Pending", "Completed",
"Error", "Undeliverable", "Not Required"]

String clientId = DaoFactory.getClientDao().getClientByCode("digital").getId();
String deliveryDetailsId = null
String deliveryMessageLike = null
int fetchLimit = 10000

SubmissionDao submissionDao = DaoFactory.getSubmissionDao();
List allResults = []

deliveryStates.each {
    allResults.addAll(submissionDao.getSubmissionDeliveryList(it,
                                                    clientId,
                                                    deliveryDetailsId,
                                                    formCode,
                                                    deliveryMessageLike,
                                                    startDate,
                                                    endDate,
                                                    fetchLimit))
}

if (includeSaved == 'true') {
    def savedList = submissionDao.getSavedSubmissionList(null, clientId, null, null, startDate, endDate)
    def filteredResults = savedList.findAll{ it.formCode == formCode }
    allResults.addAll(filteredResults)
}

//logger.info "Total records retrieved - getSubmissionDeliveryList.size:${allResults.size}"

return allResults
```

Final Look



Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Interacting with TransactField using Custom URLs
- How to rescue attachments from your iOS device
- How to pre-fill static map images into TransactField Tasks
- Determining the IP address that a form is requested from

Compatibility

Since	
Deprecated	

Distinguish Between "Save on Page Change" and "Timed Save"

Unknown macro: 'redirect'

Background

The Fluent Function model uses triggers to initiate functions/services during form processing. These functions/services can be bound to these triggers for each Form Version. One of these triggers, "Form Update", is ambiguous because it can be initiated by either 1) a page change within the form, or 2) a timed background save. In most cases, this ambiguity is not a problem. However, because the former is triggered by the user, with validation taking place on the page data, and the latter is passive, with no validation, some applications need to distinguish between these events. For example, pushing the XML form data to a 3rd-party application (e.g. Salesforce) might work fine at a page change, but might contain invalid data for a timed save.

Solution

This section documents one possible solution.

Make a Place to Store the New Metadata

1. In the Form XML Data, add an element /AvokaSmartForm/Meta/AutoSaveType.
2. If you are using Value Objects to map the Form Data to a Java object, add a String variable autoSaveType to the Meta value object class.

Update the Metadata Before Calling the Service

1. In Maestro, in Form Options / Policies / Background Save On Page Change, disable the option "Background Save on Page Change".
2. In Maestro, replace the functionality disabled in step 1 with the following: on each page, add a "Page change" rule that executes code similar to the following:

```
/* Let the save function know that this background save is due to a
   page change, to distinguish it from one triggered by a timed background save. */
data.autoSaveType = "PageChange";

var promise = Form.backgroundSave();

/* Settle the promise before resetting the state. */
promise.then(
    function(result) {
        console.debug('Form Background Save Result: ' + result);
    }).catch(function (error) {
        console.log('Form Background Save Error: ' + error);
    }).finally(function () {
        /* Reset the state. */
        data.autoSaveType = "";
    });
```

Update the Service to Use the New Metadata

The Groovy code below is contrived and for illustration purposes only. It assumes that the XML Form Data element /AvokaSmartForm/Meta has been unmarshalled into the value object **appMeta**.

```
boolean isFormUpdateForPageChange(FuncParam param, Meta appMeta) {
    logger.debug("Trigger: " + param.trigger)
    if (FuncParam.TRIGGER_FORM_UPDATE.equals(param.trigger)) {
        logger.debug("meta.AutoSaveType: " + appMeta.autoSaveType)
        return "PageChange".equals(appMeta.autoSaveType)
    }
    return false
}
```

Handling Transact Form Function Exceptions in Maestro Applications

Unknown macro: 'redirect'

Transact Form Functions are easy to use for the happy path scenarios when no errors or exceptions are taken into consideration. The simplest implementation can look like this:

```
Transact.formFunction(ffName, '0.1.0', 'myMilestone',ffData).then(function (response))
```

However, in real life errors do happen, so Transact Form Function call implementations must cater for all sorts of errors and exceptions that may occur and handle both the success and error return messages. There are several ways of doing this; the following pattern is recommended: the Form Function captures all responses and returns different flags, for success or error statuses, and a detailed error message. This article explains how to use this pattern and provides examples of both server and client source code.

Transact Manager - the Server Side

If there is going to be an error, most developers would expect it to happen in a Transact Form Function that runs in Transact Manager. This is where calls to other systems are made, and most data manipulation occurs, so any exceptions should be handled here. However, placing a try-catch block around the code isn't enough. We need a consistent way to return a function call result, a success or a failure, back to the Maestro application's JavaScript, so it can effectively handle it and possibly display to a form end user.

When you develop Transact Functions, try to make them as self-contained as possible with regards to how they send data back to the application. Maestro applications have limited data passed back by a Transact Function when an exception occurs in Transact Manager. Usually, it means that something catastrophic happened on the server side.

To address this limitation, you should create consistently named variable pairs and return them in the **FormFuncResult** object; for example, **executionStatus** and **errorMessage**.

You can use the **executionStatus** variable to indicate whether the Transact Function call has followed a normal execution path or encountered a problem. A good development practice is to create a set of constants to use across your Transact SDK projects. The example below uses constants as String objects. Condition codes are returned to indicate the following outcomes: success, problems with the input data, problems accessing third party APIs, or trapped exceptions.

You can use the **errorMessage** variable to hold an additional detailed message to compliment the **executionStatus** value. If the Transact Function doesn't capture an extra error message, this variable will be empty.

When an exception occurs, the Transact Function will catch the exception and populate the response object with one or both of **exceptionStatus** and **errorMessage**, so the Maestro application can better manage the user experience.

Maestro Application - the Client Browser Side

The **Transact.formFunction()** call from an application to a Transact Manager service returns a promise. You can use this callback for successful, also known as happy path, results. However, this callback doesn't allow you to handle all possible conditions gracefully. You will need the second callback **catch()** added to the **then()**. This callback is executed if an error has occurred; for example, a Transact Manager server failure or a network issue between the application and the Transact Manager server. The catch (function(err)) input parameter is an error object containing a message which you can process and display to the end user.

The **catch()** block handles any rejections of the initial promise, specifically during the first **then()** callback. One use case is when a system-level error is detected in the Form Function in the Transact Manager server, the application can check this in the first callback and throw a new Error there. That Error object can then be processed in the **catch()** block.

Remember that **then()** and **catch()** promises are generic JavaScript structures and syntax, not Maestro specific.

Code Examples

The Click Rule JavaScript and Form Function Groovy Script below illustrate the exception handling pattern. They allow a user to select a test scenario, which can be one of the following options: Success, **DataError**, **SystemError**, Exception, and TM Error. The console log output shows the path of each scenario through the code.

The Click Rule JavaScript

```
var ffName = "Form Function Full Example";
var ffData = {"testType":data.selectTest};

if(data.selectTest === "TM Error") {
```

```
  // Changing the name used will cause TM to report an error.
  ffName = "Will Not Work";
}

console.log("");
console.log("Start...");

Transact.formFunction(ffName, '0.1.0', 'myMilestone', ffData)
.then(function (response) {

  console.log("The Form Function was called, and returned something.");

  if(!Util.isBlank(response)) {
    if (!Util.isBlank(response.data) && !Util.isBlank(response.data.executionStatus)) {
      switch(response.data.executionStatus) {

        case "DATA_ERROR":
          console.log("The return value status code indicates DATA_ERROR, with message: " + response.data.
errorMessage);
          console.log("An exception is not being thrown based on design.");
          break;

        case "SYSTEM_ERROR":
          console.log("The return value status code indicates SYSTEM_ERROR, with message: " + response.data.
errorMessage);
          console.log("An exception can be thrown here if design dictates.");
          throw "SYSTEM_ERROR - " + response.data.errorMessage;
          break;

        case "SUCCESS":
          console.log("The return value status code indicates SUCCESS, with data: " + JSON.stringify(response.
data));
          break;

      }
    } else {
      console.log("An empty data object indicates an service problem.");
      console.log("An Exception/Error will be thrown manually from JavaScript.");
      throw "Empty Data Object";
    }
  } else {
    console.log("An empty response object indicates an infrastructure problem.");
    console.log("An Exception/Error will be thrown manually from JavaScript.");
    throw "Empty Response Object";
  }
}, function(err) {
  console.log("TM throwing an exception gets here - Using function(err)");
  console.log("Error Message: " + err.message);
}).catch(function(err){
  console.log("JavaScript throwing an exception gets here - Using promise's catch(function(err))");
  console.log("Error Message: " + err);
});

console.log("End of call.");
console.log("Calling Form Functions is an asynchronous operation.");
console.log("Execution continues here before the Form Function returns.");
console.log("");
```

The Form Function Groovy Script

```
package com.avoka.funcs.fffullexample;

import com.avoka.tm.func.*
import com.avoka.tm.query.*
import com.avoka.tm.svc.*
import com.avoka.tm.util.*
import com.avoka.tm.vo.*
```

```
import javax.servlet.http.*

public class FormFunctionFullExample {

    // Injected at runtime
    public Logger logger

    /*
     * Perform Fluent Function call.
     *
     * returns: FuncResult
     */
    FuncResult invoke(FuncParam param) {

        def testType = param.params.get('testType').toString()

        logger.info("Test Type: ${testType}")

        FormFuncResult result = new FormFuncResult()

        def data = [
                rate   : 2.0
        ]

        // Choose execution path
        try {
            switch (testType) {
                case "DataError":
                    // Return data error
                    result.data.put("executionStatus", "DATA_ERROR")
                    String msg = "Invalid Input"
                    result.data.put("errorMessage", msg)
                    result.data.put("exchangeRate", {})
                    break

                case "SystemError":
                    // Return system error
                    result.data.put("executionStatus", "SYSTEM_ERROR")
                    String msg = "External System Down"
                    result.data.put("errorMessage", msg)
                    result.data.put("exchangeRate", {})
                    break

                case "Exception":
                    // Throw an unexpected exception
                    throw new Exception("FormFunctionFullExampleException")
                    break

                case "Success":
                default:
                    // Return sucessfully
                    result.data.put("executionStatus", "SUCCESS")
                    result.data.put("exchangeRate", data)
                    break

            }
        }
        catch (exception) {
            result.data.put("executionStatus", "SYSTEM_ERROR")
            String msg = "Exception Caught in Service - ${exception.toString()}"
            result.data.put("errorMessage", msg)
            logger.debug msg
            result.data.put("exchangeRate", {})
        }

        return result
    }
}
```

Success

The successful path, or the happy path scenario, is the result when everything in your application and Form Function works as expected. The Form Function validates the input values, any external services return to the caller function, so everything is good. In the example above, this results in the "Success" case being triggered. The response object contains data to be displayed, and the **response.data.executionStatus** value is "SUCCESS".

Note the script log output is asynchronous, which is the nature of **Transact.formFunction()** call.

DataError

Sometimes development of an application and a Fluent Function get out of sync, or more likely an external service's logic implementation changes. This means that the inputs provided by the Maestro application pass its own validation, but they fail the validation of the Form Function or the third party service. In this case, the Form Function returns "DATA_ERROR" as "**executionStatus**", and some useful error message as **errorMessage**. You should always try to catch and fix these **DataError** situations during your software development and test cycle.

Execution flow through both the Form Function and application JavaScript is often unrestricted, meaning no exceptions are thrown. The problem can be gracefully reported to the end user, so they can report it to a system administrator or developer.

SystemError

SystemError scenarios cover any error conditions more severe than data validations. In the example above, the Form Function calls an external API and the external system returns an error or throws an exception. The Form Function catches this error and returns the appropriate values in **result.data. executionStatus** and **result.data.errorMessage** to the application.

The notable difference lies in implementing the application's JavaScript throw clause, which is used when a "SYSTEM_ERROR" value is returned in executionStatus. This moves the execution path into the catch() promise, where the error can be handled differently to a DataError in this pattern.

Exception

This execution path shows how an exception being thrown in the middle of the Form Function code should be captured by a try/catch and the result object populated accordingly. This scenario is otherwise handled as a **SystemError**.

TM Error

Finally, there is the scenario where Transact Manager has failed catastrophically. The example simulates this by using a Form Function that does not exist. Some infrastructure issues can cause similar errors; for example, network is not available. There is nothing in the Form Function code to handle this kind of error, as it would never be called.

The application JavaScript has a code path that is executed only when this level of error happens, the second promise used for rejections. The **console. log()** outputs the only information available about the cause of the error **(err.message)**.

Conclusion

This error handling pattern demonstrates two things:

1. How to code for error conditions using exceptions and the additional promise functionality available;
2. A way to convey execution status between Transact Manager and the Maestro application run on the end user's device. Using **executionStatus** and **errorMessage** is a simple, and yet powerful approach. It allows the Form Function to pass error related information back to the client while keeping all error handling on the service.

Complete Examples

You can download the Maestro form and Transact Manager service examples below and try them in your TM environment.

- Transact Maestro form example: maestro-form-functionFullExample-v1.0.2-2018-07-06.zip
- Transact Manager service example: form-function-full-example-service-v0.1.0.zip

How to find the Data Model for Form Space (Velocity) Pages

Unknown macro: 'redirect'

Partial Mapping of Templates to Model

Content May Change

The templates and classes may change from time to time. This page is intended to be a helpful starting place, not a "source of truth" document. Check the pages and classes themselves for the actual behavior.

The following table lists the model variable names and types for each page.

- Java type names are not fully-qualified unless clarity is needed. For example, class String is the normal java.lang.String unless otherwise noted.
- Java types marked as an Entity are in the package com.avoka.fc.core.entity unless otherwise noted.
- Model classes that "inherit from" other model classes inherit the variables of those classes. For example, the model for ErrorPage not only includes the variables listed for it, but also all of the variables for BorderPage.
- The fact that the model can include a given variable does not mean that the variable is non-null at runtime.

Page	Model Class	Variable Name	Java Type	Inherits From
(global)	PortalUtils-Form This is not a Model class, but a utility method used by Model classes to set certain properties in the Model from the Form PortalUtils.addFormAndVersionProperties(Page, Form)	form	(Entity) Form	
		formProperties	Map<String, String>	
	BorderPage	headerPath	String	
		serverBuild	String	
		portal	(Entity) Portal	
		detectJavaScriptTimeout	int	
		disableUserProfileEdit	boolean	
		allowUserCreation	boolean	
		permissionSet	Set<String>	
error.htm	ErrorPage	entityId	String	BorderPage
		errorLog	(Entity) ErrorLog	
		errorContext	String	
		showDetails	Boolean	
		messageKey	String	
		errorMessage	String	
		errorTitle	String	
submission-cancelled.htm	SubmissionCancelledPage	formRenderUrl	String	BorderPage PortalUtils-Form
		formCode	String	
submission-saved.htm	SubmissionSavedPage	submission	(Entity) Submission	BorderPage PortalUtils-Form
		isAnonymous	boolean	
		formUrl	String	
		userAccount	(Entity) UserAccount	
		userName	String	
		logoutMsg	String	

		todoCount	int	
		evaluatedPage	String	

How to Log Out of Transact Manager from the Client

Unknown macro: 'redirect'

Background

Explicitly logging out of Transact Manager is important in a few use cases. One example is using single sign-on with a 3rd-party application: if the 3rd-party application user can logout, then login as a different user, the 3rd-party application user is now different than the TM user.

Here is the actual example that drove the creation of this page: User A logs into Salesforce, gets an SSO account in TM and passively logs in to TM. User A then exits the TM form and logs out of Salesforce. User B then logs in to Salesforce in the same browser. Because the User A TM login is still running, User B's Avoka form is prefilled with User A's data.

Solution

The solution is to explicitly log out of Transaction Manager when the user exits the Avoka form. Here is a non-exhaustive list of possible places where this should be done:

- Cancel/Exit
- Form Submission
- The **beforeunload** or **unload** browser events.

Caveat: https://support.avoka.com/jira/browse/AVT-6107 has been raised, proposing that a method with similar functionality be added to the Maestro API. Please check the status of that issue against the Maestro version you are using before using the solution proposed on this page.

To logout of Transact Manager, make a GET request to the path /logout from the appropriate TM module (portal), which does the following:

- Logs the user out of Transact Manager
- Clears the TM-related cookies

Here is some sample javascript code:

Sample Code

```
/* Logout of Transaction Manager. */
function logoutTMInstance() {
    console.debug("Start Avoka logout");
    var href = window.location.href;
    var arr = href.split("/");
    var url = arr[0] + '\/\/' + arr[2] + '\/' + arr[3] + '\/logout.htm';
    var xmlHttp = new XMLHttpRequest();
    /* Call service synchronously. Change 3rd parameter to "true" for an asynchronous call. */
    xmlHttp.open( "GET", url, false);
    xmlHttp.send(null);
    console.debug("Avoka logout complete");
};
```

PDF Receipts

Unknown macro: 'redirect'

- Dynamic PDF Receipt Service
- Forcing a Receipt to be Re-generated
- Generating Multiple PDF Receipts from a Single Submission
- How to import an Acrobat PDF form as a Transact receipt template (AcroForms)
- Inserting a Watermark in a PDF Receipt in V4.3.1
- PDF Receipting Options

Dynamic PDF Receipt Service

Unknown macro: 'redirect'

The purpose of this is wiki is to explain how the Transaction Manager Service interacts with PhantomJS so you can form the correct mental model when troubleshooting issues.

This wiki shows:

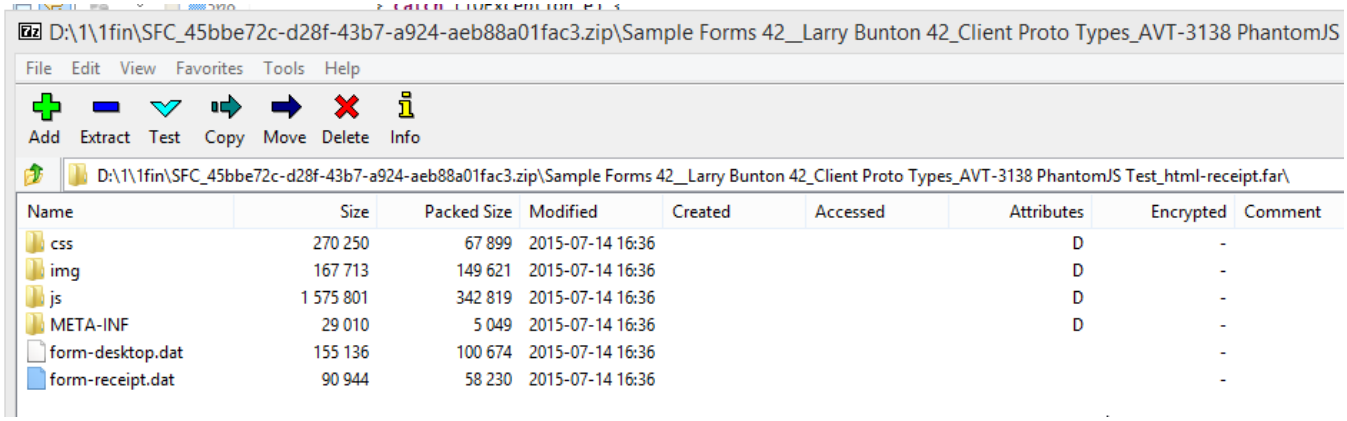
- What a form requires to render using the Dynamic Render service
- The location of the PhantomJS program and config files.
- Details the temporary directory that TM writes the form to be rendered.
- Shows the Dynamic PDF Receipt Service (TM Receipt Render) and service parameters
- The processing steps are summarised and listed
- Explains Legacy and Callback rendering and product version requirement.
- Gives an example batch file that can be used for testing from the command prompt.

How Does PhantomJS work?

Pre Requisites

The form must be deployed with the Dynamic PDF Receipt Support which creates a form-receipt.dat html receipt template.

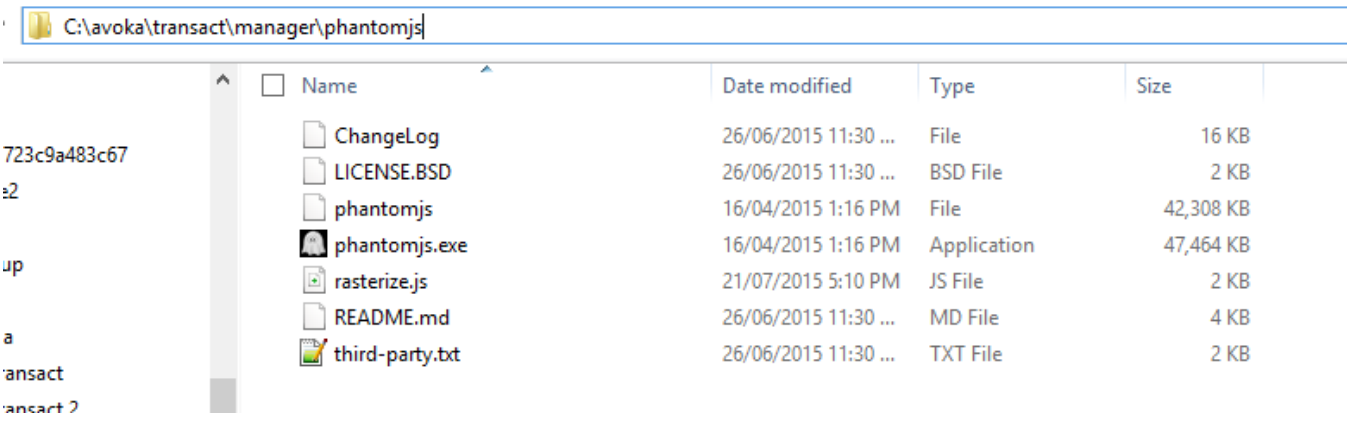
The screenshot below shows a far file opened in 7Zip showing where the form-receipt.dat exists in a far file.



Note

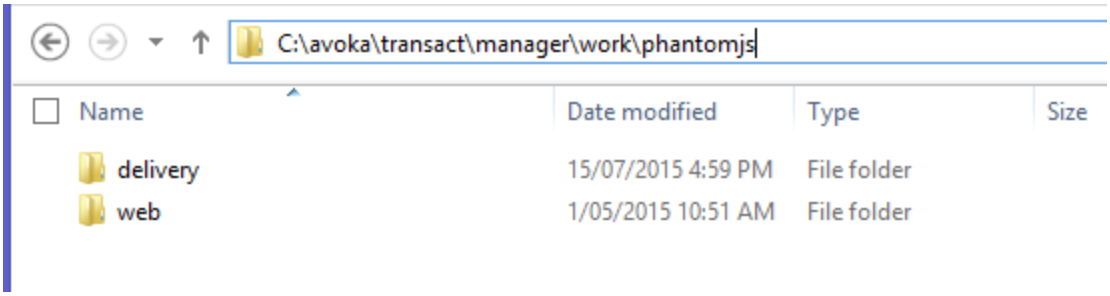
Old forms that used LiveCycle Output to create pdf's will have a form-receipt.xdp. These must be republished with Dynamic PDF Receipt Support, generated to include the form-receipt.dat html receipt template.

PhantomJS Program Files Location



The main configuration file for phantomJS is **rasterize.js**. The content of rasterize.js changes depending upon the version of Transaction Manager

Temporary Directory Location



Transaction Manager will create a temporary directory based upon the form submission id. The receipt maybe generated by different threads such as a scheduled job or by the end user clicking on the download receipt button. To avoid thread conflict between the user initiated rendering and the scheduled job.

The delivery directory is used when a system is calling a schedule job. For example:

\avoka\transact\manager\work\phantomjs\delivery\1234

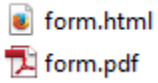
The web directory is used when a user clicks to download the receipt. For example:

\avoka\transact\manager\work\phantomjs\web\1234

Temporary File

The form.html is the **input** created by TM.

The form.pdf is the rasterized pdf **output** that is created by phantomJS.



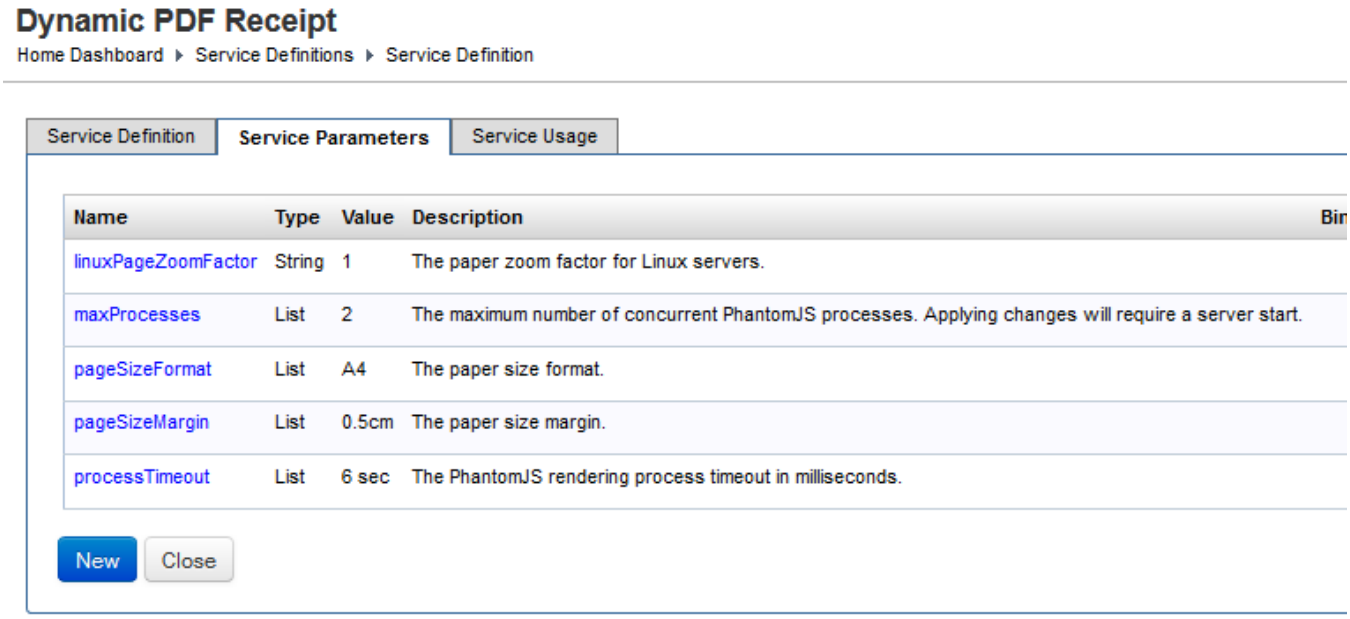
Note

- The directory is cleaned up (Deleted) after the rendering is complete
- The 4.3.0 release of TM will contain a test tab which has the option of not deleting the temporary directory form.htm. This only applies to forms that are rendered from the test tab.

Dynamic PDF Receipt Service (Receipt Render)

The Java class behind this is **PhantomJSRenderReceiptService.java**

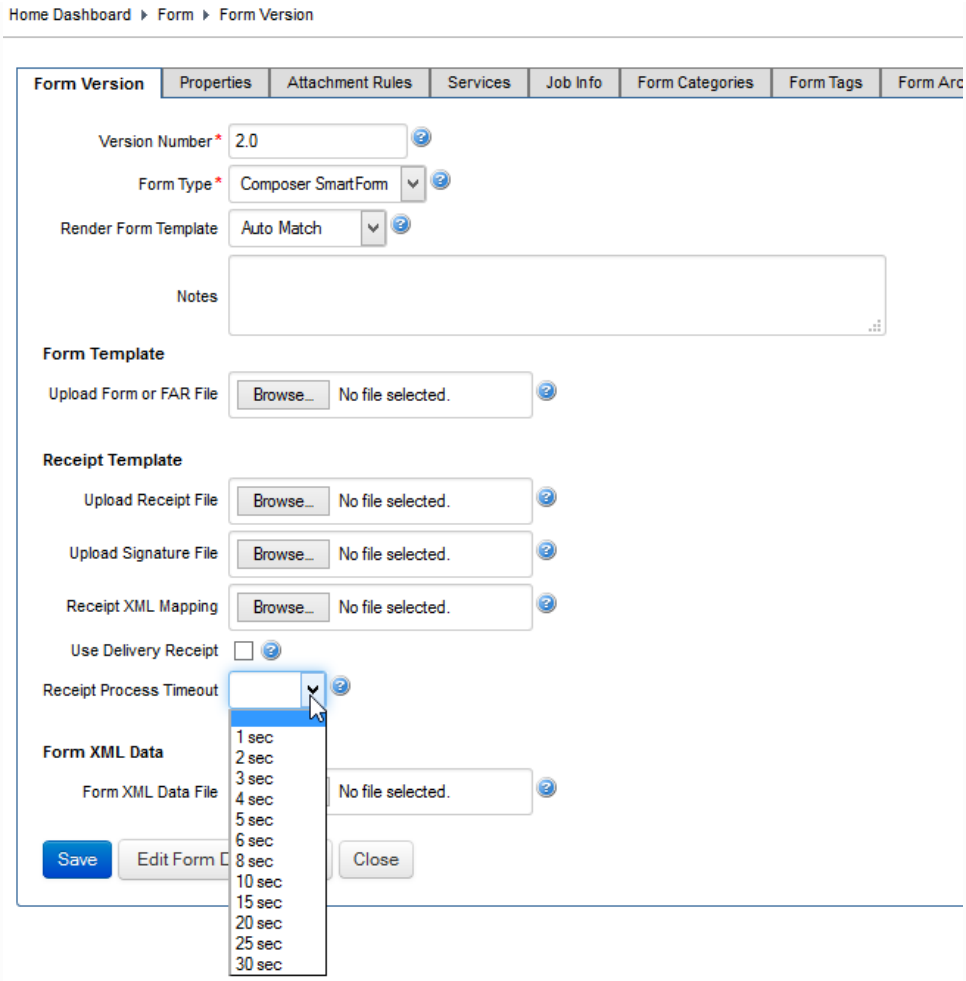
Below are the default service parameters and the description for Transaction Manager.



maxProcesses: Transaction Manager will restrict the number of instances of PhantomJS that run at the same time. This is done using a Semaphore in the Java code.

Note - Transaction Manager Version Specific

- TM 4.0.X does not have the processTimeout as a service parameter. The renderProcessTimeout is held in the rasterize.js on each server node.
  The processTimeout can be overridden in the Form Version Template.



- The **linuxPageZoomFactor** parameter was introduced in 4.2.0. For Windows it should be 1, for Linux 0.79

Receipt Rendering Steps

1. TM service generates HTML form page, injecting XML and in-lining all CSS, JS and Images.
2. TM creates a temporary directory and writes the HTML file down to disk.
3. TM calls PhantomJS with the following commands
   phantomjs rasterize.js URL filename [paperSize.format] [paperSize.margin] [processTimeout] [pageZoomFactor]
   for example
   phantomjs rasterize.js file:///D:/1/phantomjs/form.html D:\1\phantomjs\form.pdf A4 1cm 60000 1

> **Note: TM Version Specific**
>
> The parameters used in step 3 are for TM 4.2+

4. PhantomJS opens HTML file and renders form
   The time this takes depends upon whether the form implements Legacy Mode Rendering or Rendering with Callback mechanism see below.
5. PhantomJS rasterized HTML file creating a PDF document
6. TM reads PDF document (form.pdf) off disk

Legacy Mode Rendering

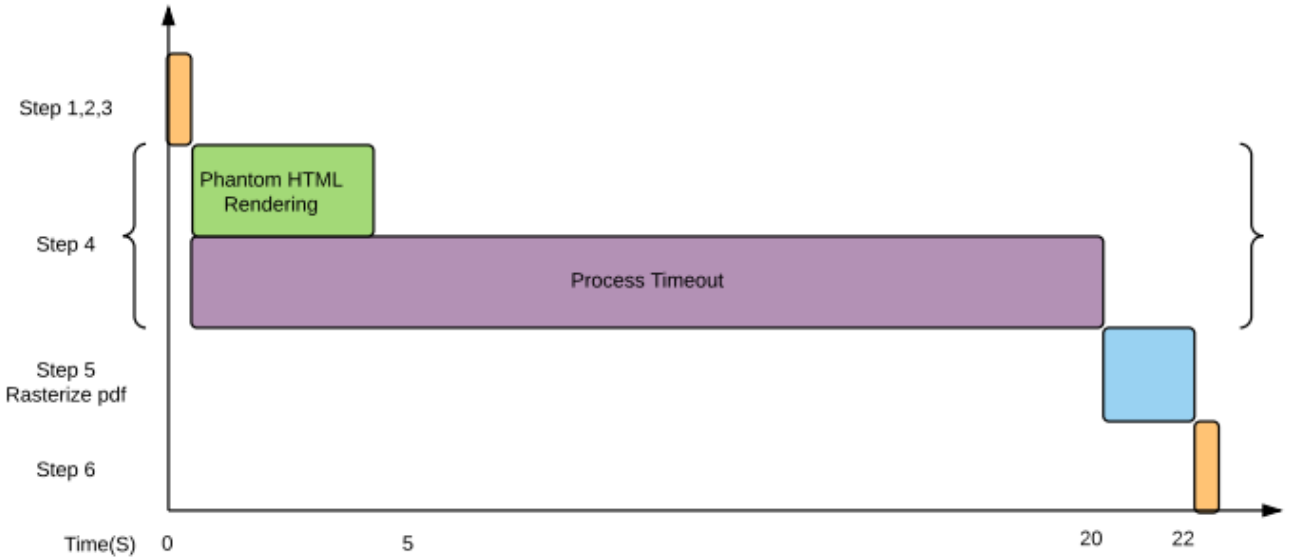
Used if:

- Composer Form Version: 4.1 SP2 or earlier
- Transaction Manager: 4.1.5 or earlier

What the process timeout does is it forces phantom to wait for that duration before it tries to rasterize the form into the pdf.

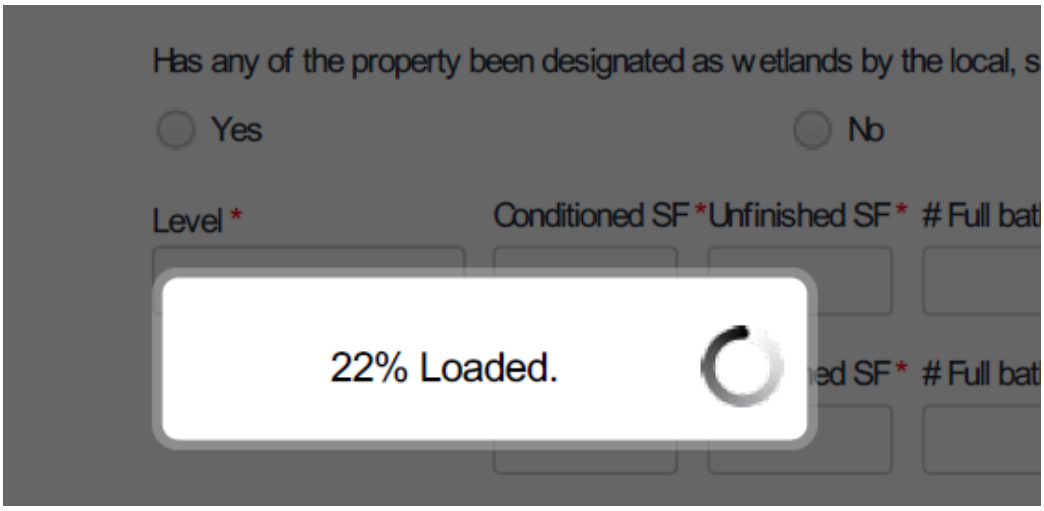
For example if we set a process timeout of 20s for a form.

- Step 1,2,3 are performed in milliseconds
- Step 4 PhantomJS may render the html in 4s.
- It then waits for a total of 20s for the process timeout to pass. This is another 16s before executing the next step- rasterize to create the PDF.
- Step 5 Rasterizing can add an addition 1-5 seconds, lets say this takes 2s.
- Step 6 is done in milliseconds
- This gives a total time of a bit over 22s to render the form



The problem with this is that the rendering is waiting around idle for 16 additional seconds to every receipt processed.

If we set a timeout of 0s then phantom may not have rendered the html before it rasterized the html into a pdf. You end up with a receipt pdf that may not have all the fields, it may be greyed out and it may also contain a percentage complete dialog as per the screenshot below..



Best practise (TM 4.1)is to set the processTimeout just a bit more than the time it takes phantom to render.

Rendering with Callback Mechanism

This is used with:

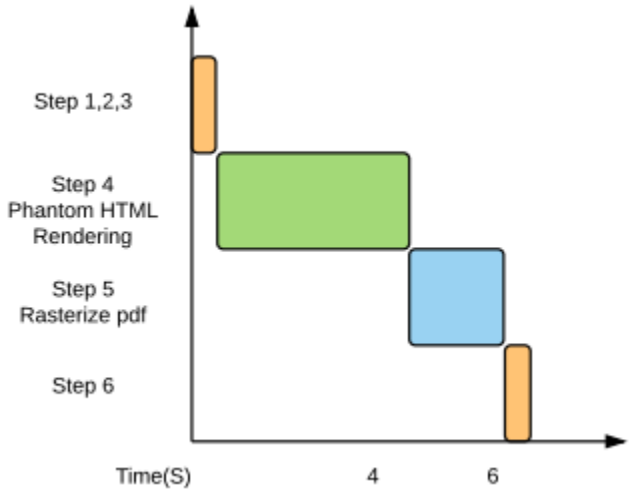
- Composer HTML Form receipts with versions greater than 4.2+, The current Composer 4.1 sp2 does not contain the callback, Note the callback may be backported to the next 4.1 service pack starting from sp3.
- Maestro forms
- Transaction Manager 4.1.6 + is required

To test if a form contains the callback, search for "**callPhantom**" either in the far files -> form_receipt.dat or in the generated form.html.

When the HTML rendering (Step 4) has finished the form calls back to to Phantom JS rasterize.js to say I am done, you can proceed with the rendering.

A default process timeout of 20s is set with the forms to deal with cases where the form fails to trigger the callback. Jack Coulter said that if a form is taking over 20s to render its usually because of a bug in the form.

- A process timeout of 20s set for any form that uses callback
- Step 1,2,3 are performed in milliseconds
- Step 4 renders the html in 4s.
- The form calls back into the rasterize.js onCallback() method to say I'm finished rendering goto Step 5.
- Step 5 Raterizing this takes 2s.
- This gives a total time of a bit over 6s to produce the form.



TM Timeout

Why is TM allowing my phantom receipt to take longer that the timeout specified? I have specified a timeout of 20s but my receipts take 22s to run.

Transaction Manager creates the phantom receipt in a new thread. It kills the thread if that thread is taking **2 x timeoutProcess.**

With the above example it gives the instance of phantom **40s** to complete Steps 1-6**.**

Testing

The following batch file calls phantomJS to render a html file. This can be the form.html produced by TM.

It calls phantomjs to produce the form.pdf file.

It backs up the last 5 files so you can compare the files.

test.bat (Windows)

```
 echo %time%
@REM phantomjs rasterize.js file:///D:/1/phantomjs/form.html D:\1\phantomjs\form.pdf A4 1cm 60000 1
@REM phantomjs rasterize.js file:///D:/1/phantomjs/form.html D:\1\phantomjs\form.pdf A4 1cm 30000 1
@REM phantomjs rasterize.js file:///D:/1/phantomjs/form.html D:\1\phantomjs\form.pdf A4 1cm 0 1
phantomjs rasterize.js file:///D:/1/phantomjs/form.html D:\1\phantomjs\form.pdf A4 1cm 20000 1
echo %time%
d:
cd \1\phantomjs
dir form.pdf
@copy form_4.pdf form_5.pdf
@copy form_3.pdf form_4.pdf
@copy form_2.pdf form_3.pdf
@copy form_1.pdf form_2.pdf
```

```
@copy form.pdf form_1.pdf
del form.pdf
pause
```

Troubleshooting Tips

View the HTML version of the receipt in the TM receipt test harness

In the TM Receipt Test Harness there is a button 'Test HTML Receipt'. This will render the HTML that will be sent to phantom to rastorize.
Using the network tab in the debug tools look to check that all resources that the form attempts to load are actually available. You will need to refresh the page to be able to observe this.
If the phantom process has problems accessing resources this can cause a number of odd receipt issue and particularly plays havoc with Web Fonts. Timeouts are also a symptom of this.
Also look for any resources that are loaded from a location other than the TM server. If these types of resources are used receipt rendering can have problems as the phantom receipt process running in the server environment will need to reach out for these resources and this level of network activity will also be slow and may well be blocked. Including all resources and fonts in the far file is the best approach.

Debugging

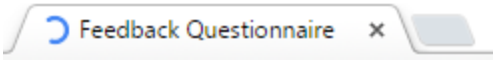
If there are more than 2 (*ServiceParameter.maxProcesses*) **PhantomJS.exe** programs running but not finishing in Task Manager (Windows) or Linux equivalent. TM won't be able to render forms.

Replacing the **rasterize.js** on Staging whilst the TM was running caused an issue where the running PhantomJS.exe failed to stop. The result was that 4 processes were left running. Transaction Manager started to render forms correctly after these were killed off.

> **Note**
>
> Only replace rasterise.js when the Avoka Transaction Manager server has been stopped.

Utilising Chrome as a Debugging Tool

Since PhantomJS is built from the same WebKit that Google Chrome orginated from it is possible to use the Chrome web browser to help identify problems. When a page is loaded in Chrome the active tab will dispay a status icon, a circulating blue ring



. This icon shows when the page is making http requests and waiting for responses, and when it is actually loading resources for the displayed page, images, flash or javascript files for example.

- Anti-Clockwise indicates the browser is making http requests or waiting for a response.
- Clockwise indicates that the browser has all the components and is loading them into view or running any scripts that are part of the web page.

This can be useful for diagnosing errors in reciept generation. As a case, an issue was identified when reciepts were timeing out during the render process. This was caused by javacript running when the reciept is rendered that was failling. When the form.html was loaded into Chrome the page displayed but continued to display the loading icon providing a point of investigation to locate the cause of the issue.

Forcing a Receipt to be Re-generated

Unknown macro: 'redirect'

You may encounter a need to re-generate a receipt after it has already been generated.

The code in the Fluent SDK looks something like this:

```
new TxnUpdater(param.txn)
  .setReceiptStatusReady()
  .update()
```

This works by setting the receipt status to "ready", which means the receipt generation background process will kick in and regenerate the receipt.

To actually get at the generated receipt, you need code something like this:

```
//generate pdf receipt
byte[] receiptBytes = new ReceiptSvc()
  .setTxn(txn)
  .renderReceipt();
```

Note that the receipt generation is a background process, and may take some time to execute. You cannot assume that the receipt will be generated immediately.

Generating Multiple PDF Receipts from a Single Submission

Unknown macro: 'redirect'

Requirements

Sometimes it is necessary to generate multiple different PDF receipts from a single submission. There are several use cases for this, for example:

- You may want a different PDF receipt depending on some data in the form. For example, a slightly different variation of the receipt depending on which state the applicant is from.
- You may want to generate multiple separate documents, and concatenate them together. For example, a single application form may result in both a credit card and a check account PDF to be generated. You may want to concatenate these two documents together, or perhaps just provide the two separate documents to downstream business processes.

Compatibility

Since	
Deprecated	

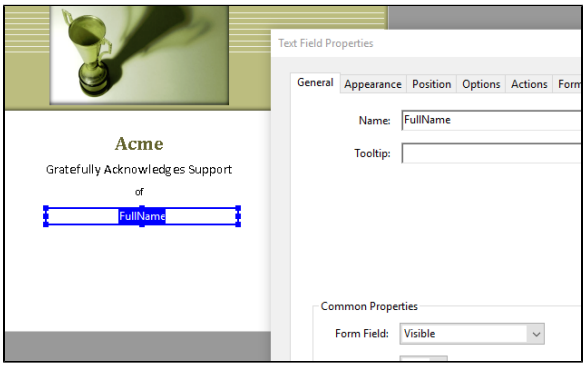
Overview

For this particular example, we are going to generate a certificate for a course attendee. When the user fills in the form, they will be able to choose between different PDF receipts, and green receipt, and a receipt with a picture of flowers on it.

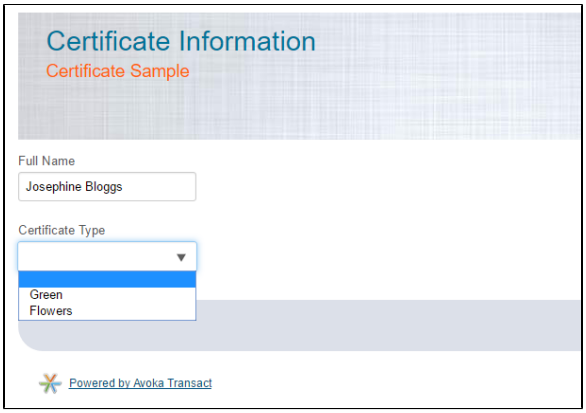
In this case, we are going to generate “pixel perfect” or AcroForm receipts. Please refer to this article for more information about AcroForm Receipts: How to import an Acrobat PDF form as a Transact receipt template (AcroForms).

The code is slightly different if you want to generate other types of receipts, but the principles are similar.

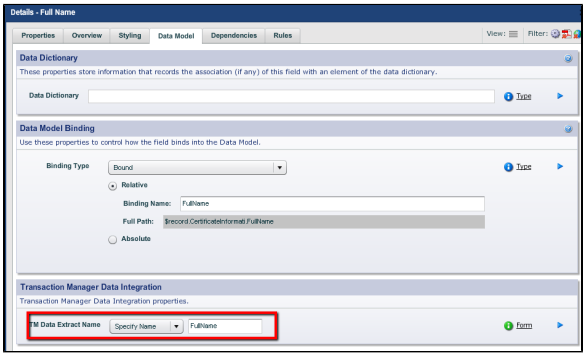
Each of the Acroforms has a single Acroform field placed on it – this field is named “FullName”. For consistency, both forms have the identical field name – but in a real scenario there can be a combination of common fields and form-specific fields. You just need to make sure that the data entry form provides all the necessary fields.



The data entry form has two fields, a field for the person’s name, and a dropdown containing the values “Green” and “Flowers”.



Each of these fields has a data extract name. We will use the data extract values for binding to the Acroform, as well as for the logic in our script.



The techniques for creating and generating an AcroForm receipt is described in the AcroForm article. But for the multiple receipt example, we have to solve two additional problems.

1. We need somewhere to store the two (or more) PDF receipt template files. By default, Transact Manager only stores one receipt template per form.
2. We need to create a custom receipt rendering service. By default, the various PDF rendering receipt services that Transact Manager provides only generate a single receipt.

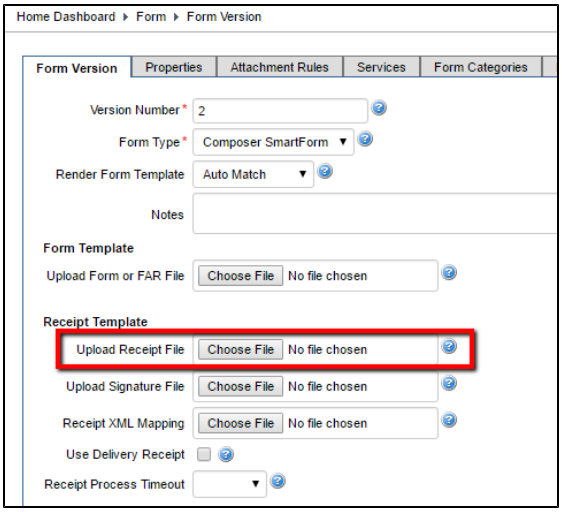
Storing multiple PDF receipts

In order to store multiple receipts, we will create a “dummy” form in Transact Manager for each receipt. This form is not intended for data entry, and it will never be visible externally. It’s simply used as a place to store the receipt.

There are two ways to do this. Either:

1. Deploy your data entry form twice, under two different form codes. Or
2. Deploy your data entry form once. Then go into Transact Manager, and manually create a new form. You will need to upload a data entry form when you create this form, because Transact will not allow a form to be created without a data entry template. However, this can be any form, since the data entry form will never be visible.

Once you have defined your data entry form plus your additional “dummy” forms, you can upload the AcroForm PDF receipts to each of these form codes as shown below:



Custom Rendering Service

The Groovy code is shown below. This is an example of how you might write this script, and of course your needs may vary. The code is hopefully reasonably self-evident for Groovy programmers. You may want to refer to http://itextpdf.com/ for more details about the iText PDF library which does the work of merging the field data into the AcroForm.

```
/* Render Receipt Groovy script.
   Script parameters include:
       form : com.avoka.fc.core.entity.Form
       request : javax.servlet.http.HttpServletRequest
       submission : com.avoka.fc.core.entity.Submission
       serviceDefinition : com.avoka.fc.core.entity.ServiceDefinition
       serviceParameters : Map<String, String>
   Script return:
       a PDF receipt document of type com.avoka.fc.form.service.
DataDocument
*/
import com.avoka.core.groovy.GroovyLogger as logger
import com.avoka.fc.core.dao.DaoFactory
import com.avoka.fc.form.service.DataDocument
import com.itextpdf.text.pdf.AcroFields
import com.itextpdf.text.pdf.AcroFields.Item
import com.itextpdf.text.pdf.PdfReader
import com.itextpdf.text.pdf.PdfStamper
import org.apache.commons.lang.Validate
logger.info "Start render"

final DATA_EXTRACT_CERTIFICATE_TYPE = "Certificate Type"
final FLOWER_FORM = "certificate-flowers"
final GREEN_FORM = "certificate-green"

// Get the Data Extract
Map<String, String> dataExtractMap = submission.dataExtractMap
logger.info "dataExtractMap=" + dataExtractMap

// Using the Certificate Type field data to determine which form code to
use
def certificateType = dataExtractMap.get(DATA_EXTRACT_CERTIFICATE_TYPE)
def formCode = ""
if (certificateType == "Green") {
  formCode = GREEN_FORM
} else {
  formCode = FLOWER_FORM
}
logger.info "formCode=" + formCode

// Get the appropriate form object
def form = DaoFactory.getFormDao().getFormByFormCode(formCode.
toLowerCase())
```

```
// Call the function to render the AcroForm and return the result
def receiptDocument = renderAcroForm(form, dataExtractMap)
logger.info "End render"
return receiptDocument

// This function uses iText to inject the fields from the data extract map
into the AcroForm receipt
def renderAcroForm(form, dataExtractMap) {
    final CONTENT_TYPE_PDF = "application/pdf"
    Validate.notNull(form, "form parameter is null")
    Validate.notEmpty(dataExtractMap, "dataExtractMap parameter is null")

    def templateVersion = form.getCurrentVersion()
    Validate.notNull(templateVersion, "form.currentVersion parameter is
null")

    byte [] templateData = templateVersion.getTemplateVersionData().
getReceiptFileData()
    Validate.notNull(templateData, "form.currentVersion.
templateVersionDate.receiptFileData parameter is null")

    // Create the output stream
    ByteArrayOutputStream stream = new ByteArrayOutputStream(256 * 1024)

    // Setup the PDF receipt template
    PdfReader templatePdfReader = new PdfReader(templateData)
    PdfStamper pdfStamper = new PdfStamper(templatePdfReader, stream)

    // Bind the AcroForm field values
    AcroFields acroFields = pdfStamper.getAcroFields()
    Map<String, Item> fieldMap = acroFields.getFields()
    for (String fieldName : fieldMap.keySet()) {
        if (dataExtractMap.containsKey(fieldName)) {
            acroFields.setField(fieldName, dataExtractMap.get(fieldName))
        }
    }

    // "Flatten" the AcroForm with the bound field data - the fields will
no longer be editable
    pdfStamper.setFormFlattening(true)
    pdfStamper.close()
    return new DataDocument(stream.toByteArray(), CONTENT_TYPE_PDF)
}
```

Related articles

- PDF Receipting Options
- How to import an Acrobat PDF form as a Transact receipt template (AcroForms)
- Generating Multiple PDF Receipts from a Single Submission

How to import an Acrobat PDF form as a Transact receipt template (AcroForms)

Unknown macro: 'redirect'

Some organisations have invested heavily in creating glossy PDF forms which are designed to be printed, filled, and posted in, or alternatively opened, filled electronically, and saved and emailed. Sometimes these assets can be re-used to ensure that:

- downstream manual processing processes do not need to be modified, or
- customers are provided with a copy of their application which meets marketing guidelines
- Wet signature processes can be implemented, using copy which has already been ratified by legal

This guide describes how an existing "print & fill" PDF forms can be used by Transact as a pixel-perfect, "artwork PDF" template, without writing any code. This technique requires Adobe Acrobat (additional licenses); Other techniques exist and are not covered by this guide.

What is an AcroForm?

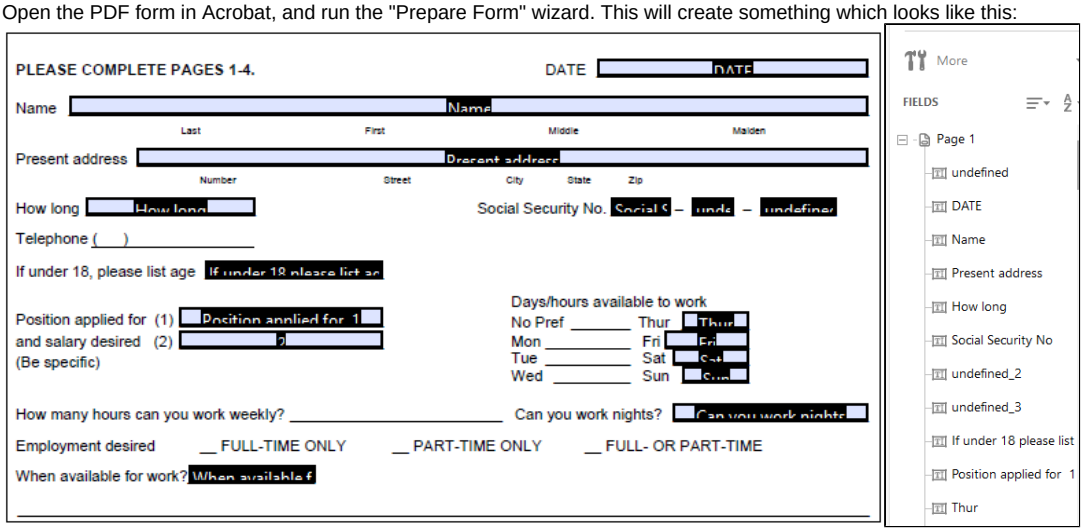
Forms which are created in Acrobat are called "AcroForms"; some PDF readers (Including Acrobat Reader) are capable of rendering these files as interactive PDFs, with features such as fields and radio buttons, and allows saving of data within the file. These should not be confused with interactive XFA forms which function in a specialized Adobe ecosystem (Adobe LiveCycle Designer ES & only Adobe Reader).

Avoka Transact re-purposes these PDFs as Templates associated with HTML5 forms on Transaction Manager, where HTML5 field data is merged with the template after submission for use as a Document of Record.

High Level Overview of steps required

At a high level, you need to follow these steps

1. Create your form in Composer, using the new form wizard. Drag and drop pages, sections, and fields, and add business rules from the palette, as you normally would. This produces your default dynamic PDF receipt, which will be discarded later. Publish your form to Transaction Manager, as you normally would.
2. Open the PDF form in Acrobat, and run the "Prepare Form" wizard. This will create something which looks like this:



3. In Acrobat, correct any issues. You may need to manually add fields, or rename them to make more sense. If you have authorisation to make changes to the form, now would be a good time to replace "comb" fields with regular fields, and remove any character place-holders such as parentheses in a telephone field, or slash characters in a date field.
4. In Composer, add Data Extracts to each field. The name of the data extract must match the name of the field declared in Acrobat. Additionally you will need to add some invisible fields (Data Fields) with calculation rules to reformat any dates or other fields which need conversion before printing.
5. In Transaction Manager, open the form version, and upload a receipt file. Then under Services, set the Receipt Render Service to "Static PDF Receipt".

Each of the steps 2-5 is described in detail below:

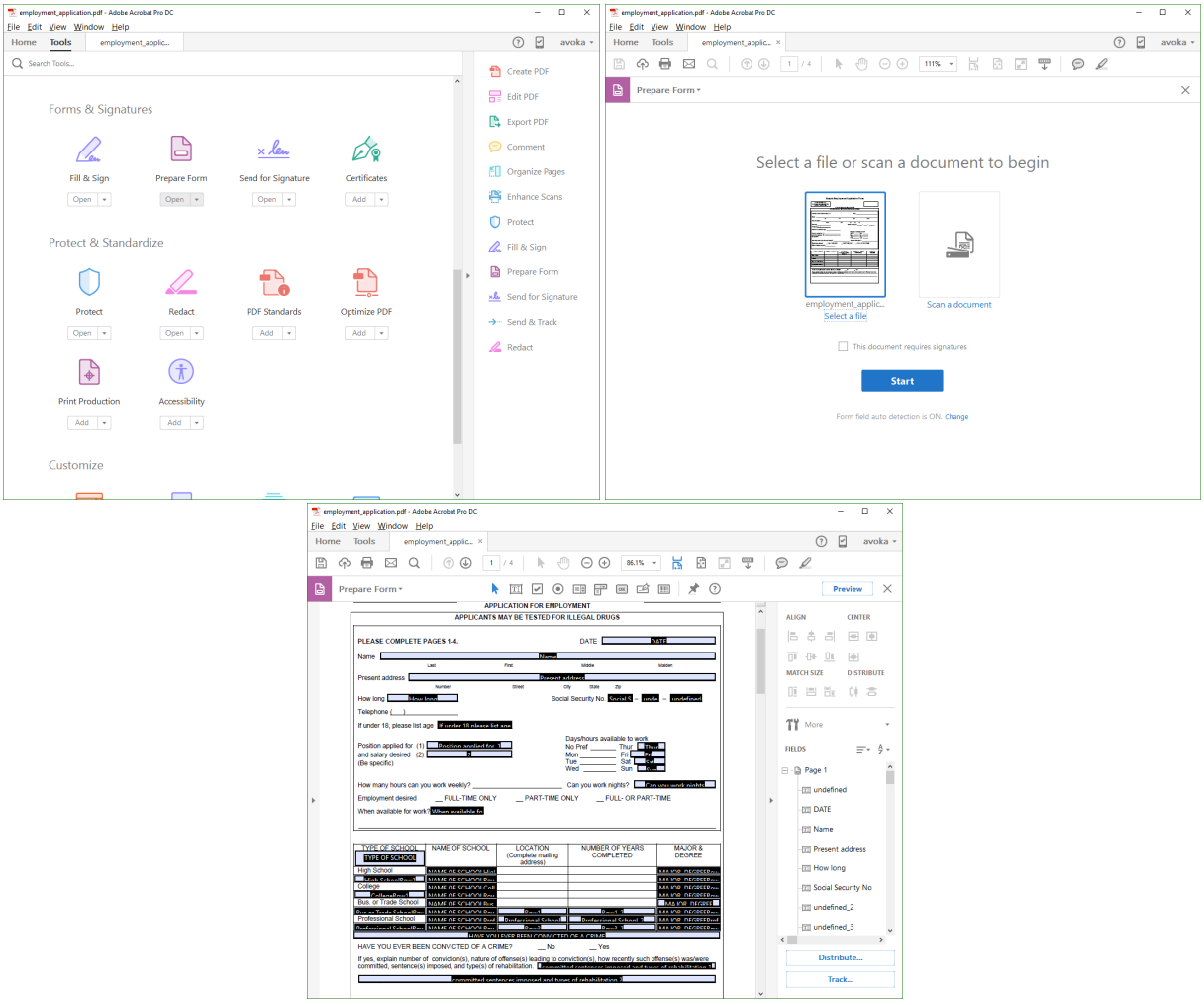
Step-by-step Guide

Step 2: Creating AcroForms

Adobe Acrobat Pro DC (2015 Release)

In order to create your AcroForm using Adobe Acrobat Pro DC (2015 Release) follow these steps:

1. Open the PDF form in Acrobat
2. Go to "Tools", and select "Prepare Form" (Open), under the "Forms & Signatures" heading
3. Select "Start" (take note that "Form field auto detection is ON" is displayed under the button)
4. You can now "Save As..." and continue to the next section.

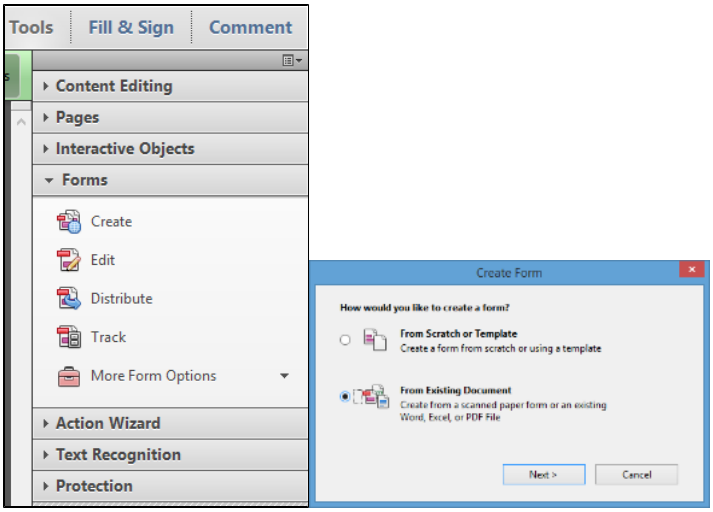


"Prepare Form" in Adobe Acrobat Pro DC

Acrobat XI and below

Acrobat 9 or Acrobat XI users can follow these steps:

1. Start, by accessing the "Forms" menu (under Tools, on the right-hand pane).
2. Select "Create", to start the wizard
3. You can now "Save As..." and continue to the next section.



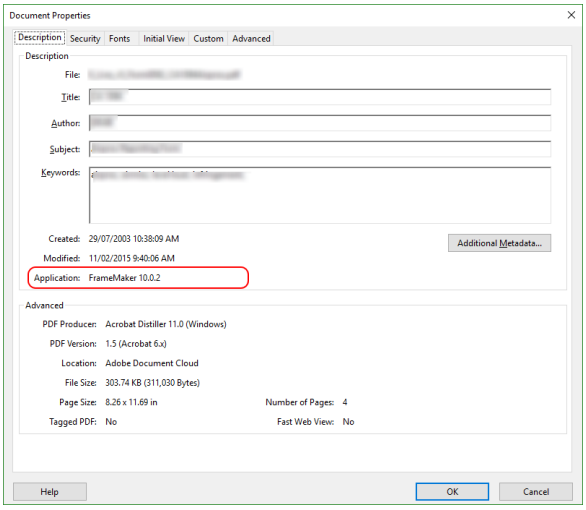
Creating a form in Adobe XI and below

Possible Issues & workarounds

If you have any issues with the above steps, you may need to check what type of PDF you have. There are many types of PDF, some of them may cause some issues. For example:

- Secured PDFs cannot be manipulated in any way without a password.
- XFA PDFs may have a .pdf file extension, but can only be edited with certain Adobe products (Adobe LiveCycle, not covered in this guide)
- Some PDFs may already be converted to an AcroForm, in which case you have less work to do.

The best way to get around any issues is to print the form to PDF using a non-Adobe product such as PDFcreator, CutePDF or your browser's built-in PDF engine. You may need to try some different products to see what works best for you. You can check which application was used to produce it by opening it in Adobe Acrobat (or Reader), and selecting "File" -> "Properties"



Inspecting the file properties in Acrobat

Another way around PDF security or unsupported file types is to either physically print them, or take screenshots, and then use Acrobat's OCR capabilities to scan in the pages.

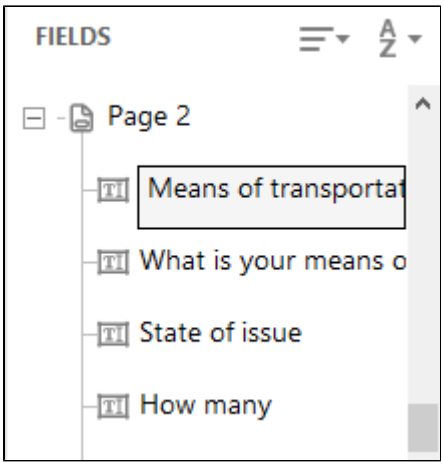
Step 3: Correcting automatic conversion issues

Automatically Detected Fields

For each automatically detected field, you will need to inspect to check:

- Field size, alignment & centring
- Font (Acrobat often chooses large and ugly fonts)
- Field names (it will become helpful to correct any auto-naming of fields to be something that is human readable)

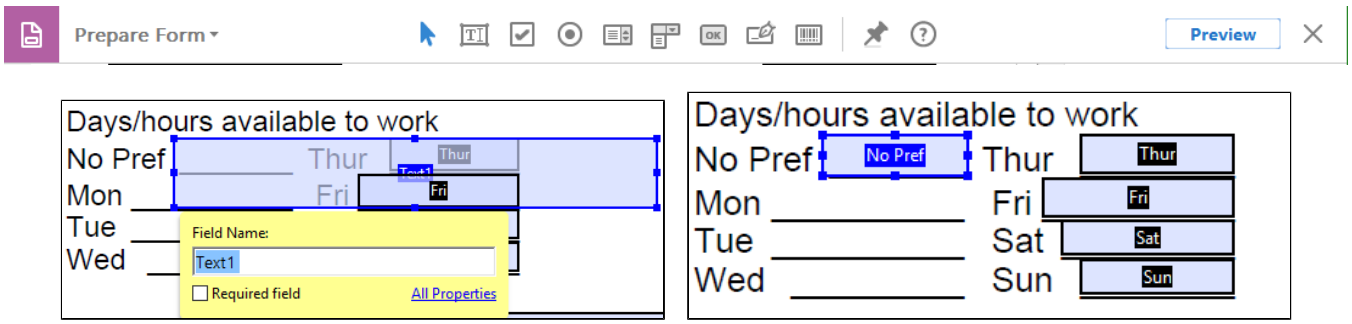
The toolbar on the right will help you quickly rename and reorganise all the interactive objects on the page. You can also bulk edit by selecting more than one field and right-clicking to access the properties panel.



Acrobat Pro DC - Fields Toolbar

Missed Fields

Sometimes the Acrobat forms wizard misses fields, and you will be required to add them manually. In Acrobat DC, there's a toolbar on the top of the page to add fields, checkboxes and radio buttons. Dropdown lists should be added as plain text field.



Adding a missing text field

When you are done, please think about "global" properties such as fonts. You can select all the fields on the form, and edit them in bulk, to ensure consistency.

Dealing with character-separated fields (Dates / phone numbers with area code)

One of the issues that you will encounter is how to deal with character separated fields, such as dates: " / / ". You have two options on how to deal with this: *Override* the PDF design or *re-factor* the data in Composer.

The easiest option and the option with the best result is to add fields with an opaque (coloured) background that cover up the character separators in the original PDF design. This may not be permitted in all circumstances, as the business may mandate that no changes are made to the layout (EG: OCR in downstream processing may fail if you change the exact layout).

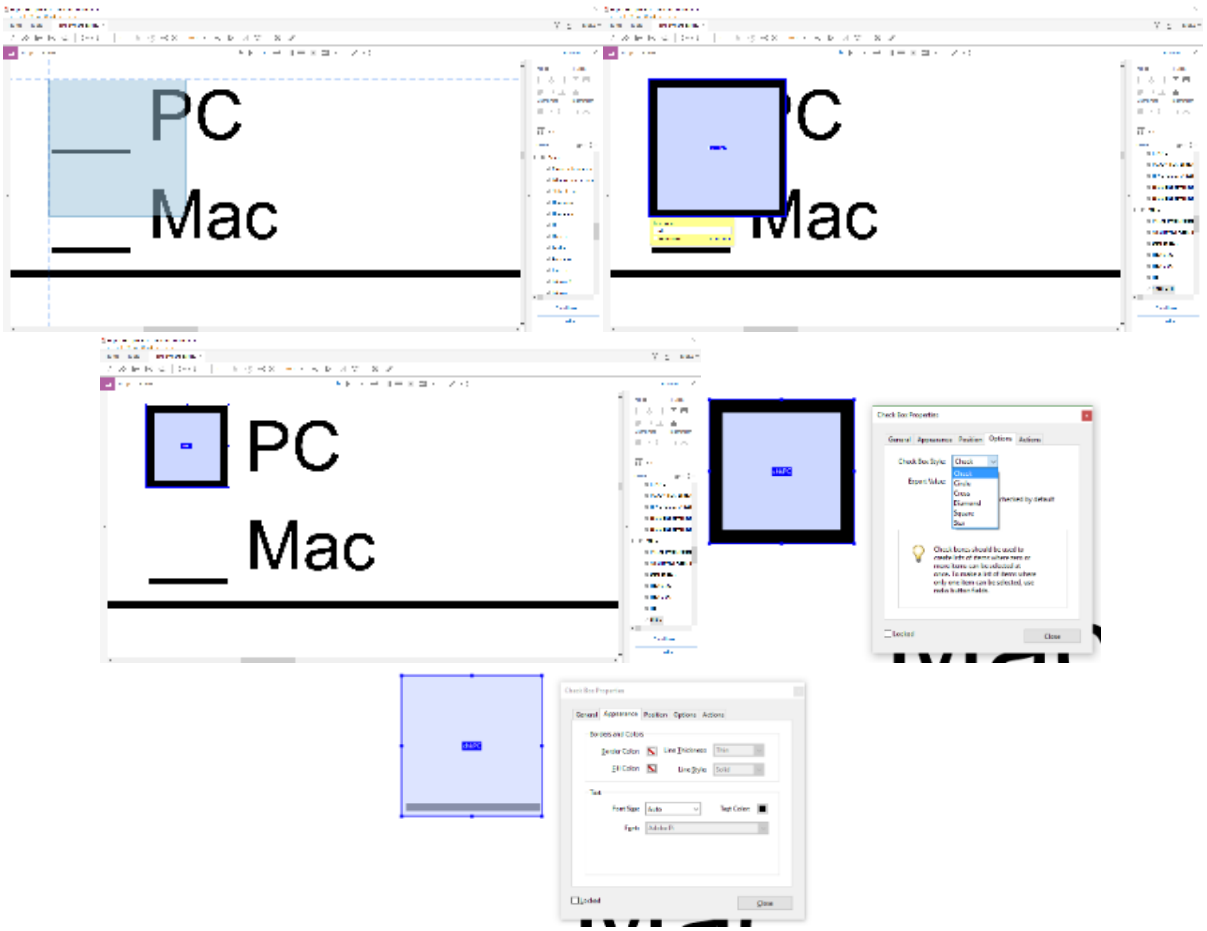
If you cannot override the field layout, then you will be required to add 3 separate text boxes for the date (Day, Month, Year), ensure that all values fit in the spaces created.

Re-factoring the format of field values

With some fields, you will be required to do some re-factoring of the field values. For example, Transact stores dates in a format of YYYY-MM-DD, and you will want to re-order this to be human readable, or to be displayed in 3 separate fields (pre-formatted character separators). This configuration is performed in Composer, and is described in the next step.

Check Boxes

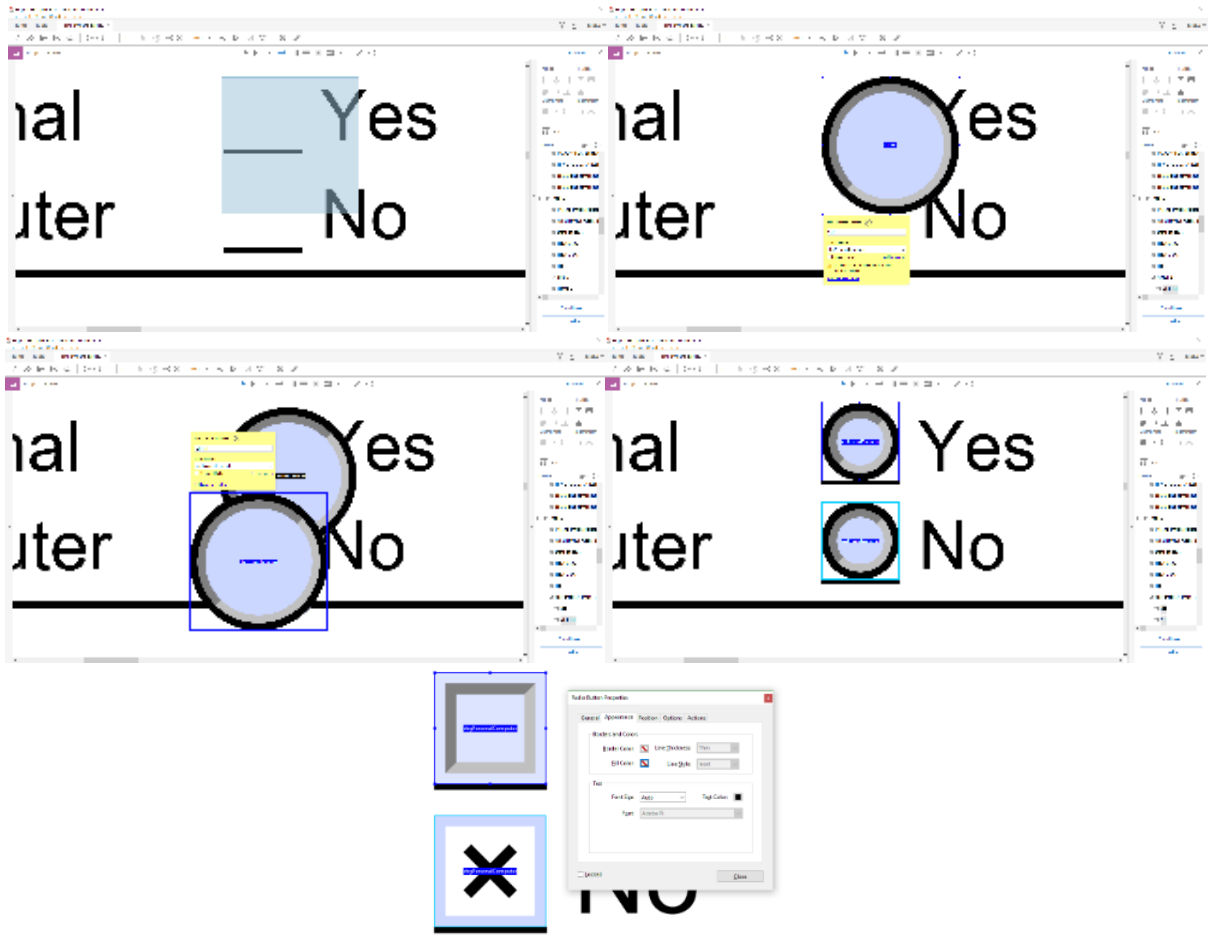
Check Boxes are quite straightforward. Simply add them, style them, and name them appropriately (Choose a convention which makes sense, such as prefixing the name with "chk"). You will need to resize them to fit in the correct area, and the best way to achieve a neat and accurate layout is to zoom right in to the part of the page where you're going to place it. Acrobat allows you to align, centre, match size and distribute objects for perfect layout. By default, Acrobat uses a check (tick) character within a box, but you can change this to a cross by editing the Check Box Properties, Options tab: Change the "Check Box Style" configuration (You may want to "Bulk edit" this setting, by selecting all your check boxes to edit). There are many layout options including border and fill colour.



Adding Check Boxes to a form & configuring the appearance

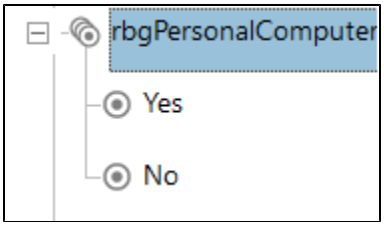
Radio Buttons

Radio buttons require membership in a radio button group. Acrobat attempts to help you with this by providing a wizard which helps you add multiple options in a list, but you will most likely need to re-factor the configuration to fit into a structure in the "Fields" editor. There is a limitation here: The name of the Radio Button Group should not contain spaces in order to be compatible with Composer.



Adding a radio button group and configuring appearance

When you are done, your Fields tab should look something like this:

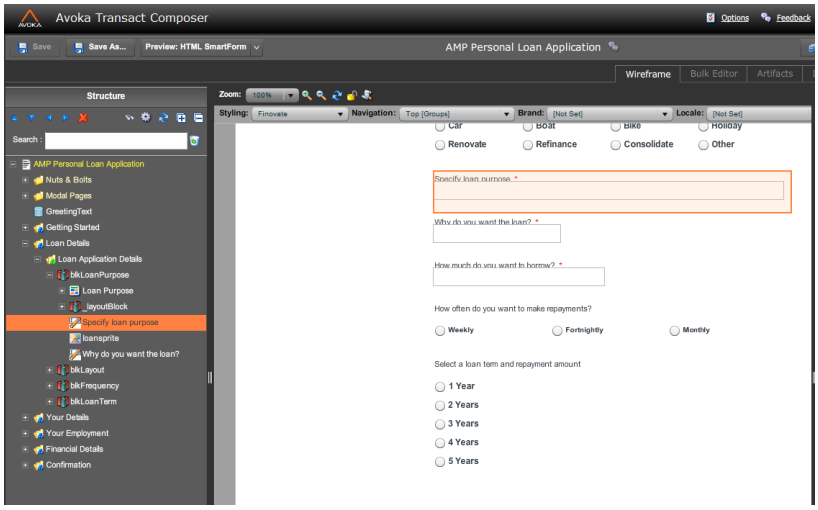


Step 4: Composer Configuration

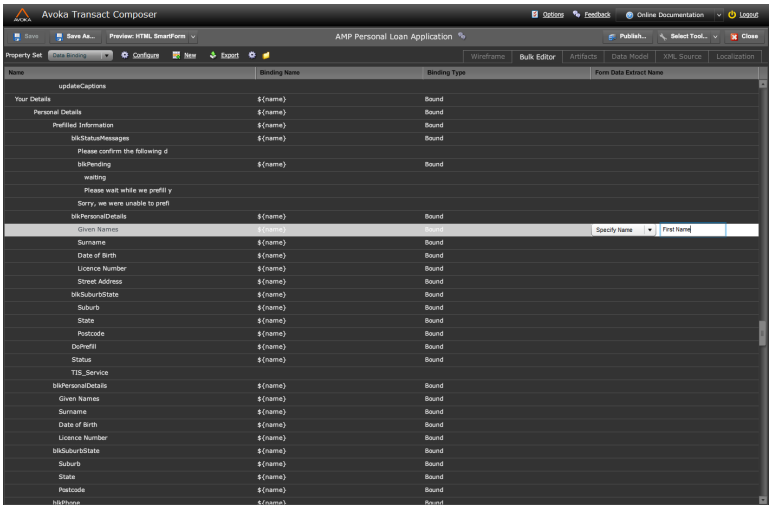
In Composer, you will be required to perform two tasks:

1. Bind Composer fields to PDF fields using Data Extracts
2. Re-factor any formatting of field values (such as dates)

To configure a data extract, you can either inspect the form wireframe/structure, or you can use the bulk editor, which displays the form fields in a table. The bulk editor is a great way of setting Data Extract properties in a tabular interface. To enable it, select the "Bulk Editor" tab, then change the property set to "Data Binding". Scroll down past all the internal fields (at least 10 screen pages), then click in an empty space for the drop-down list to appear.



Setting the Data Extract Name using the wireframe and property editor



Setting the Data Extract Name using the Bulk Editor

To configure data extracts from the wireframe/structure, double click on the field, then go to "Data Model", and set the "Form Data Extract Name".

Whichever process you choose, it is important to:

- Select "Specify Name" in the drop-down list
- Type in the name of the field. This name must match the name defined in the PDF form defined in Step 3 above.

Re-factoring the format of field values

With some fields, you will be required to do some re-factoring of the field values. For example, Transact stores dates in a format of YYYY-MM-DD, and you will want to re-order this to be human readable, or to be displayed in separate fields to be printed in pre-formatted fields with character separators. Follow these steps to ensure data is in the format expected by the PDF form.

1. Add a "Data Field" to your form, adjacent to the field in question. A data field is just an invisible field.
2. Add a scripted calculation rule to the data field.
3. Add some JavaScript to perform to parse and format the date or alternatively add string operations on the dependent field (See sample code below)
4. Add the Data Extract to the Data Field instead of the original user interface element

Reformatting using the Composer library

```
return sfc.parseDate({date}, "DD-MM-YYYY").format("DD/MM/YYYY")
```

The above line of code encapsulates several operations in one line. If you are new to JavaScript, you may wish to analyse the sample code below, which achieves the same outcome using string operations

Example code (for beginners)

```
var date = {date};                  // Link to date field as a dependency
var dateArray = date.split("-");    // "YYYY-MM-DD"  ==>  Array [ "YYYY", "MM", "DD" ]

var day   = dateArray[2];           // The 3rd element in the array "DD"
var month = dateArray[1];           // The 2nd element in the array "MM"
var year  = dateArray[0];           // The 1st element in the array "YYYY"

return day + "/" + month + "/" + year;  // "DD/MM/YYYY" (adding strings will concatenate them)
```

You may wish to extend this example with other types of fields. E.G.: you can use the same technique to re-factor phone numbers (use str.substring(0, 1); to get the first 2 digits in the string).

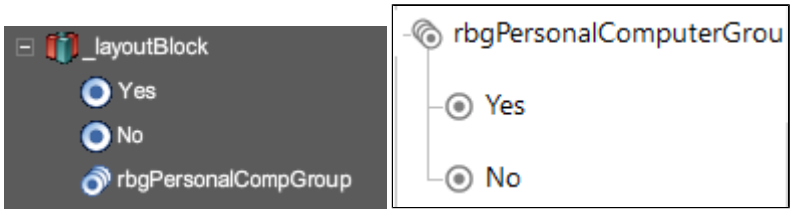
Radio Buttons & Check Boxes

Check boxes in Composer should be configured the same way as text fields: with the Data Extract name matching the AcroForm field name.

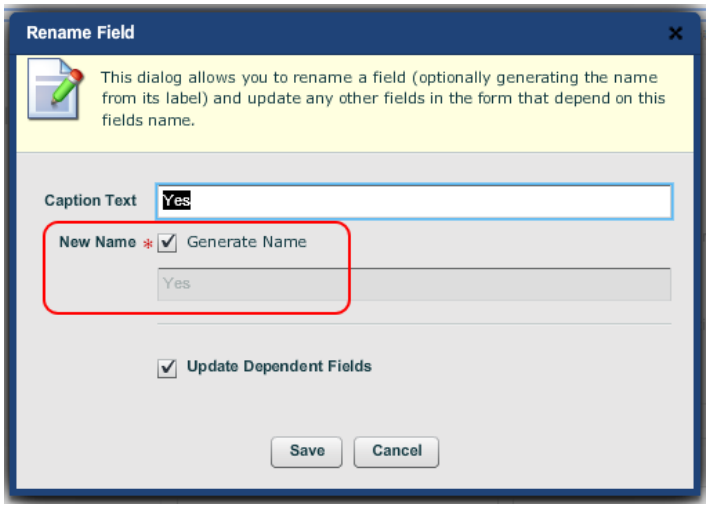
Radio buttons require a little extra care, ensuring that:

- The Composer Radio Button Group **data extract** matches the AcroForm Radio Button Group **name**
- Each Composer Radio Button **name** (not caption) matches the AcroForm Radio Button **names** exactly (check this for each radio button)

It's usually easier to perform modifications in Acrobat, conforming to Composer design.



Composer structure and Acrobat structure with matching names

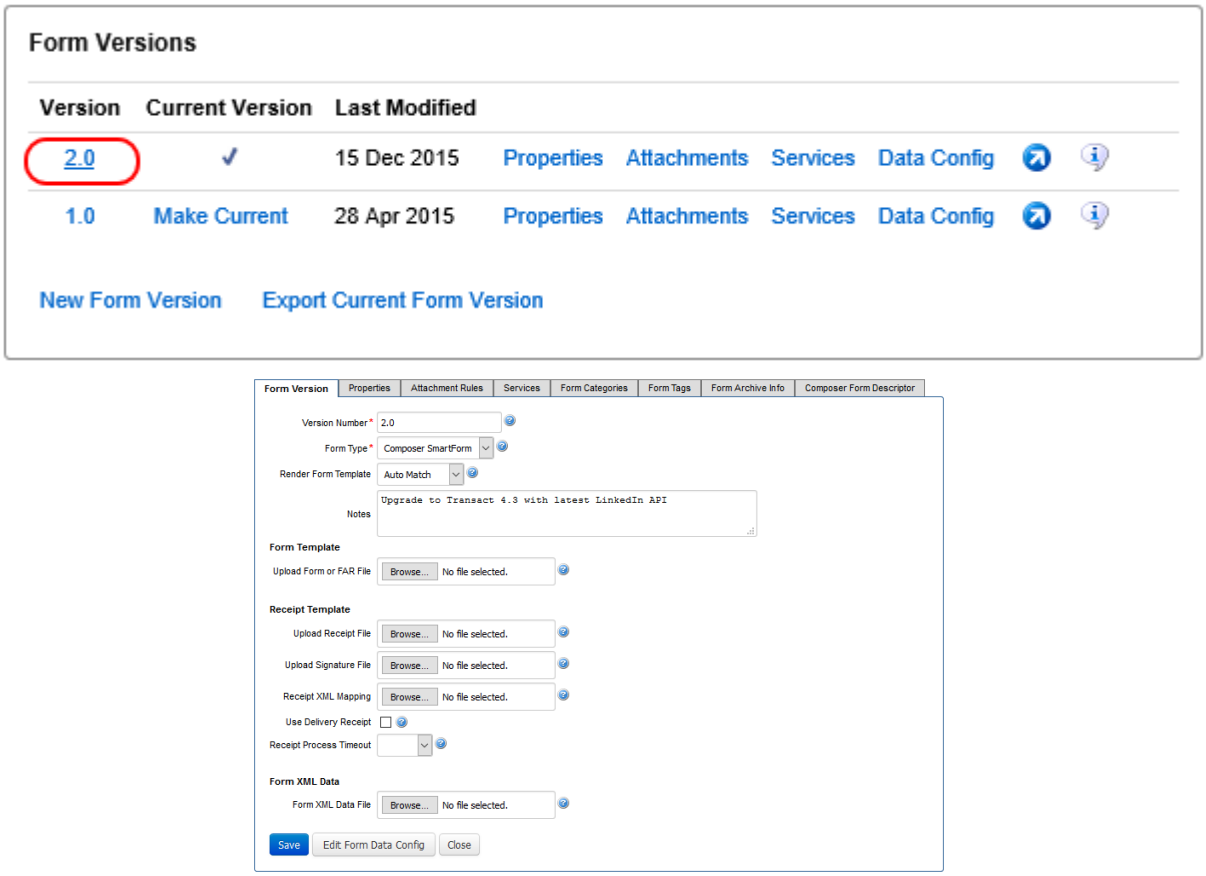


*Note that the **name** of the field does not necessarily match the **caption***

Step 5: Transaction Manager Configuration

When you have finished editing the AcroForm, it can be uploaded into Transaction Manager for use as a PDF Template for the receipt. First though, please ensure that you have published any changes out of Composer.

In the form dashboard, you can upload a Receipt file using the form version dialog.



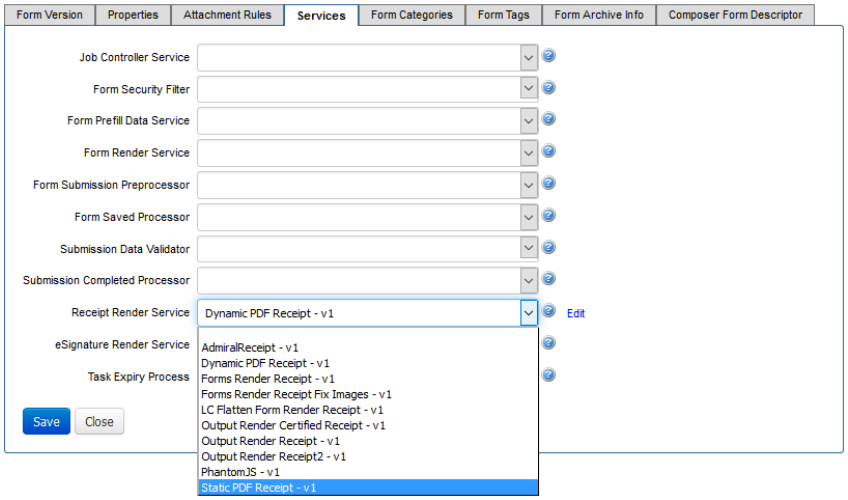
Transaction Manager Form Dashboard (Form Versions) and Form Version dialog

Follow these steps to import the file:

1. Log into Transaction Manager
2. Under Forms, find the form you wish to update
3. When you select the form you will see the Form Dashboard. On the top-right corner, you will see the Form Versions dashboard element (above)
4. Select the current version (highlighted in red above)
5. This will display the "Form Version" dialog. Under "Receipt Template", select the browse button adjacent to "Upload Receipt File".
6. Attach the AcroForm PDF file and save

Now that the file has been uploaded, it needs to be used by the receipt service:

1. On the form dashboard, in the form versions dashboard area, select the "Services" link adjacent to the latest version.
2. Change the Receipt Render Service to "Static PDF Receipt"



The "Receipt Render Service" selector

Extending this solution

A code-free solution can be great for rapid prototyping or building business-oriented solutions quickly, without relying on developers to contribute to the project. If you have access to Groovy developers, you can skip step #3, and and build server-side scripts to link form data Xpaths to AcroForm fields. Any data value re-factoring can then be performed in the custom script, rather than in Composer. Developers then have access to an extended suite of capabilities including:

- Concatenation of receipt with other static files (EG: PDS)
- Handling of attachments (as PDF attachments or pages)
- Enrichment of content (adding data from external services)

The full list of functions available to developers is available in Transaction Manager online documentation.

Related articles

Content by label

There is no content with the specified labels

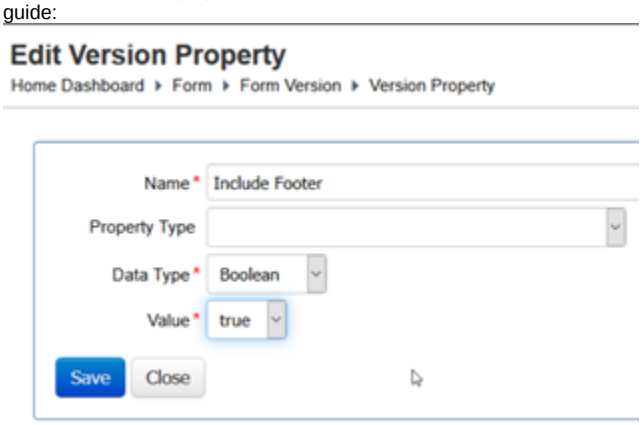
Inserting a Watermark in a PDF Receipt in V4.3.1

Unknown macro: 'redirect'

Demonstrating how to use a Groovy service to render the standard PDF receipt using iText to overlay HTML and thus creating a watermark on every receipt rendered.

Step-by-step guide

1. Publish your form to Transaction Manager. All the work is done in Transaction Manager so no additional work is needed in Composer.
2. Go to Services > Service Definitions
3. Import the Groovy Service : service-archive-Watermark_Receipt_Render_Service-2016-03-22.zip.
4. Add 2 form version properties called 'Include Footer' and 'Include Header' i.e. Form dashboard > Properties using the screenshot below as a guide:



5. Go to the form version > Services > and include the Receipt Render Service.

6. To change the position of your watermark, you may need to update the following code found in the GroovyScript:

```
def addHeader = addOverlayContent.curry(
getTemplate("Header HTML Template"),
 [envName: tmEnv],
 245, 40, 600, 832
);
```

Note: The coordinates above are: lower left X, lower right Y, upper right X, upper left Y

For further help on laying out your watermark, see http://api.itextpdf.com/itext/com/itextpdf/text/pdf/PdfDiv.html#layout(com.itextpdf.text.pdf.PdfContentByte, boolean, boolean, float, float, float, float)

7. Render and submit your form to test the generated receipt. Or use the Receipt Test Harness on the Form Dashboard page in TM.
8. If its not working, check the System -> Error Log. If you get the following error, check to see that you have created the 2 form properties.

Related articles

- Inserting a Watermark in a PDF Receipt in V4.3.1

PDF Receipting Options

Unknown macro: 'redirect'

There are a number of different ways to generate a PDF receipt (or document of record) in Avoka Transact. This article will outline each of the techniques, as well as presenting some pros and cons of each approach.

The options are:

- Transact PDF Receipts
- XFA Receipts
- Field Overlay Receipts
    - using Acrobat
    - using Livecycle Designer
    - using Texcel
- Custom Receipt Generation

This article does not recommend one of these approaches over another - the choice will depend on the requirements of each situation. Each solution has its own pros and cons. In addition, each tool used has a learning curve, and productivity will be determined by the skills of those using it.

Transact PDF Receipts

Transaction Manager is able to create a PDF receipt directly from the Transact form design. This receipt will be very similar to the interactive form, but with a number of modifications to optimize it for printing. These modifications include:

- Reducing the use of color (users often complain that it uses up too much printer ink)
- Removal of the navigational menu, so that the form becomes a single long multi-page document.
- Removal of all interactive features such as buttons and drop-down lists.

The exact optimizations used can be modified by adjusting a Composer skin.

Pros

- Requires no additional configuration or effort. (It is built into the product.)
- Honors all business rules in the form, such as hide-show logic and repeats.

Cons

- Does not generate PDF/A compatible PDFs. This may be a requirement for some organizations.
- Has limited support for headers, footers, and pagination options.
- Some problems have been reported with layouts, particularly character spacing on Linux.
- Since interactive forms tend to be quite spaced out, this can result in lengthy receipts with multiple pages and low field densities.
- Has a different layout to existing paper forms. (This may be an issue for downstream systems or people. This may also result in additional approval steps.)
- Not suitable for the case where a customer partially completes a form interactively, prints it out, and then completes the rest of the form using ink. (This can be achieved with additional effort in the form design.)
- Since the length and layout of the document may vary based on hide-show rules and repeats, the location of fields on the page cannot be guaranteed. This may impact downstream systems such as OCR, imaging, or digital signatures.

XFA Receipts

Composer is able to generate a PDF in XFA format. This is a format suited to high-fidelity PDF documents.

Pros

- Has strong capabilities for producing high-fidelity PDF documents, including headers, footers, and pagination options.
- Text formatting and layout is good.
- Produces PDF/A receipts.
- Honors all business rules in the form, such as hide-show logic and repeats.

Cons

- Requires Adobe LiveCycle. This incurs additional costs, and additional effort to provision and maintain.
- Since interactive forms tend to be quite spaced out, this can result in lengthy receipts with multiple pages and low field densities.
- Has a different layout to existing paper forms. (This may be an issue for downstream systems or people. This may also result in additional approval steps.)

- Not suitable for the case where a customer partially completes a form interactively, prints it out, and then completes the rest of the form using ink. (This can be achieved with additional effort in the form design.)
- Since the length and layout of the document may vary based on hide-show rules and repeats, the location of fields on the page cannot be guaranteed. This may impact downstream systems such as OCR, imaging, or digital signatures.

Field Overlay Receipts

Instead of generating a complete form, it is possible instead to import an existing (flat) PDF form, and lay the data fields over the top of this existing form. Regardless of the technology used to do this field overlay, there are some pros and cons.

Pros

- High fidelity - the receipt will look the same as the existing PDF.
- No need for additional approval steps, as the receipt is the same as the existing document.
- Information density is high, because the existing PDF has probably already been designing to minimize the number of pages.
- This type of form can be used if there is a need to partially complete the form, print it out, and then complete it with pen and ink.
- Receipt PDF is identical to the existing PDF, which will minimize downstream implications, including retraining. It also eliminates the need for additional approvals.
- PDF/A supported.

Cons

- Additional development effect is required to perform the task of overlaying the data fields over the top of the existing PDF.
- Additional development effort is required to map the XML data fields into a flat field naming structure. This can be done either within Composer, or using a mapping file in Transaction Manager.
- Hide/show rules are not honored. All fields are always displayed.
- Repeats are only honored to the extent that they are repeated on the original form. The form designer must be careful not to allow more repeats that are presented on the PDF. Mapping of repeats can be complicated.
- Field lengths must be limited in the interactive form to ensure that they don't overflow the space allocated on the PDF.
- Multi-line text must be controlled carefully to ensure they they don't overflow the space allowed in the PDF.
- Requires very detailed work to accurately size and place fields so that they match the underlying PDF.
- To some extent, the ability to re-design the data entry experience is limited. In particular, no new fields can be added, since there is nowhere to place them.

Field Overlay Receipts - using Adobe Acrobat

Adobe Acrobat has the ability to overlay fields over the top of an existing flat PDF. This uses a technology known as AcroForms.

Pros

- Relatively inexpensive per-seat development license.
- Makes no modifications to underlying PDF, and so the receipt maintains 100% fidelity with the original PDF.
- Avoka Transact supports AcroForms technology with no additional provision or configuration.
- Has a field recognition feature, which allows fields to be automatically recognized. This can save a lot of time, particularly when large numbers of PDF forms need to be converted. The effectiveness of this feature varies widely, based on the specifics of particular documents. It is suggested that this feature be tested against a representative sample of documents before being used on large scale.

Cons

- The field creation and editing environment within Acrobat is considered by many to be quite rudimentary, particularly for larger forms, though this is subject to personal preferences.
- No ability to modify the underlying PDF document. (For example, to remove explanatory text that may be inappropriate, or to make minor modifications to the form.)
- If a new version of the underlying PDF document is produced, then the entire process needs to be repeated on the new document. Copying and pasting fields may help in some cases.

Field Overlay Receipts - using Adobe LiveCycle Designer

Adobe LiveCycle Designer has the ability to overlay fields over the top of an existing PDF document.

Pros

- Relatively inexpensive per-seat development license.
- Makes no modifications to underlying PDF, and so the receipt maintains 100% fidelity with the original PDF.
- The field editing and creation environment is reportedly better than Acrobat, though this is subject to personal preferences.

Cons

- Requires Adobe LiveCycle. This incurs additional costs, and additional effort to provision and maintain.
- No ability to modify the underlying PDF document. (For example, to remove explanatory text that may be inappropriate, or to make minor modifications to the form.)

- If a new version of the underlying PDF document is produced, then the entire process needs to be repeated on the new document. Copying and pasting fields may help in some cases.

Field Overlay Receipts - using Texcel

Texcel FormBridge is a third-party product that can intelligently recognize fields in an existing PDF document. In many cases, this results in a very quick conversion of existing PDF documents into AcroForms which can then be imported into Transaction Manager. Some manual cleanup is almost always required. This is particularly useful when large numbers of PDFs need to be converted.

In addition, Texcel will also produce a list of fields and their attributes. These field definitions can be imported into Composer, and uses as the basis for the form design in Composer. This can accelerate the production of forms in Composer, as well as automate the mapping of fields between Composer and the generated AcroForm.

The effectiveness of the conversion is dependent on the structure of the original PDF documents. It is suggested that this feature be tested against a representative sample of documents before being used on large scale.

Pros

- Automated recognition of fields can save a lot of time.
- Kills two birds with one stone. The recognized fields are used both to generate a AcroForm PDF, and to import into Composer as a starting point for building the interactive form.
- In addition, the boilerplate text such as instructions, product disclosures and legal copy are also able to be imported into Composer, reducing the amount of effort in doing this.
- Strong and capable editing environment.
- The underlying PDF document is converted into lines and text. This means that the PDF itself is editable, if this is required.

Cons

- Texcel incurs additional cost to convert each page of the original PDF.
- Because the underlying PDF is converted, there may be some differences between the original PDF and the generated one. The differences, if any, will depend on how the original PDF was constructed. Some configuration options in Texcel can be tuned, but ultimately, Texcel should be tested on sample documents.

Custom Receipt Generation

Any third-party tools or systems can be used to generate PDF receipts, as long as these systems have an interface that can be invoked from Transaction Manager.

Pros

- No limits as to what can be achieved.

Cons

- The third-party tools systems must be purchased, provisioned and managed.
- A Transaction Manager service must be developed that can invoke the receipt service.
- PDF receipts can be quite large. If the system is not hosted close to Transaction Manager, this may incur processing delays or bandwidth costs.

Prefill

Unknown macro: 'redirect'

- How to prefill Cascading Dropdowns from Transact Manager
- Prefill or prepopulation with anonymous users

How to prefill Cascading Dropdowns from Transact Manager

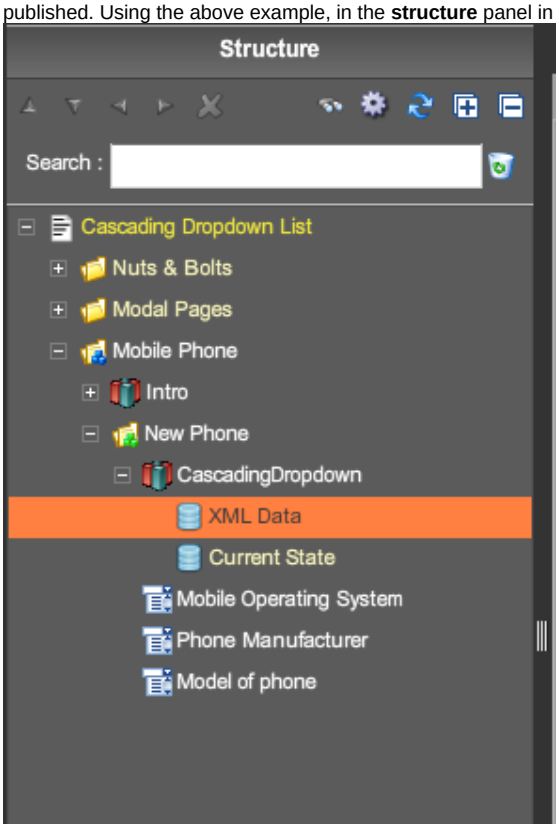
Unknown macro: 'redirect'

This guide will step you through how to update Xml data that defines a Cascading drop down in a Composer form from the form properties defined within Transaction Manager.

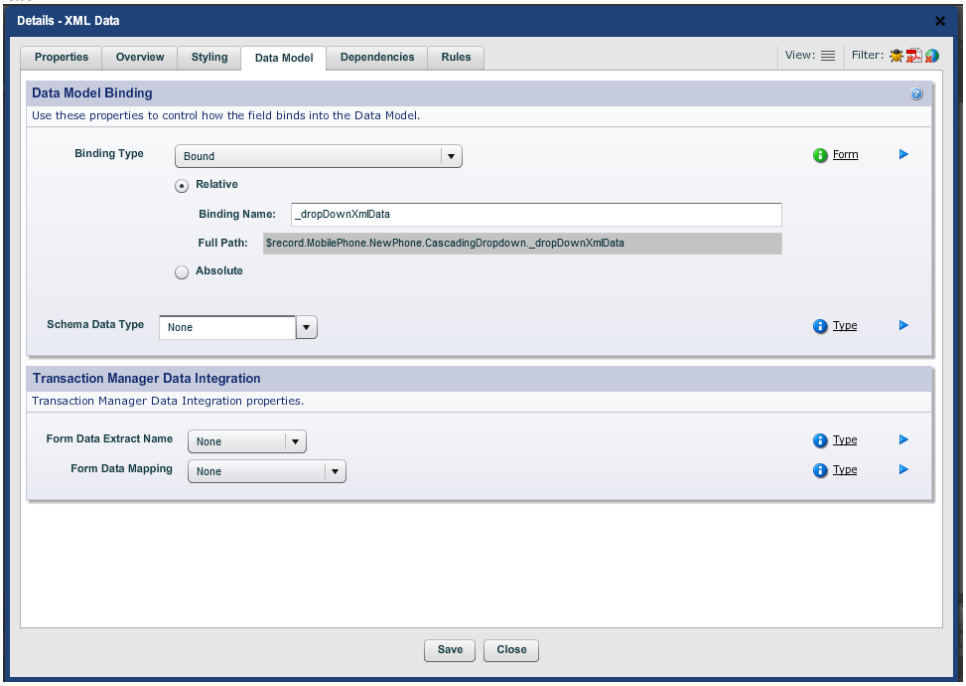
The form used in this guide was built following this guide: https://composer.demo.avoka.com/composer/doc/dropdown_lists.htm

Step-by-step guide

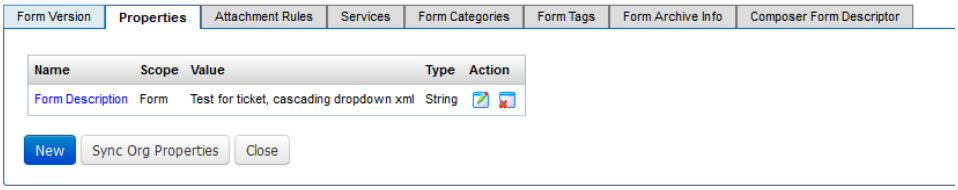
1. The first thing that you need to do is to adjust the setting for the xml data model bindings in the form. This is done in composer before the form is published. Using the above example, in the **structure** panel in composer navigate to the Xml data element.



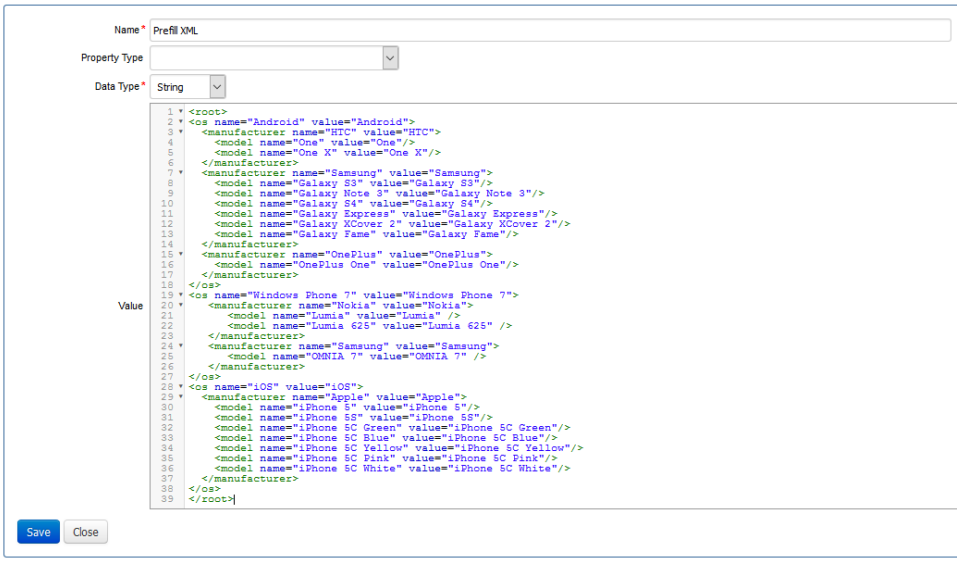
2. Open the **properties** for the xml data. Navigate to the **data model** tab and change the **binding type** to **Bound.** Note the path as you may need it later.



3. Now publish the form to Transaction Manager.
4. Navigate to your form in Transaction Manager and click on **properties** in the **Form Versions** block for the form you are making these changes to. Here we are going to add a new form property with some modified XML that will be used to update the form in Transaction Manager.



5. Now click new and fill in the **Name** and set the **data type** to **String.** A new field will become visible; this is where you add the modified xml that will be used to update the form. For this example, another manufacturer was added to the Xml from the original example used.

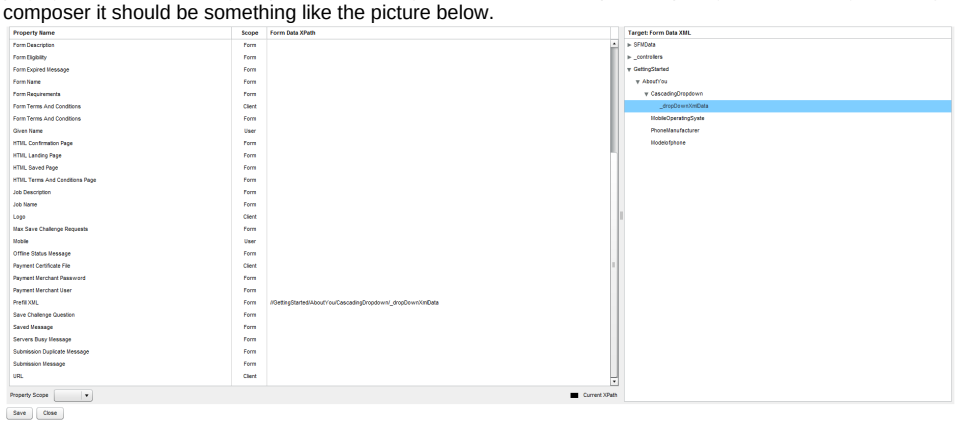


Prefill XML

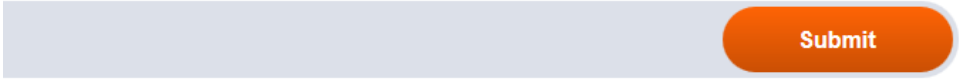
```
<root>
<os name="Android" value="Android">
  <manufacturer name="HTC" value="HTC">
    <model name="One" value="One"/>
    <model name="One X" value="One X"/>
  </manufacturer>
```

```
  <manufacturer name="Samsung" value="Samsung">
    <model name="Galaxy S3" value="Galaxy S3"/>
    <model name="Galaxy Note 3" value="Galaxy Note 3"/>
    <model name="Galaxy S4" value="Galaxy S4"/>
    <model name="Galaxy Express" value="Galaxy Express"/>
    <model name="Galaxy XCover 2" value="Galaxy XCover 2"/>
    <model name="Galaxy Fame" value="Galaxy Fame"/>
  </manufacturer>
  <manufacturer name="OnePlus" value="OnePlus">
    <model name="OnePlus One" value="OnePlus One"/>
  </manufacturer>
</os>
<os name="Windows Phone 7" value="Windows Phone 7">
  <manufacturer name="Nokia" value="Nokia">
    <model name="Lumia" value="Lumia" />
    <model name="Lumia 625" value="Lumia 625" />
  </manufacturer>
  <manufacturer name="Samsung" value="Samsung">
    <model name="OMNIA 7" value="OMNIA 7" />
  </manufacturer>
</os>
<os name="iOS" value="iOS">
  <manufacturer name="Apple" value="Apple">
    <model name="iPhone 5" value="iPhone 5"/>
    <model name="iPhone 5S" value="iPhone 5S"/>
    <model name="iPhone 5C Green" value="iPhone 5C Green"/>
    <model name="iPhone 5C Blue" value="iPhone 5C Blue"/>
    <model name="iPhone 5C Yellow" value="iPhone 5C Yellow"/>
    <model name="iPhone 5C Pink" value="iPhone 5C Pink"/>
    <model name="iPhone 5C White" value="iPhone 5C White"/>
  </manufacturer>
</os>
</root>
```

6. Hit save and navigate back to the starting form page. Now click **Data Config**, located in the same field as Properties from earlier. In data config navigate to the **Property Prefill Mapping** tab. Now click the edit button.
7. On this page you will see a list of properties in a large table. Scroll down till you see the one you created earlier. On the right hand side of the page you will see the Targets pane click through the links there to open the path you saw when you changed the bindings for the Xml data in composer it should be something like the picture below.



8. You now want the drag the _dropDownXmlData over to the property you created to make the link. Click save and then save again when it returns to the data config page. Now return to the form page. Now when you load the form in Transaction Manager it will reflect the changes made to the XML.



Related articles

- Transaction Delivery
- Transaction Reporting Feed
- TField and Locations
- Collaboration Jobs - Rolling Back To a Previous Step (State)
- test

Prefill or prepopulation with anonymous users

Unknown macro: 'redirect'

Problem

We have several out of the box features (e.g. Save and resume, task assignment) for anonymous users. How do we solve the problem of pre-population or prefilling of data for anonymous users?

This article looks at several options for different use cases. The important thing to remeber here is which of these are safe to use for Personally Identifying Information (PII) and which are only to be used to pass non PII data such as a product choice on a website.

"Anonymous" is actually a bit misleading. The user might be authenticated in another system, they are just not a registered user in TM.

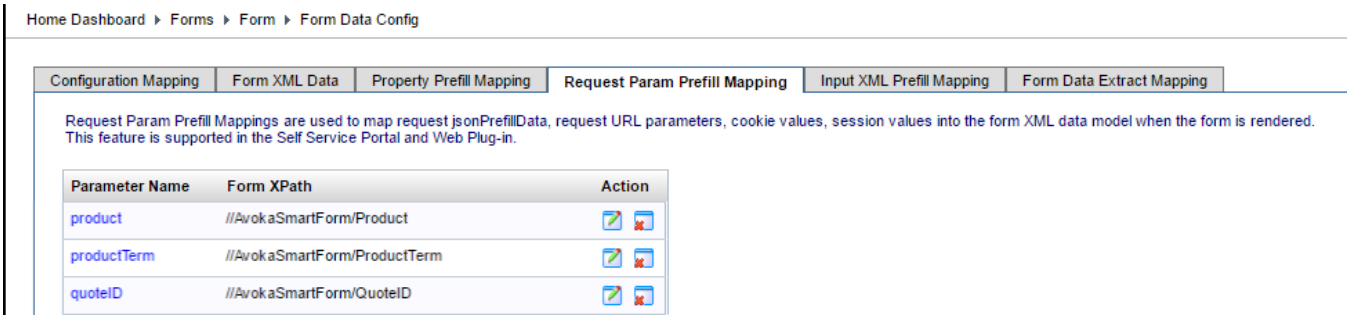
Solution

There are several options when tackling this problem.

- URL Parameter passing on a GET request
- HTTP request headers on a POST request
- Cookies
- Short life tokens
- Encrypted string

URL Parameter passing on a GET request

It doesn't get much simpler than this. All you need to do is add some form prefill mappings using the same name as the URL parameter you use. e.g.



Would work with a url such as https://tm.demo.avoka.com/test/servlet/SmartForm.html?formCode=cgf-apply-ga&product=single&productTerm=Fixed&quoteID=GTM495780

Pros

Very simple

OOTB

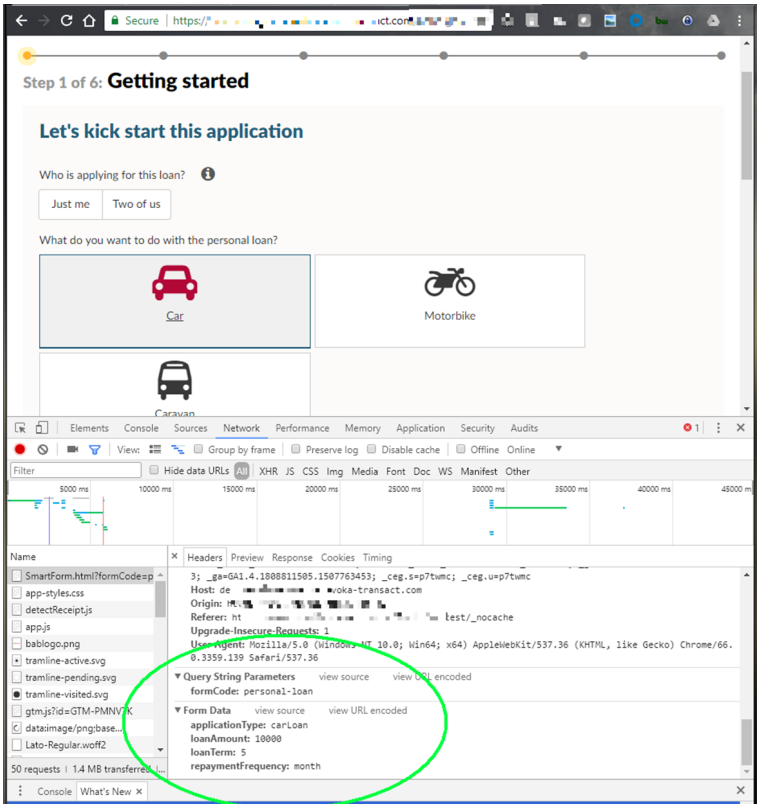
Cons

It should only be used for non sensitive or PI data i.e. you should not be passing data such as the customer name

HTTP request headers on a POST request

This is pretty much the same mechanism as the above method but the data name/value pairs our provided as header attributes on the POST request. This implementation is very common in our solutions.

E.g.



The TM config is exactly the same.

Pros

Simple

OOTB

Slightly more sophisticated than URL parameters

Cons

It should only be used for non sensitive or PI data i.e. you should not be passing data such as the customer name

Encrypted Cookies

This is not an out of the box option and requires the creation of a custom form component and two TM groovy services. For full details on how to implement this type of integration contact Kevin Mortimer

The service requires three components:

1. **Cookie Manager** (Composer/Maestro widget) - added to the form
2. **XXXCLIENT Cookie Decryption** (Transact Dynamic Data service) - called by the Cookie Manager widget
3. **XXXCLIENT Cookie Encryption** (Transact Submission Completed Processor service) - associated with the form

How it operates:

- **XXXCLIENT Cookie Encryption**: extracts form values from the "**XXXCLIENTData**" node and populates an encrypted cookie named "**XXXCLIENTData**".
- **Cookie Manager**: holds data fields populated from the encrypted "**XXXCLIENTData**" cookie. Also hold a business rule that invokes the "**XXXCLIENT Cookie Decryption**" dynamic data service to decrypt the contents of the cookie. The business rule is only invoked once during the form's initialisation phase.
- **XXXCLIENT Cookie Decryption**: Dynamic data service invoked via form business rule within the Cookie Manager widget.

On submission the data from the form is encrypted and made available to any subsequent form in the solution that the user might access.

Pros

Fairly secure

Cons

Relies on the next form being opened on the same device

Short life tokens (SLT)

There are often several client systems from which secure Personally identifiable information (PII) can be sourced. For this reason, Avoka can provide a simple reusable prefill implementation that can be consumed by several “client system” sources e.g.

- Online Banking
- Customer portals
- CRM

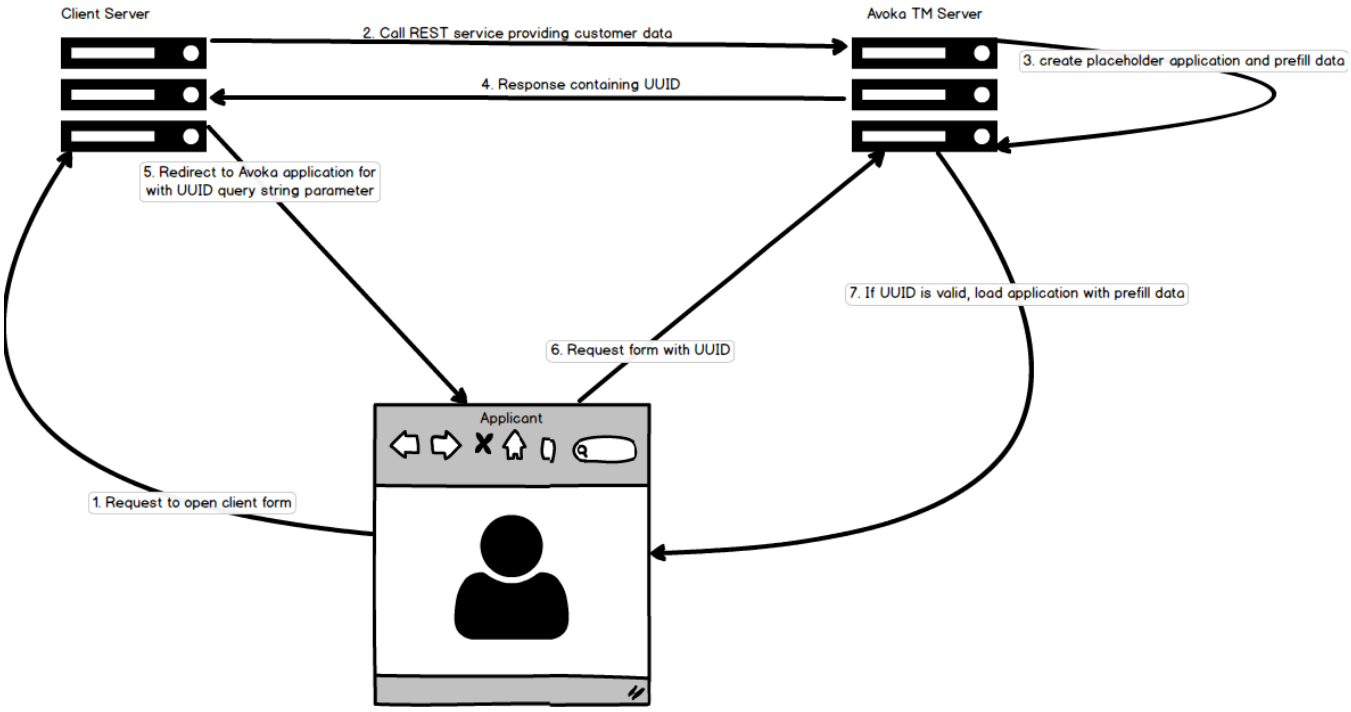
Avoka will expose a RESTful service to enable the “client system” to push customer data to Avoka when a customer requests the form from within the “client system” application. Avoka will pass back a unique identifier (a UUID) that will enable the client system to build a URL request that will open the Avoka form with the prefilled data.

For pre TM 5.0 implementations a "container" submission will be used to store the data. Since TM 5.0, storing the temp prefill data in MemCache for the very short time is possible and is a neater design than creating a temporary submission object to hold the prefill data for a short time frame.

The token will be short lived, if the UUID (submitkey) passed is for a container submission older than 30 seconds (configurable) Avoka will load an empty form. The container will be removed.

If the form prefill is successful, the form will load, and the container will be removed.

A tidy up task will be scheduled to remove any containers older than 1 minute.



If the RESTful customer data service (2) is unsuccessful the UUID will not be returned.

If the form prefill service in Avoka fails for one of the reasons below, Avoka will load an empty form.

For the prefill service call (2), IP whitelisting and basic auth. will be used to secure access.

Several URL parameters will need to be passed with the Request (5):

Example prefill data provided could be:

1. Salutation
2. First Name
3. Last name
4. Date of birth
5. Address

6. Mobile
7. Email
8. etc.

Pros

secure option

single prefill service required in Avoka

securely pass data from multiple sources

Cons

does require some work by the client

For full details on how to implement this type of integration contact Unknown User (jbasey), Miroslav Botka or Suraj Silva.

Encrypted string

This is an enhancement on the prefill mapping GET or POST request, rather than sending name value pairs in clear text you encrypt all your prefill data into a single string. This is not a common implementation.

```
Generally the outline is as follows:
 # A public/private key pair is generated for the TM instance. The encryption (public) key is shared with the
external calling site/portal.
 # The external (referring) website/portal generates the secure token and adds this to the HTTP request, either
in the payload data or as a request cookie.
 # The TM prefill service parses the token from the request, decrypts and prefills the form per BAU.

The encryption / decryption code is largely boiler-plate Java code and easily transportable, e.g.:
/**
 * Encrypts or decrypts the specified byte value
 * Presumes that the decryption value has been Base64 decoded already, e.g.:
 * def URLCodec urlcodec = new URLCodec()
 * def byte[] value = urlcodec.decode(aString).decodeBase64()
 * Presumes that the encrypting value is in UTF-8 encoding, e.g.:
 * byte[] value = aString.getBytes("UTF-8") *
 * @param value
 * @param encrypt
 * @param keystore
 * @param keystorePass
 * @param keyAlias
 * @param aliasPass
 * @return byte[]
 */
private byte[] encryptDecrypt(byte[] value, boolean encrypt, byte[] keystore, String keystorePass, String
keyAlias, String aliasPass) {
 // access the keystore and load the decryption key
 KeyStore keyStore = KeyStore.getInstance(KeyStore.defaultType)
 ByteArrayInputStream bais = new ByteArrayInputStream(keystore)
 keyStore.load(bais, keystorePass.toCharArray())
 Key key = keyStore.getKey(keyAlias, aliasPass.toCharArray())
 int keyBuffer = key.getModulus().bitLength() / Byte.SIZE
 Certificate certificate = keyStore.getCertificate(keyAlias)
 Cipher cipher = Cipher.getInstance(key.algorithm)
 // encryption
 if(encrypt) {
 cipher.init(Cipher.ENCRYPT_MODE, certificate.publicKey)
 ArrayList encryptedArray = []
 value.toList().collate(keyBuffer-11).each() { bytes ->
 def encryptedData = cipher.doFinal((byte[])bytes)
 encryptedArray.addAll(encryptedData)
 }
 return (encryptedArray as byte[])
 }
 // decryption
 else {
 cipher.init(Cipher.DECRYPT_MODE, key)
 StringBuffer decryptedStr = "" << ""
 value.toList().collate(keyBuffer).each() { bytes ->
 def decryptedData = cipher.doFinal((byte[])bytes)
 decryptedStr << new String(decryptedData)
 }
 return decryptedStr.toString().getBytes("UTF-8")
 }
}
```

Pros
- Fairly secure

- prefill data is encrypted.
- it can be a little more flexible to a roll-you-own solution without resorting to more structured models (like SAML).

Cons
- for larger payloads, encrypted data will need to be sent as a POST request.
- encryption keys need to be maintained (on both sides).
- prefill data is encrypted. Debugging and operational support becomes more difficult.

Related articles

Form Prefill Data

Prefill

Integration

- About Avoka Transact Integrations
- Integration Capabilities in Transact
- Strategies for integrating forms to existing websites
- Google Analytics Virtual Page Integration
- How to add Google Tag Manager support
- How to switch address lookup provider
- How To Open TaskUserWebService in SOAP-UI
- Transaction Manager Integration with Salesforce
- Transact Manager - Sharepoint Integration
- How to decrypt Json Web Token (JWT) in TM

About Avoka Transact Integrations

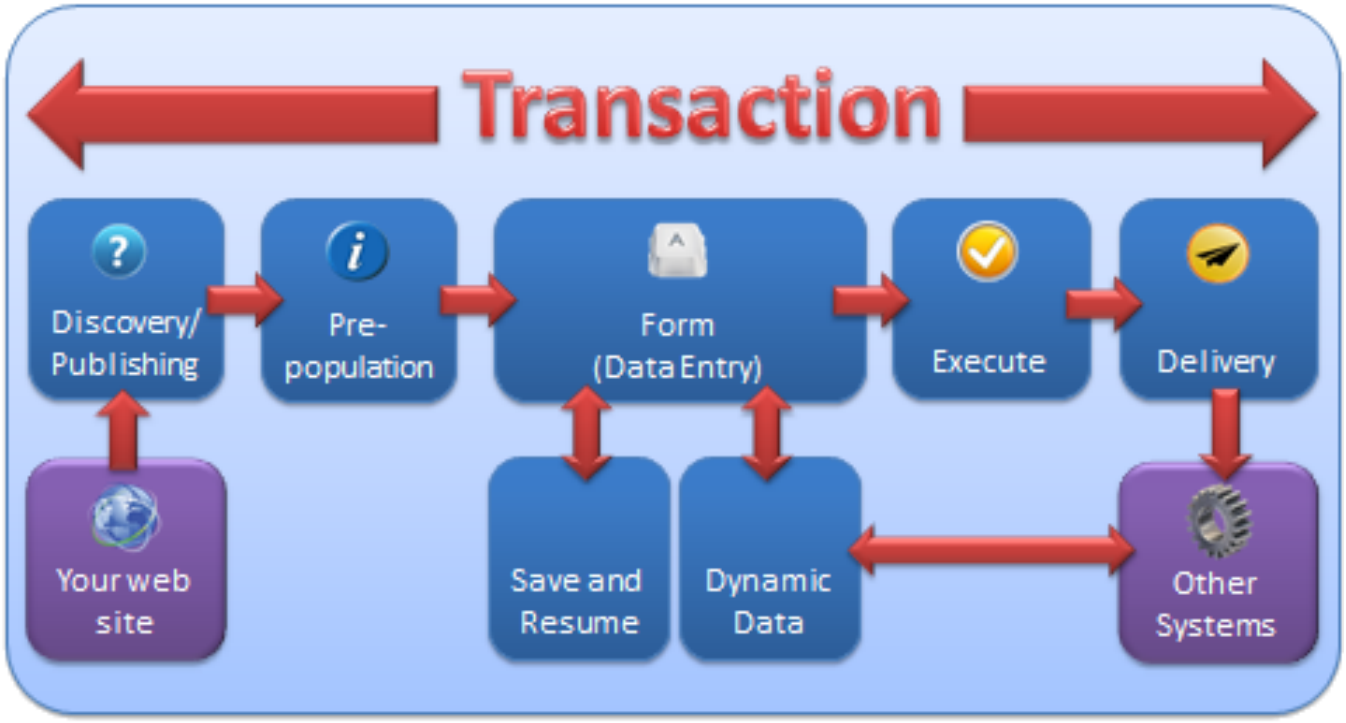
Introduction

Avoka Transact enables digital interaction between an organization and its customers. Because of the importance of this interaction, there is often a need to integrate between Transact and other systems. In some cases, the integration required can be minimal, but integration can be quite extensive.

- Integration areas occur at different stages within the interaction between a user and the organization.
- Within each integration area, there are different integration implementations, to support different technologies used by different organizations.

Integration Areas

The diagram below illustrates the 5 key areas in which integrations generally occur during transactions.



Stage	Overview	Details
Discovering Transactions	Your users need to be able to discover the transactions that you provide. There are a number of ways to do this. The simplest is to have a link on your web site which launches the transaction.	Things can get more sophisticated than this. For example: • You may want to have customized links, which are different for different users. Generally this is so that you can provide custom pieces of data in the link URL, generally so that you can pre-populate some of the fields in the form from this data. This helps to make it easier for users to fill out forms (see Pre-population below). • If your users are logged into your web site, you may want to pass their identity to the transaction. This again allows the form to pre-populate some of the fields. • You may want to embed the form into your existing web site, rather than linking to it. This is generally not recommended in modern responsive and task-oriented designs, although it is possible to implement with Transact. • Another option is to send the user a link to the transaction using some other mechanism, such as via an email. You can send them a link to a blank form, alternately you can also use Transaction Manager to create a custom link that contains a partially pre-filled form to make their lives easier. • Users can also save a transaction that they have partially completed. This will generate a custom link for them, that will enable them to return to the transaction at a later date. There are also other ways for them to get back to the

		transaction if they lose the link. You can configure a challenge question to ensure that only the original author can access their saved transaction.
Pre-population	We want to make it as easy for your users to complete a transaction as possible. Transact is specifically designed to make the entire experience as intuitive and simple and engaging as possible. You can make the data entry process even simpler by pre-populating (or pre-filling) information into the form that you already know about them, which means that they won't have to type the information themselves. Depending on what information is available, this can dramatically	We can generally provide the pre-fill data in one of the following ways: • Front-end data passing. If your existing web site knows who the user is, you can pass specific pieces of data about that user directly to the transaction. This can be done via URL parameters, cookies, lookup tokens, and a variety of other techniques. • Back-end lookups. If the user has been authenticated by your web site, you can pass their identity to the Transaction Manager server. We can then use this identity to look up information in back-end business systems, or Transaction Manager's own profile database, to pre-fill the form. Pre-population is also sometimes provided by using a previous submission's data to populate a new submission. This is useful when a transaction has to be repeated every year (or periodically). and the data doesn't change much from submission to submission.
Streamlining Data Entry	During the data entry process, we want to make it as easy as possible for users to complete the form, and get the product or service they are requesting. There are a number of ways we can do this. For example: • We can populate fields in the form from third party services, such as LinkedIn or Facebook. (This is similar to pre-population, but occurs once the form has already loaded, rather than prior to loading.) • We can do calculations and lookups to provide them more information, or to validate their data entry. For example, we might verify the name of the account holder that they are trying to transfer funds to, or calculate the approximate repayments for their loan. • We can provide them ways of speeding up their data entry. For example, instead of forcing them to type in an address laboriously, field by field, we can allow them to just start typing an address, and provide a matching set of possible candidate addresses, and then auto-complete the individual fields. (Like the address picker in Google Maps.) • We can ask the user to upload a document, such as a photo of their license. We could use OCR technology to recognize the text on the document, and used to populate fields on the form. • We can give the user real-time validation of their eligibility. For example, we could integrate with a credit scoring service to provide an indication about whether their loan is likely to be approved or not.	Data entry optimization is usually done using one of two techniques: • Client-side integration. This occurs directly between the form (executing in the client's browser), and the information system. The LinkedIn pre-population widget is this type - no server-side installation or configuration is required. • Dynamic Data. In this case, the form will invoke Transaction Manager, which will in turn invoke the actual service. Several Composer widgets are designed to work with Dynamic Data Services. Transaction Manager can provide added value, such as data and protocol transformation and security.
Execution	Execution is the process of adding additional value to the transaction after the data entry step. It can consist of a number of added-value steps, some of which require may integration with back-office or third party systems.	Some examples of added value integrations include: • Adding electronic signatures using third-party eSignature solutions. • Processing payments using third-party payment gateways. • Retrieving credit scores or identity verification results using third-party services. • Using Enterprise Document Management systems to generate documents or welcome kits to provide to the customer. • Providing a review-and-approval step prior to delivery.
Save and Resume	Saving a form and returning to it to resume editing is an important capability. This is shown on the diagram for completeness, but is in fact implemented purely inside of Transaction Manager. There are no user-modifiable extension points.	
Delivery	Delivery is the process of delivering the results of the transaction, in both human-readable and machine-readable format, to the organization. Of all the different points of integration, delivery is the most important. The reason is obvious - unless we deliver the data that was collected (in some way) to the organization, there is no value achieved for the organization. The range of different delivery options is highly varied. It can be as simple as an email containing a PDF as an attachment which is	

	processed manually, or as complicated as direct integration of different parts of the data into a number of different back-end business systems. Delivery can even be performed for transactions that haven't been completed by the end user, as this abandonment information can itself be useful.	

Authenticated and Unauthenticated transactions

Avoka Transact supports the use of both authenticated or unauthenticated transactions.

Authentication is usually the process of identifying a user using a user-name and password, or sometimes more sophisticated methods.

An unauthenticated user is a user who hasn't logged into Transaction Manager, and doesn't have an identity in Transaction Manager. This does not necessarily mean that they are unknown, and usually some information will be known about them, often including their email address.

Similar capabilities are available for authenticated and unauthenticated users, but are implemented in slightly different ways:

Capability	Authenticated	Unauthenticated (Anonymous)
Pre-population	A user's identity can be used to pre-fill forms with information that is already known about that particular person. This assumes that the user's identity already exists in an existing system.	Pre-population can still be achieved, but is implemented without the user actually logging in. For example, pre-fill data can be passed to the transaction using URL parameters.
Save&Resume	An authenticated user can save a partially completed transaction. By logging in again later, the user can view their partially completed transactions (known as drafts), and resume editing. The user can also log in from a different device.	Save and resume is implemented via a simple reference code and challenge question (similar to an airline booking code).
Transaction History	When an authenticated user logs into Transaction Manager, they can see the history of their submissions. (Note that they will usually only be able to see a summary of the transaction, not the content, because generally data retention policies will result in the details being purged.)	Transaction history can be provided to in a number of different ways, such as a confirmation email. Historical information is often maintained in other systems that maintain a more complete history of the customer's interactions with the organization, such as CRM systems.
Tasks	A task can be assigned to a user who is authenticated in Transaction Manager, and appears in their task list.	A task can be assigned to an unauthenticated user via email or any other system that can display a URL. The user will simply click on the link in order to perform the task.

A question that is often asked is whether to use authenticated or unauthenticated access. Transaction Manager supports both types extremely well. However, it is important to note the following regarding authenticated transactions.

1. Many organizations already have the concept of identity and usernames/passwords. Usually the organization will not want to introduce a new and different username/password, but rather re-use the existing mechanism.
2. Users are often reluctant to sign up with a username and password just in order to be able to save and resume a transaction later. (We have data that supports this assumption.) This can increase transaction abandonment, because users simply don't want to register. It's usually wise to allow users to complete a transaction without requiring them to register.
3. Identity can also be complicated to implement, especially the identity of your customers outside the firewall. Transaction Manager does support both common signon using LDAP, and Single Sign On (using SAML), but these can add complexity to the implementation.

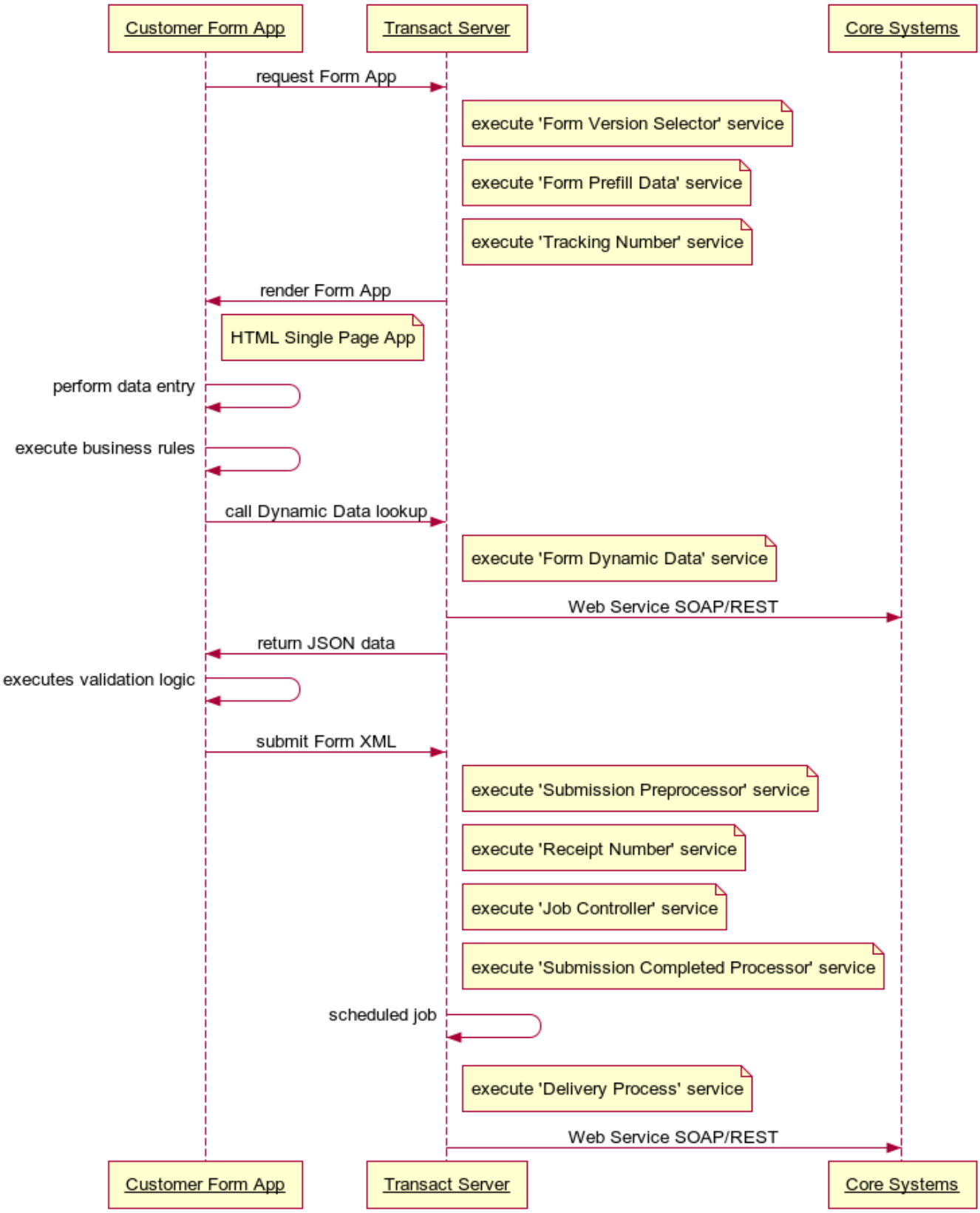
Because of the above, we generally recommend the following:

- External customers be treated as "anonymous" users, without trying to establish or confirm their identity online. This may have a minor impact on their ability to pre-fill their forms (although there are ways to achieve this), but in general, treating them as anonymous dramatically simplifies the integration requirements, the security implications, and the time to market. As outlined above, most features in Transact such as save and resume, task assignment and pre-fill have been designed to work with either anonymous or authenticated users.
- Internal users (such as help desk and call center staff) be treated as authenticated users. This is usually much simpler to implement, since all internal users are already authenticated on the network, and both LDAP and SSO are relatively simple to implement.

In some cases authenticated end users may be required for business reasons, and Transact does support this extremely well - but bear in mind that it may affect your time and cost of implementation.

Extension Points

Integration is performed in various areas of the transaction life-cycle though the use of defined extension points. Developers can inject code or other invocations at these defined extension points. The main points are shown in the interaction or sequence diagram below, although some extension points have been omitted for simplicity.



The Five Integration Areas in Detail

Delivery

Delivery integration is extremely important to every organization. Unless you deliver the data collected to the organization in some way, you are ultimately not gaining any value. This area is listed first, because it is almost always implemented, whereas many of the other integration areas are optional.

There are generally three types of artifacts that are delivered:

- **Receipt.** A human-readable version of the form, usually in the form of a PDF document. This is known as a receipt, and can be generated in several different styles.
- **Data.** A computer-readable version of the data, in the form of an XML document. This is suitable for integrating into other systems.
- **Attachments.** Any supporting documents, known as attachments. These may be photographs, scanned documents, or electronic documents of various types.

Avoka Transact has a large number of different ways in which deliveries can be performed, ranging from simple to sophisticated.

Delivery Type	Description	Examples
Email	The PDF, XML and attachments are attached to an email, which is then emailed to an individual or group email box for processing. The person/people monitoring this email account process the PDF, XML and/or attachments manually. This approach is usually the simplest option for low volume and/or low value transactions, but it can also be used very successfully as a short term solution while more sophisticated integrations are being built.	There is really not very much to do, other than making sure Transact has access to a mail server, and identifying the correct email recipients.
Fax Gateway	If your organization already has a fax gateway, then the inbound faxes will generally be delivered as attachments to an email message, or as files in a folder. Transact can easily be configured to do the same.	Avoka have previously integrated with the RightFax fax gateway.
Web Services (SOAP)	Transact has a SOAP API that lets external systems retrieve submissions (including their PDF, XML and attachments). This API can be called by any client that is SOAP-compliant. This approach has the advantage that no firewall holes are required since this is a pull invocation.	Several of Avoka's customers have used the SOAP API in order to retrieve submissions.
TIA	Avoka Transact includes a pre-built (but extensible) application known as the Transact Integration Agent. This Java-based application uses the SOAP API to retrieve submissions from Transaction Manager, and deliver these into external systems. A number of standard integration end-points are provided, such as folder drop, or customers can customize TIA to perform any type of integration that is possible from Java.	The Australian Dept of Industry uses TIA to deliver submissions into a number of different agencies in a multi-tenanted environment. This provides a level of separation between the Dept of Industry who manage and administer the system, and the agencies who only receive submissions destined for them.
ESB / Mulesoft	MuleSoft is an Enterprise Service Bus. Avoka has developed a Mule connector which uses the SOAP API to retrieve submissions from Transaction Manager. Once a submission has been retrieved, it can be easily integrated with any of the systems that Mulesoft supports, which are listed here: https://www.mulesoft.org/connectors Mulesoft also has a sophisticated data mapping tool for mapping data from one form to another.	
REST API	Transact has a REST API that that lets external systems retrieve submissions (including their PDF, XML and attachments). This is similar to the SOAP API, but is a more modern protocol, easier to use, and supports high availability architectures.	
TMRDC	The Transaction Manager REST Deliver Client is a very simple and self-contained application built in the Go language. It uses the REST API to retrieve submissions from a Transaction Manager server. The default implementation is to store these submissions in a folder/directory. However, source code is provided, and can be customized to perform other types of integrations.	
Groovy	Transaction Manager allows delivery integrations (and other types of integrations) using the Groovy scripting language. Groovy is a simple yet powerful scripting language that has been built into Transaction Manager. Groovy gives you access to many of the internal Java API's provided by Transaction Manager, and also has capabilities for calling third party services and systems, including REST and Web Service API's, so it is a powerful tool for building sophisticated integrations. Groovy is often used to "glue" Transaction Manager to another system, rather than being used as a programming language. A full set of the available Transaction Manager API's can be provided upon request if desired.	Examples!!! They abound.
Java	For even more sophisticated integrations, Transaction Manager allows Java modules to be developed and uploaded into Transaction Manager, where the methods they contain can be invoked at defined extension points. This provides access to the full power of Java, plus all of the Transaction Manager Java API's.	

In addition to these general purpose delivery mechanisms, several specific integrations have been developed for particular end-points.

These have been integrated into Transaction Manager as pre-built integrations, and they include:

Delivery End-point	Description

Microsoft SharePoint	This integration allows submissions to be delivered directly into a Sharepoint instance.
SalesForce	This integration allows for: 1. OAuth 2 authentication against SalesForce identities. 2. Delivery of submission information directly into SalesForce for marshalling into destination objects. 3. API level form assignment to known and anonymous users directly from SalesForce, encapsulating APIs on Transaction Manager. 4. UI objects to present a categorized list of available forms directly inside SalesForce, for either opening immediately or pushing to SalesForce contacts. These may be automatically pre-filled with data from SalesForce objects. 5. UI objects to review the active and completed forms for a particular object within SalesForce.
Marketo	Creates or updates the visitor's Marketo LeadRecord using the submitted form data.
Adobe LiveCycle	Transaction Manager can deliver submissions directly into Adobe LiveCycle Process Management. Process Management is a BPM system that allows back-office business processes to be easily developed.

Finally, Avoka staff and our customers have used some of the above capabilities to integrate with a large number of off-the-shelf and custom systems. Some of these include:

- PEGA (BPM system)
- SAP (ERP system)
- Microsoft Dynamics (CRM system)
- FTP/SFTP (file transfer protocols)
- Shift (a dealer management system)
- Carflow (a dealer management system)
- Huddle (a document management system)
- K2 (a BPM system)
- IBM Lombardi (a BPM system)
- Telco systems (several bespoke systems at a large Telco)

The Delivery Service extension point with Transaction Manager provides a number of built-in added value services, including:

- **Retry capabilities.** If a delivery fails, it can be automatically or manually retried. This can be helpful for cases when a delivery end-point is temporarily unavailable.
- **Delivery checkpoints.** If a delivery consists of multiple steps (for example, send an email, then update a database, and then store a file in a directory), then each part of this delivery can be set up as an individual checkpoint. If one of the steps fails, but a previous step has completed successfully, then when the delivery is retried, it will start on the first incomplete checkpoint, rather than from the beginning. This would ensure that, for example, only one email ever gets sent.
- **Alerts.** The delivery subsystem provides administrative alerts to notify system administrators when a delivery has failed so that the underlying cause can be corrected.

Discovering Forms

Every unique form in Transaction Manager has an associated URL link that you can place on your site. Any user who clicks on the links will launch the form. You can also use the form URL in any other way such as in an email. You can also link to the form from any type of "form finder" technology, or a wizard that assists a customer to find the appropriate forms. You can also embed links to forms in outbound communications such as email or social networks.

Another way for customers to complete a form is to send a customer a link to a form that has already been partially completed on their behalf. Transaction Manager includes a Task API, that allows you to create a partially completed form, generate a custom URL, and then pass that URL to the end user. This can be used to "push" forms from CRM or other systems.

An older approach is to embed the Transact form directly inside the organization's web site. This approach is rarely used, mostly because of the advent of smart-phones, which lack the screen real estate to display both the existing site and the form in the same screen. There are also other issues with this approach, including the importance of removing distractions and navigation links from the task-oriented process of completing the transaction, as well as the difficulty of effectively designing responsive user interfaces in two separate web pages that effectively work together.

Pre-Population - Accelerating Data Entry

Transaction Manager provides many different techniques for pre-filling forms. One of the simplest ways to implement this is to pass pre-population data as URL parameters in the URL that links to the form. These parameters can be customized on a per user basis in your existing web site if required. Another way is to retrieve pre-population data from data stored in cookies.

Alternately, Transaction Manager has many extension points where calls can be made to external systems, and the results of these calls can be injected into the form's data before it is presented to the user. The sequence diagram above highlights the key extension points. The extension points can be implemented as

Java or Groovy calls to external systems. These extension points also have access to many of the internal Transaction Manager API's.

Transaction Manager implements a form receipt-number generation service, to ensure that each form has a unique code. This service can be configured to auto-generate different styles of receipt number, or it can be extended to call external systems to generate the receipt number. This allows the form receipt number to match a reference number in an existing system.

Transaction Manager implements several different techniques for passing identity from an existing system to Transaction Manager, including SSO/SAML, LDAP integration, and others. If the user's identity has been established, then calls can be made to back-end system using this identity to retrieve user-specific data, which can then be injected into the form before being presented to the user.

Streamlining Data Entry

Once the user is actually within the form, we want to make their experience as streamlined as possible. One way we can do this is by designing and creating engaging, simple to use, guided experiences using Transact Composer.

We can also streamline their experience by providing contextual data within the form that:

- accelerates their data entry;
- validates their data entry; or
- provides information to them that helps them to make decisions.

Usually this type of integration is provided by custom user interface widgets within the form itself. There are generally two different mechanisms that these widgets use to communicate to external systems.

Mechanism	Explanation
Direct	The widget communicates directly to the third party service, usually (but not always) using JavaScript, REST and JSON. This is most useful for cloud-hosted third-party services that don't require usage keys, identity verification, or license/billing, and are not subject to security concerns.
Dynamic Data Service	A Composer widget communicates with a standardized dynamic data service within Transaction Manager. The dynamic data service protocol is predefined, and uses JSON data as the payload. The dynamic data service then communicates to the desired back-end service to verify or retrieve data, perform a calculation, or verify data. The results are then passed back to the widget. Dynamic data services are useful for several scenarios: • Licensing. Some external services require the user to specify a usage key of some sort. • Billing reconciliation. • Security (secure channel to backend services) • Data transformation or consolidation, protocol translation. Composer includes several pre-built widgets that support Dynamic Data calls, and custom widgets can also be created.
Hybrid	Occasionally it is useful to build widgets that use a combination of direct access and dynamic data services. For example, an initial call may use Dynamic Data to establish credentials with a third-party, but then use direct access to the third party service for greatest efficiency.

Some examples of pre-built services that communicate with third-party services include:

- Experian, which provides postal address lookup services.
- GreenID which provides identify verification services.
- Transaction Manager Reference Data, which allows data to be stored in Transaction Manager, and displayed in a variety of different ways.

Many Transact customers have used the generic Dynamic Data-aware widgets to expose their back-end web services, REST services, databases and bespoke systems in a variety of ways within the form.

Execution

Execution is largely about adding additional value to the overall transaction, after the basic data completion step has been completed. Some of these integrations are part of the process of completing the transaction, and are visible steps to the end users, whereas other types of integrations occur behind-the-scenes.

The following examples list some of the types of integrations that are possible at execution time with Transact.

Integration	Description

Electronic Signatures	Transact can integrate with Electronic Signature providers in order to provide an improved level of signature compliance. (Note that Transact provides several types of electronic signature capability directly in our system, without requiring third-party services.) Transact has already been integrated with DocuSign and Silanis, but the same principles can be extended to similar electronic signature providers. Electronic signatures are usually implemented in one of two ways: 1. **Immediate signature.** A receipt (or document of record) is created, and submitted to the third-party signature provider, along with signatory meta-data. The signature provider processes this request, prepares the document for signing, and returns the signatory id to Transact. Transact then redirects the user to the signature provider web site, where the document will be presented for signature. The signature providers may also request additional identification information. Once the user has signed the document, the signature provider will redirect back to the Transact web site, so that the transaction can be completed, delivered, etc. 2. **Signature by email.** A receipt is submitted to the signature provider, along with signatory meta-data.The signature provider will prepare the document for signing, and then send an email to the signer for processing. Transaction manager will notify the applicant to check their email address. The user will click on the link in the email, and complete the signature process in the signature provider's web site. Transact will be notified of the signature completion, and will complete the transaction, including delivery etc. The immediate signature approach is a more seamless experience for the end user. However, the signature by email approach can have a higher level of identity for the signer, because it implicitly validates their email address. It is also generally the only way in which a document can be signed by multiple signers - this is usually supported by signature providers in either parallel or sequential flows.
Payment processing	Many transactions require an associated payment. The Avoka form will calculate the fee, and also specify what payment methods are available. When the form has been submitted, Transaction Manager will redirect the user to a payment provider's website, along with meta-data about the payment, such as the amount. This site is known as a Payment Gateway. The Payment Gateway will process the payment, and then redirect back to Transact, passing payment meta-data. Transact will inform the user of the outcome, as well as store the payment-meta-data to allow for reconciliation reports to be run. Transact never stores or processes credit card or banking information directly. This is a specialist service that requires specific levels of security and privacy compliance, and is always performed by a specialist payment providers. Transact has built-in integrations with several providers, and other providers can be implemented as required.
Additional Verification	Additional Verification of the user's information for purposes such as credit scores or identity verification can be implemented within Transact itself. These can also be implemented post-delivery in the organization's back-end processes, but performing them within Transact provides the ability for the end-user to be informed of the result while they are still within the transaction, and therefore have an opportunity to retry the application and correct the information.
Review and Approval	Transact includes a simple but powerful review and approval capability known as Collaboration Jobs. Transactions that have been completed by end users can be progressed through several review/approval steps, by different individuals or groups. Steps can also be designed to retrieve information, perform processing, or make decisions. Review and approval is often implemented in larger and more sophisticated BPM or Workflow systems, but the Review and Approval capability in Transact provides a number of benefits: • There is no need to purchase or maintain a BPM or Workflow system. • Review and Approval is very simple to configure, and very easy to use for the reviewers.
Document Generation	Transact produces a receipt automatically, in PDF format, providing a document of record for the customer and for the organization. Transact can produce a document of record using: • Composer-generated receipt templates (automatically) • Hand-created Adobe LIveCycle Designer receipt • Hand-created Adobe Acroform receipt (also known as pixel-perfect receipt). However, in some cases, the documents that need to be created are more sophisticated or different to what can be achieved in Transact natively. There are a number of systems known as Customer Communication Management systems which are designed to produce customer communication such as offers, receipts, contracts and information packs. These systems can be used by Transact to generate the document of record.

The Economics of Integration

It is very important to remember the consider why we do integration - we don't do integration just for the sake of it, we do integration order to provide specific value to the organization. Integration can often be the most time-consuming and costly part of any implementation. It's very easy to fall into the trap of doing integration because it's possible or seems obvious, but it is wise to always step back and consider what value integration is providing to the overall business objectives relative the cost of implementation and future maintenance.

Why do we do integration?

Usually we do integration in order to achieve operational efficiencies. Integration usually saves humans from having to perform manual tasks, which in turn reduces costs. These efficiencies may be realized by the end-user (eg save typing and frustration through pre-population) or by the organization itself (save money and reduce errors by eliminating

Assess the Value of Integration

A survey performed by the Forrester group in 2015 asked "What initiatives are likely to be your organization's top business priorities over the next 12 months?"

The top three responses were:

- Grow revenues (77%)
- Improve the experience of our customers (74%)
- Reduce costs (61%)

It is interesting that improving user experience is regarded as significantly more important than reducing costs. And that growing revenues is the most important criterion in the entire survey.

- Avoka Transact is all about improving customer experience, and growing revenues (although it also leads to improving efficiency).
- Integration is all about improving operational efficiencies, and reducing costs.

It is important to keep in mind that in many cases, the primary goal of implementing Transact is improving user experience, and growing revenue. You should therefore consider very carefully the value of doing integration compared to the cost. It may be that you can achieve a large proportion of the value of Transact without doing any integration.

We're not saying that you shouldn't do integration, just that you should be aware of the costs and the value.

Consider Deferring Integration

Another factor to consider is time to market. Transact can be implemented very quickly, and realize important benefits such as improved user experience and improved conversion rates. However, integration, because it touches back-office systems, needs to be very carefully developed and tested, and this can often take a significant amount of time.

One common approach is to implement Transact with little or simple integration, and realize the primary benefits as early as possible. More sophisticated integration can then be performed at a slower and more careful pace - and because Transact delivers the data in XML format, it's easy to add integration later. Integration can be prioritized for the most important transactions first, with lower value or lower volume transactions deferred till later.

Some of our most successful implementations had started with minimal integration at the beginning – but over time the integration has been phased in separately.

Consider Leveraging Existing Paper/image-based Processes

Ideally the goal is to completely replace paper/images with all-digital transactions. However, in reality we rarely achieve 100% success - there are almost always going to be some customers who insist on hand-writing on paper. This means that you almost always needs to maintain your existing back-office processes which currently handle paper or images of paper.

Since you're going to need to maintain these back-office processes anyway, one useful approach is to augment or plug into these back-office processes using digital artefacts, rather than trying to create a completely all-digital back-office process. In other words, deliver a digital document (generally a PDF) into your existing paper- or image-based systems. If you do this, integration is usually very simple and quick.

Hidden Costs of Process Re-engineering

Process re-engineering is costly. This is not just about technology, but about people, skills, training, management, and related costs. Overall change management is often overlooked, and can be hugely costly. You don’t want to introduce a new digital channel, and at the same time, completely re-engineer your back-office processes. Instead, one option is to simply plug digital artefacts into existing back-office systems. You get the benefits of a new digital customer-experience, but you don’t have to retrain anyone, build new processes and procedures and manuals, hire new staff with different skills, or put new managers in place. You have the opportunity to re-engineer and optimize your existing back office processes at a later time.

Our strong recommendation is not to combine process re-engineering with a Transact (user-experience improvement) project, otherwise there is a possibility that both projects will fail. Process re-engineering and change management are hard and expensive, and shouldn't be embarked upon lightly.

Related articles

Error rendering macro 'contentbylabel'

parameters should not be empty

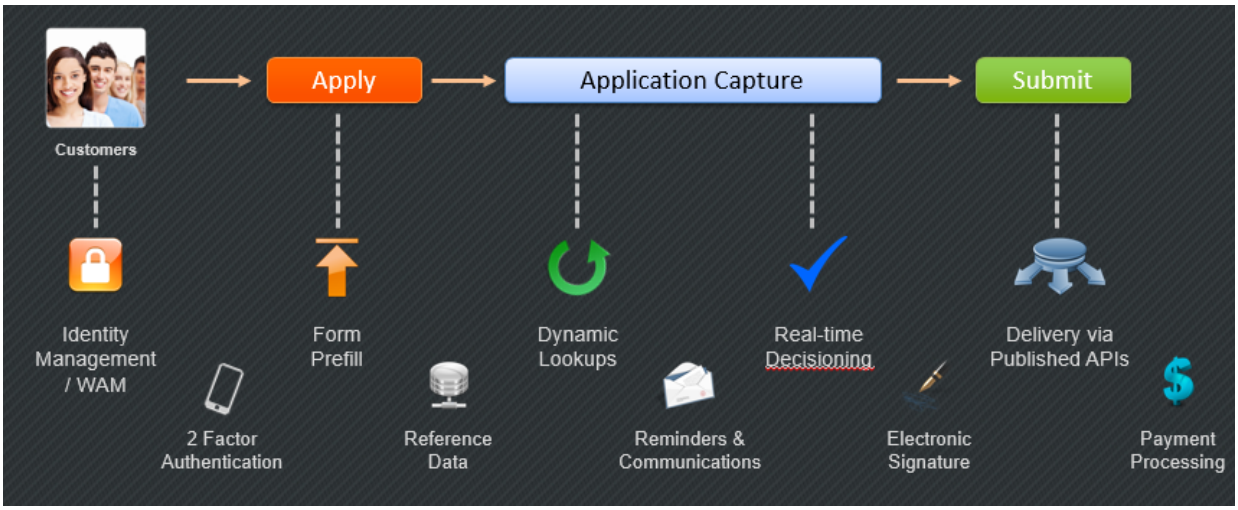
Integration Capabilities in Transact

Overview

Avoka Transact provides extensive capabilities for extending the platform, and for integrating with third-party or organization systems. This brief article outlines some of the capabilities and techniques.

The Form Execution Life-cycle and Extension Points

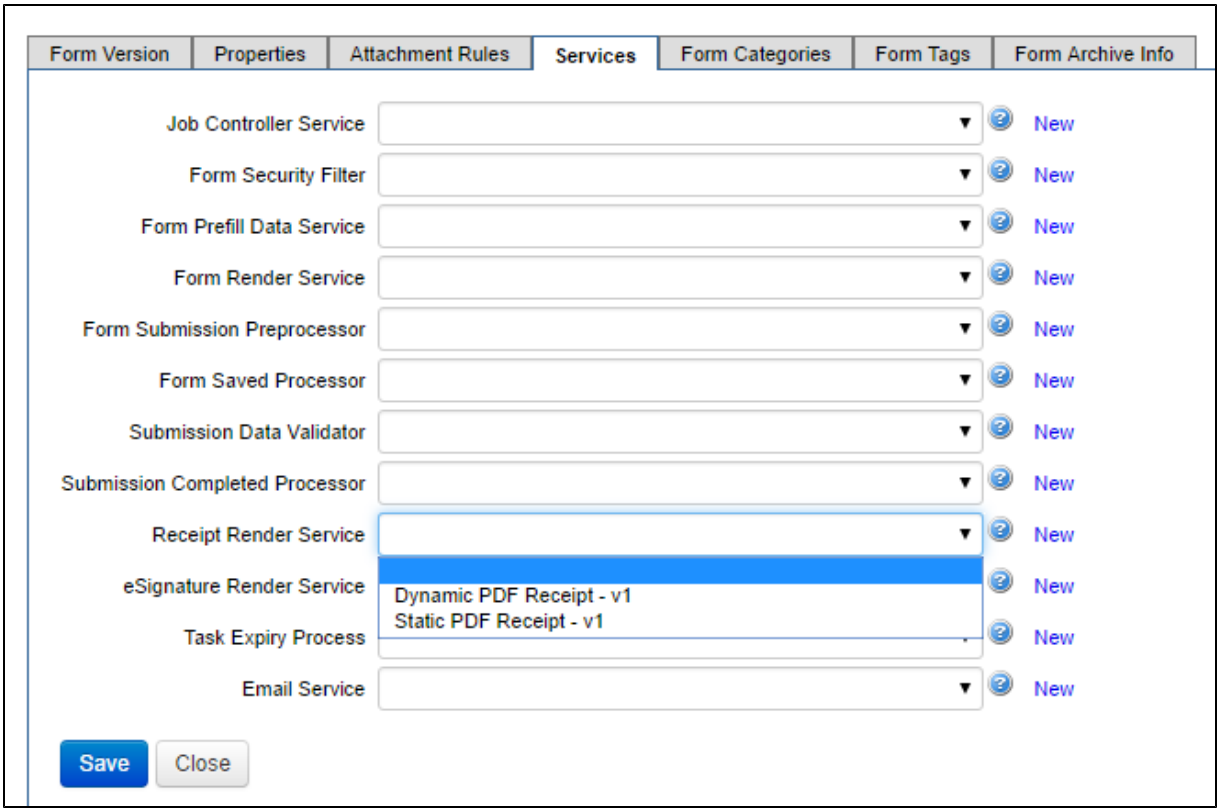
There are many different points in the life-cycle of a user interacting with a form, between the time that the user first clicks on a link to open the form, and the time that the transaction is complete. We won't go into all the details, since there are over 20 different life-cycle events, but some of the key events are shown in the diagram below.



Some of these points are handled by built-in capabilities within Transact, but others require integration with third-party or organization systems.

Transact Manager provides extension points at many of these life-cycle events to allow integration between Transact Manager and these other systems.

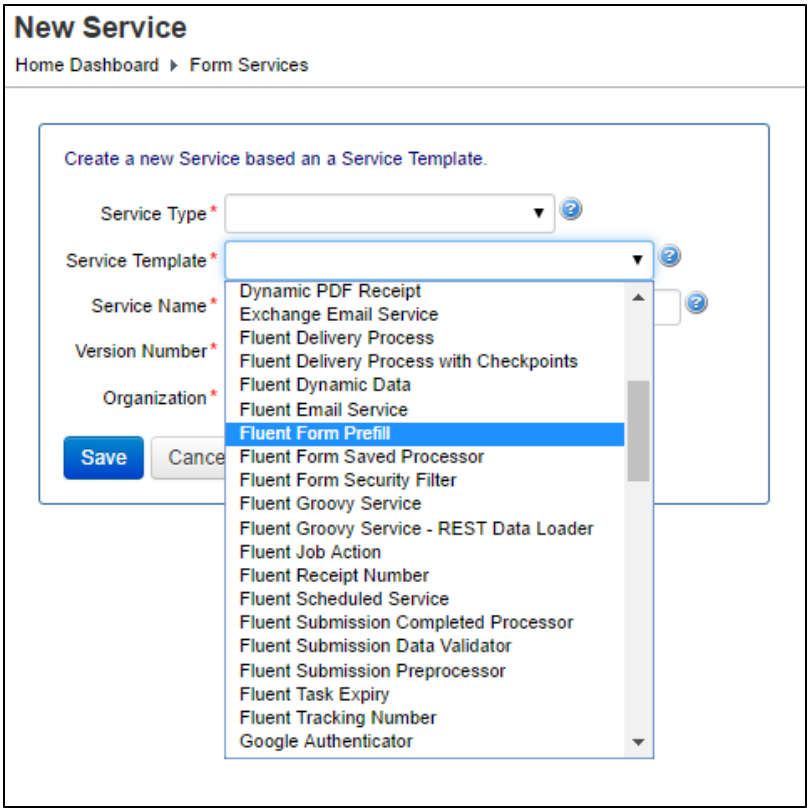
The screenshot below shows some of these extension points:



Each of the drop-down lists specifies a service that may be invoked at that particular point in the transaction life-cycle. You may select an existing extension point (as is shown in the Receipt Render Service in the screenshot), or you may create a new extension point by clicking the "New" button.

In order to build a new extension point, you author the extension point in a language called Groovy. Groovy is a simple yet powerful scripting environment that most developers find intuitive and productive.

When you create a new service, you are not simply presented with an empty screen and begin typing. Over the years, we have collected a series of templates from real projects that can be used as starting points for development of different types of services. A subset of the available templates is shown in the screenshot below:



The template will create a starter script which allows a programmer to simply "fill in the blanks" for their particular service.

Calling External Services and Transact APIs

Once you start building a service, you will often need to invoke external services to actually perform the work. These services will usually be exposed as either SOAP or REST web services. These services may already exist, or they may be built as part of the Transact project. They can be built in a number of different ways, including hand-built using coding languages, or the use of specialized integration system or an Enterprise Service Bus.

Libraries for calling these external services are built into the Transact Platform. An extract from the Transact documentation shows this example of calling an external REST service:

Examples

The example below performs REST request and returns a JSON object

```
import com.avoka.tm.http.*
import com.avoka.tm.util.*

Map params = [:]
params.id = request.getParameter('customerId')
params.account = request.getParameter('accountId')

String username = svcDef.paramsMap.username
String password = svcDef.paramsMap.password

// execute GET request and return a HttpResponse object
HttpResponse response = new GetRequest('https://service.mycorp.com/secure/rest/accounts/')
      .setParams(params)
      .setBasicAuth(username, password)
      .execute()

if (response.isStatusOK()) {
    // get Path object from the response
    Path path = response.getPathContent()
    ...

} else if (response.isStatusNotFound()) {
    // object not found
    ...

} else {
    throw new RuntimeException(response.statusLine)
}
```

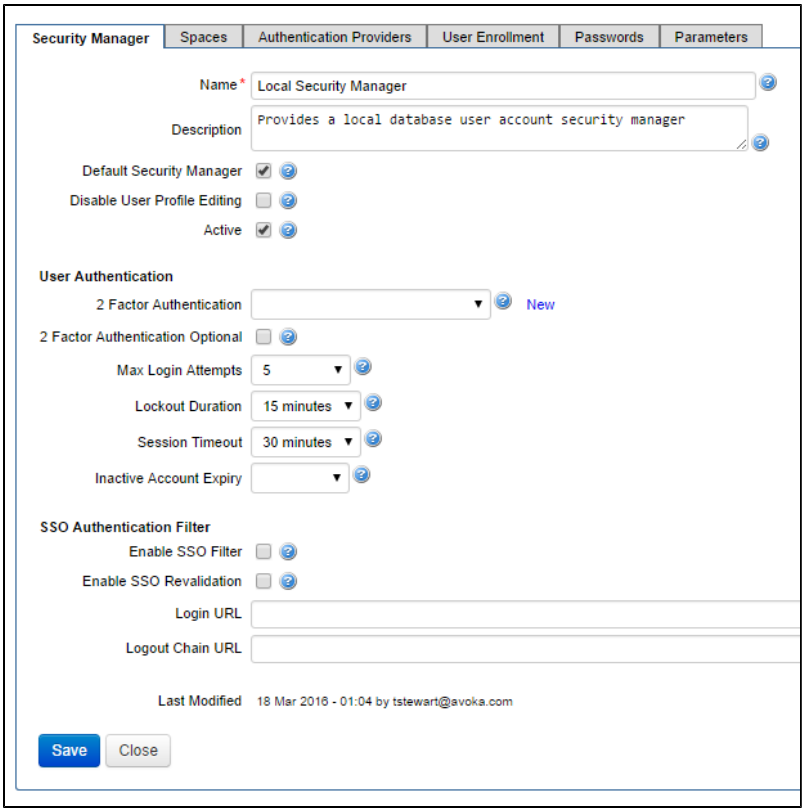
This illustrates how easy it is to invoke these external services. Of course, more sophisticated services can be developed.

It is often also necessary to invoke Transact Manager itself in order to get more information from the system, or to update the transaction. This is also available through a set of comprehensive Transact API's.

Web Site Integration and Single Signon

It is almost always required to integrate a form into an existing web site. Usually this is achieve through a simple link or button. In the past, more complex integrations were performed, such as embedding forms within existing html pages, although in modern sites this is rare, largely due to the needs of smaller screens and mobile devices.

It is also sometimes necessary to provide single-sign-on integration with existing systems. This is provided by an extensible single-sign-on mechanism that is part of the security management sub-system within Transact. SAML, oAuth and two-factor authentication are supported. A screenshot is shown below:



Form Level Integration

Sometimes it is useful to perform integration directly between a form and another service. This can be performed in two ways:

1. Directly between the form and the service. This can be implemented directly using JavaScript in the usual way. Generally the integration will be packaged up into re-usable component that can be simply dropped into any form that needs it.
2. Using Transact Manager as a proxy service. This mechanism is known as Transact Dynamic Data services. The form uses a standard mechanism for passing data in and out, which both programmers and non-programmers can easily use and configure. This invokes a service on Transact Manager (written in Groovy). The Groovy service has all the power of the Groovy programming language and supplied libraries, and so arbitrary complexity can be simplified to ensure that the service exposed to the form is easy to use. There are several advantages to implementing the system this way, including ease of use, centralized configuration, integration with Transact licensing audit logs, and more. This is the technique usually used.

Strategies for integrating forms to existing websites

Unknown macro: 'redirect'

Summary

The modern trend is for launching forms from existing websites is to open a new browser tab/window. This has a few important advantages:

- Works better on mobile devices where screen real-estate is limited.
- Prevents the user from moving away from the data entry experience by clicking on links in the surrounding page.
- Is simpler from an implementation perspective

Detailed Analysis

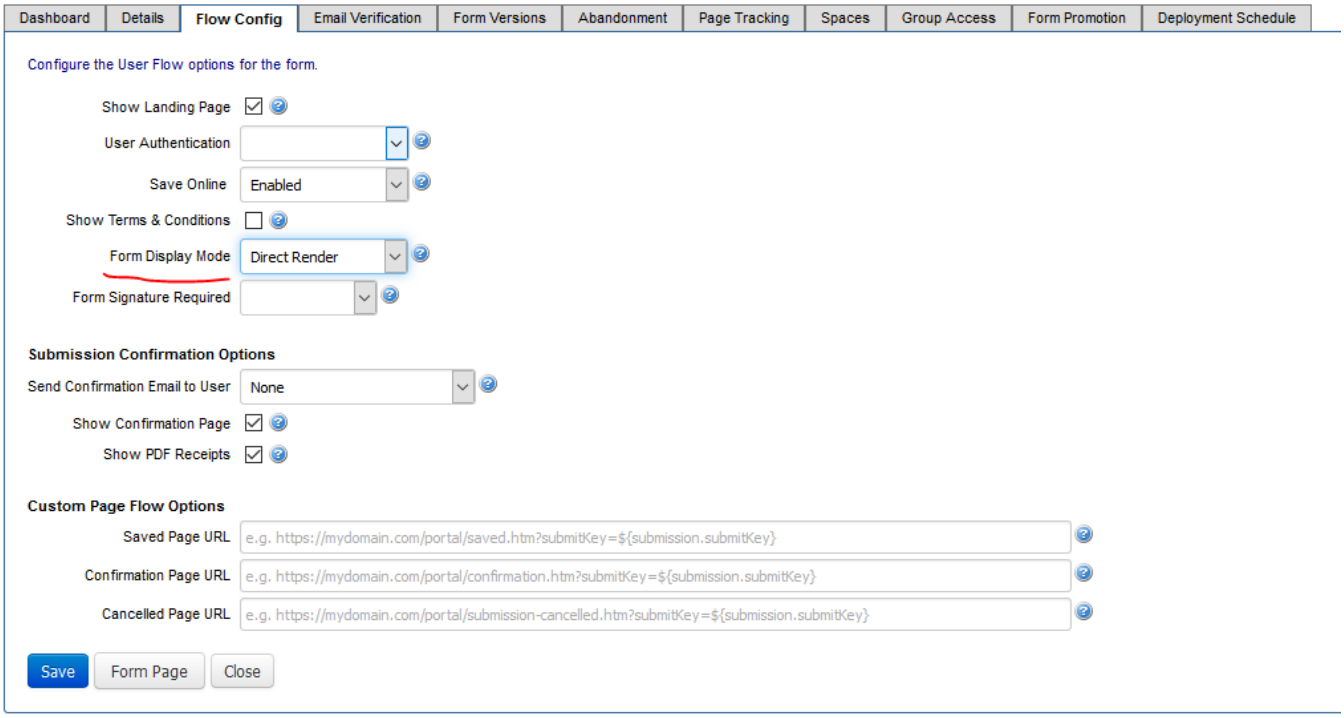
Integration Strategy	Form Display Mode	Pros	Cons
iFrame Embedding	**Embed with IFrame**	• Simple • No "leakage" of JavaScript or CSS from parent page to form page	• Pages although appear to be integrated are actually separated causing a number of issues • Refreshing of the page is problematic • Back button does not work properly • Difficulties in maintaining styling with significant effort needed on BT to maintain • Difficult to pass control between the pages • Complexities in the JavaScript as the parent control needs to be passed to Avoka • Potential responsive design issues, especially on smaller devices
<DIV> Embedding	**Embed with Div**	• Solves some of the problems associated with iFrames	• Complexity around the sharing of JavaScript used to render the contents of the form into the DIV element • Potential for code bleed/CSS clashes • Not recommended by Avoka
Launch new tab/window	**Direct Render**	• Clean code separation • Clear boundary of ownership • Control easily and intuitively passed between NextGen UI and the Avoka on-boarding pages • Control to the Avoka pages is managed at the URL level • Recommended by Avoka as their preferred approach with their other customers	• The form must be built with a simple header that includes a small amount of branding from the web site. • Visual design - need to be simplified in terms of shared header.

Configuring Form Display Mode

In Transaction Manager parlance, this is called Form Display Mode

Embedding the form (either with a Div or IFrame) displays the form inside a portal page, where the portal page typically provides the header, footer and sidebars.

Direct Render displays the form using a Servlet and does NOT include any elements outside the form itself.



Form URLs

These different methods of displaying the form mean that different URLs are used by Transact, as follows.

URL	Usage
<portal>/form.htm?formCode=<form code>	Form Page which will embed a **new** HTML or PDF form in the page using an IFRAME or DIV tag. The form is identified by the formCode parameter.
<portal>/secure/form.htm?formCode=<form code>	Same as above but under a secure path, which requires the user to be authenticated
<portal>/form.htm?submitKey=<submit key >	Form Page which will embed a **saved** HTML or PDF form in the page using an IFRAME or DIV tag. The form submission is identified by the submitKey parameter.
<portal>/form.htm?taskKey=<task key >	Form Page which will embed an **assigned** HTML or PDF **form task** in the page using an IFRAME or DIV tag. The form task is identified by the taskKey parameter.
<portal>/servlet/SmartForm.html?...	Same as above options but doing a **direct render**, where by the form is not being embedded in another HTML page

When a HTML form is being rendered on a mobile/tablet device (iPad, phone, etc.) it will automatically redirect to the /servlet/SmartForm.html path, even if the form is configured to be embedded in the form page. This is because embedding HTML forms does not always work well on mobile/tablet devices.

Related articles

- Task Assignment
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer
- How to create a Library in V4

Google Analytics Virtual Page Integration

Unknown macro: 'redirect'

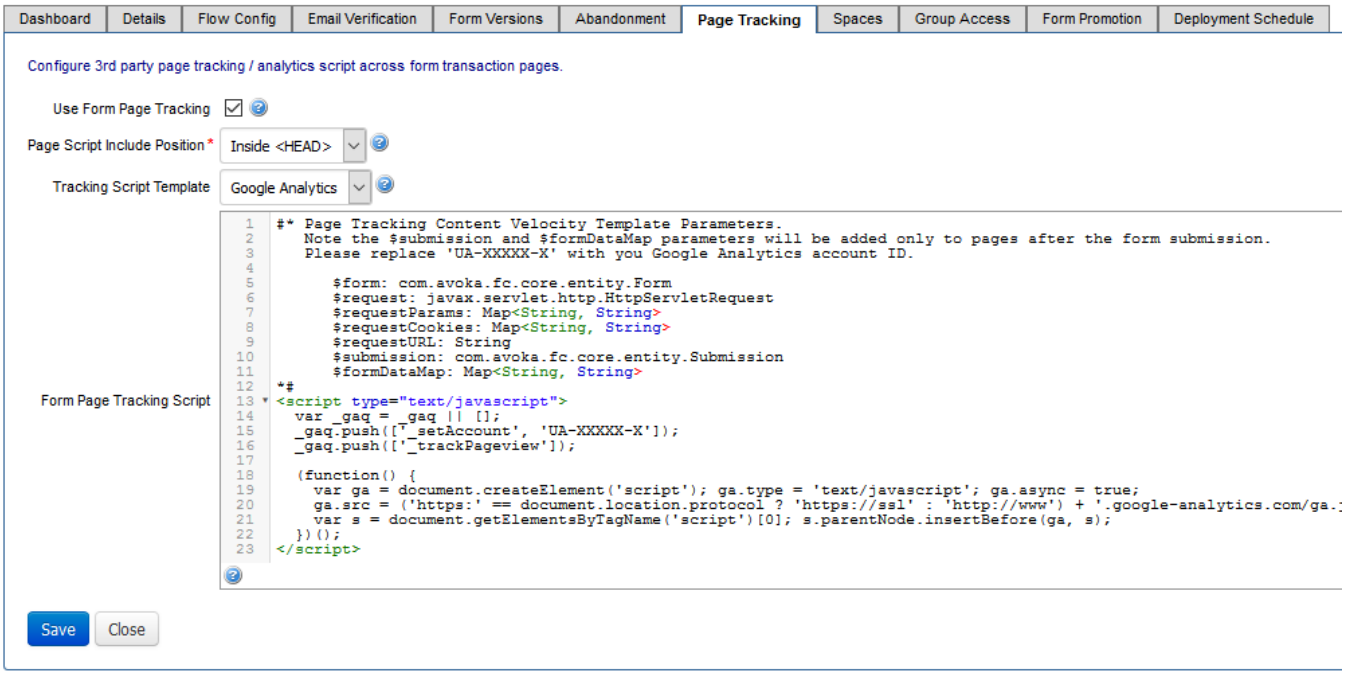
Note: this article is not a tutorial on how to use Google Analytic's UI or creating accounts.

Objective

To include page tracking via Google Analytics, each wizard page will fire a "page view" with title and url.

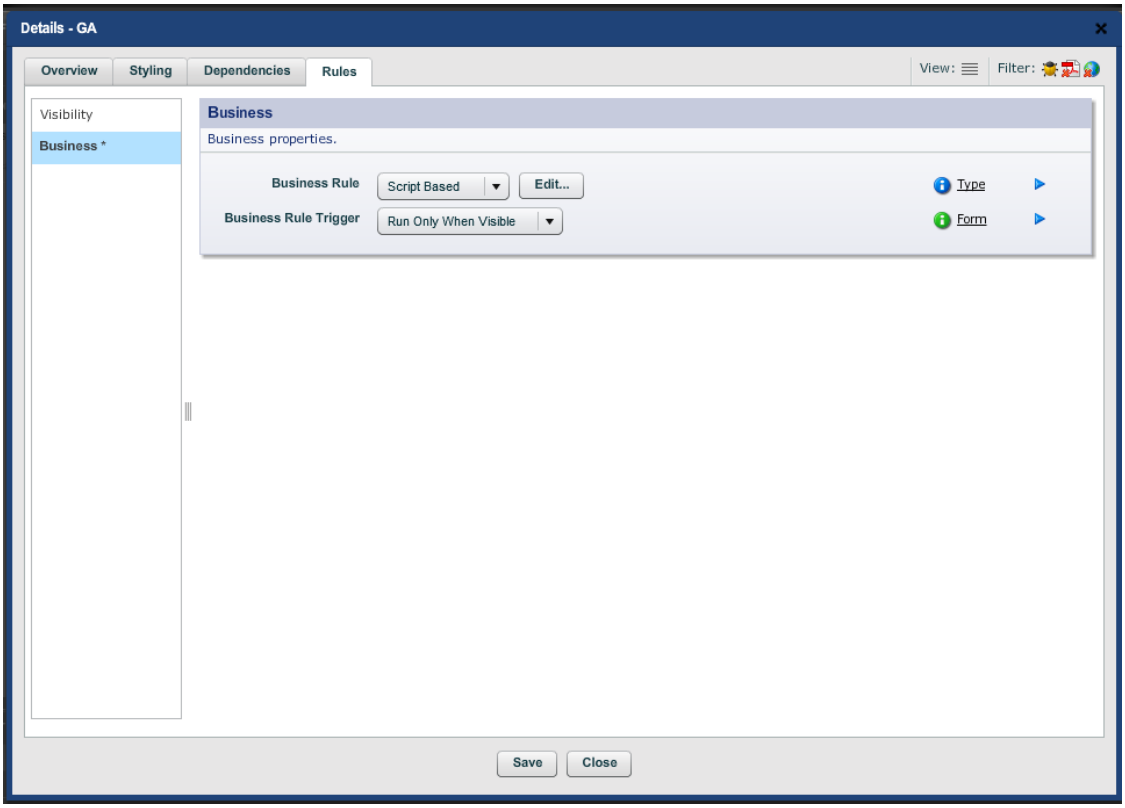
Add GA Code to Head of Page

Navigate to the "Page Tracking" tab, select the "Inside <HEAD> option" and paste in your Google Analytics code. See example below.



Create Business Rule to call the ga method on page change

Under the business rules triggering conditions check the following condition.



In the script you will need to create the virtual page views and then call the ga pageView method. See example script below.

This script uses the current page number to reference the correct object.

```
var gaObj = {
"1": {
"title": "First page",
"page": ""

},
"2": {
"title": "About you",
"page": "about_you"

},
"3": {
"title": "Your property",
"page": "your_property"

},
"4": {
"title": "Your new loan",
"page": "your_new_loan"

},
"5": {
"title": "Your details",
"page": "your_details"

},
"6": {
"title": "Your employment",
"page": "your_employment"

},
"7": {
"title": "Your finances",
"page": "your_finances"

},
"8": {
"title": "Application complete",
"page": "application_complete"
```

```
}

}

var baseUrl = "/macquarie/servlet/SmartForm/";
var currentPage = sfcInternal.av_wizard_findCurrentPage();

if(typeof ga != 'undefined') {
ga('send', 'pageview', {'page': baseUrl+gaObj[currentPage].page,'title': gaObj[currentPage].title});
}
```

Related articles

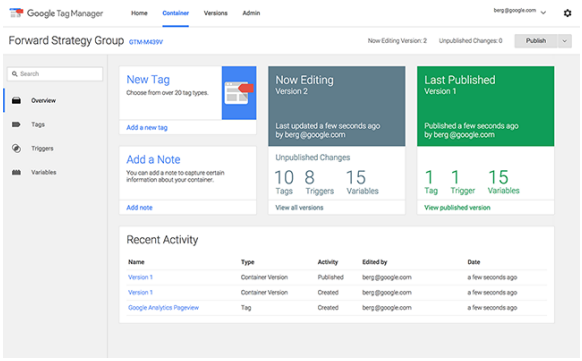
- Task Assignment
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer
- How to create a Library in V4

How to add Google Tag Manager support

Unknown macro: 'redirect'

This how-to guide assumes that you understand how GTM works, it is not a GSM training guide. For more information on GTM please visit the GTM Quick Start Guide.

"Google Tag Manager (GTM) lets you launch new tags with just a few clicks. The result? You'll enjoy faster, tighter control over your digital marketing and analytics programs."



Online web forms created with Avoka Transact Composer have a single-page architecture, whereby all of the content exists in a single rendered html page, but the user navigation flow is controlled through hide/show logic of different sections (or logical form pages). Whilst providing a better user experience, this means any standard Google Tag Manager (GTM) configurations that utilise page views to track web-site traffic will stop tracking the user movement once the form is opened – none of the in-form navigation will be tracked.

The Google Tag Manager (GTM) container discussed here provides a configurable user interface to enable web-site analytics for these forms – tracking user movement through the different sections of a form, as well as improved tracking for the separate pages that a user will be directed to after completion of the form.

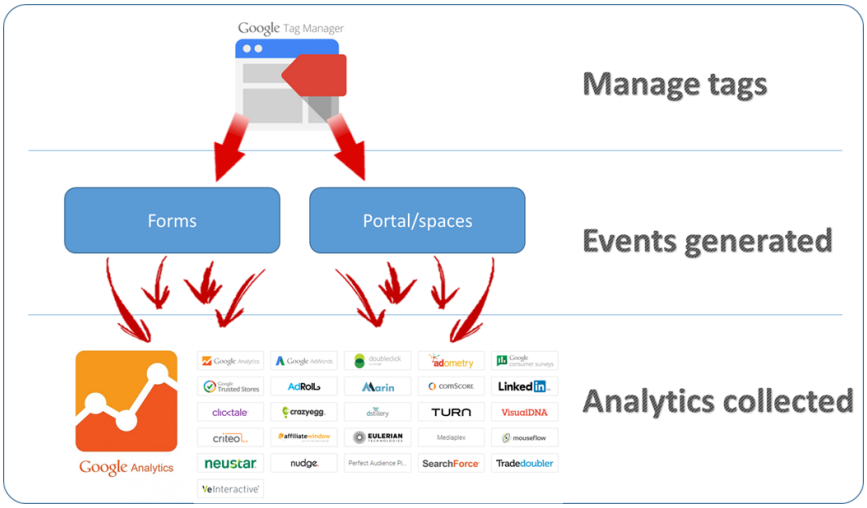
This implementation includes Tags for the Google Analytics framework, however tags could be adapted or created to support any analytics engine.

The content of this article has been provided by David Foster, I'm just the editor.

The GTM container is a fully functional entity that can be applied to any Transact Composer environment. Installation and configuration can be summarised as a series of steps:

1. Import the container into a Google Tag Manager account
2. Configure Transaction Manager to enable Analytics with this Tag Manager Account

This how-to guide assume that you already have a GTM account with the "manage" permission. You will also have to download the Avoka GTM container here.



These steps only need to be performed once for a Transaction Manager installation, but note that there is additional configuration required for each form that is to be added to the GTM implementation. This extra configuration will be covered below.

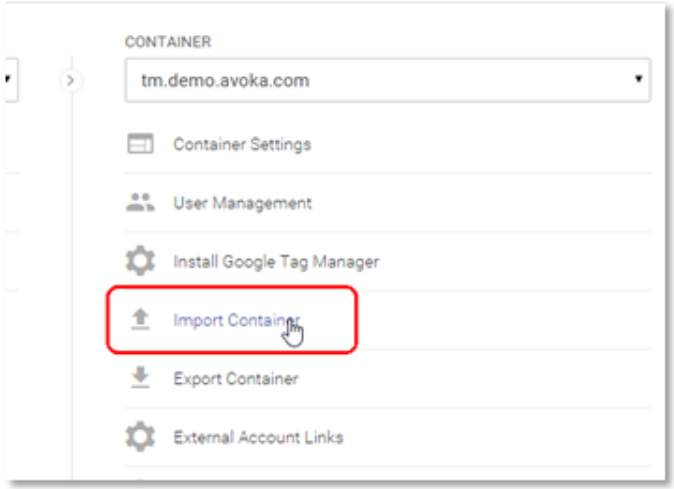
Importing the Container into Google Tag Manager

The Avoka GTM Container is designed to run in isolation – the tags pre-configured within it will only produce page views for Avoka forms and their related pages. For this reason it is safe to use either option for importing the container into GTM – either merge it into an existing container, or create a new one.

It should be noted however that if choosing to merge it into an existing container, care should be taken that pre-existing tags/triggers do not adversely affect the Avoka pages. Additionally, if new tags are being created for Avoka forms, correct use of triggers should be maintained to ensure that the tags only fire in the correct places – and not in other pages that may be connecting to the same GTM container.

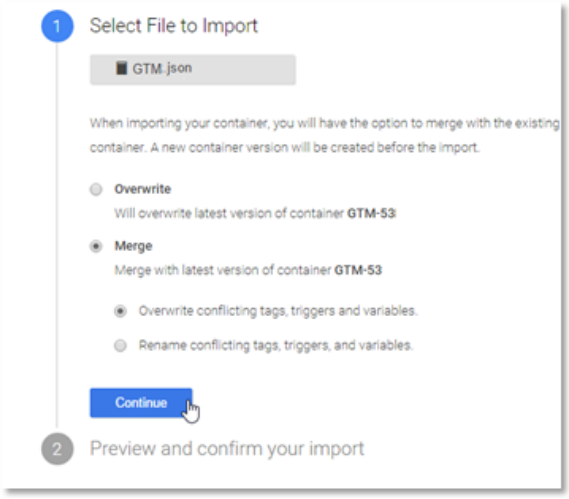
Steps for importing the container:

1. Login to Google Tag Manager and select the correct container to import into (Navigate to the Accounts tab and select the relevant container – or create a new container if required)
2. Navigate to the Admin tab and select “Import Container” from the Container menu

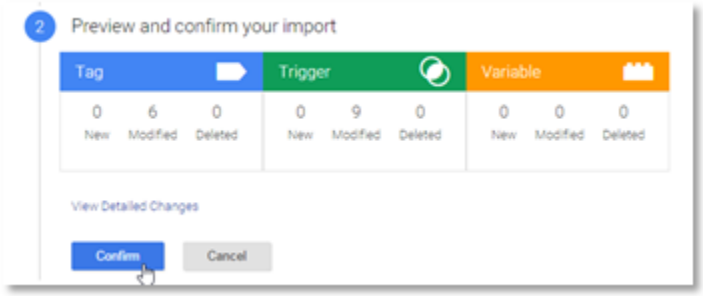


3. Under “Select File to Import” navigate to the Avoka container file on the file system

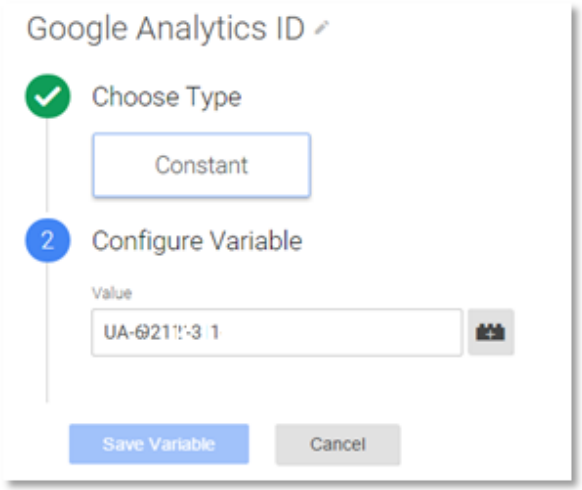
4. Select to either overwrite the existing container (for new containers only) or merge and press the Continue button



5. After viewing the Preview hit Confirm to import the container



6. Navigate to Container -> Variables and select the ‘Google Analytics Id’ variable
7. Inside this variable, enter your Google Analytics Tracking Id. This is important to ensure that page views are being sent to the correct Google Analytics account.



Configure Transaction Manager to enable Analytics

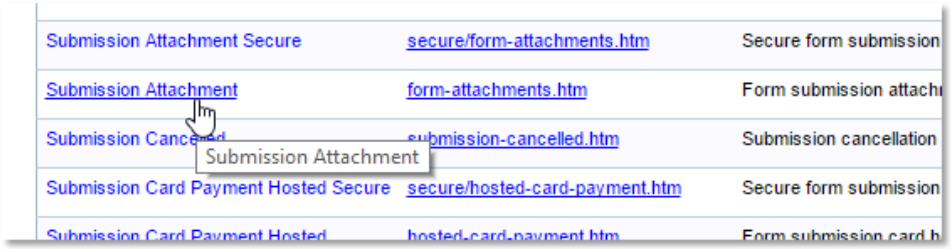
As stated, a Composer form is built using a single-page architecture, where all of the content added to the form in the Composer tool is rendered in a single html page, and hidden/shown as appropriate via browser scripting. However, the actual form content only forms part of the user-experience picture. There are also (potentially)

Landing pages, Save/Resume pages, Attachments pages and Submission Complete pages that may be presented to the user before or after they have filled in the form. These pages are also critical to deriving useful data from an analytics engine, and therefore need to be included in the GTM implementation.

Avoka TM maintains global versions of these pages that are used by all forms by default, so it becomes important to be able to identify which form the page has been opened for. For the most part this can be achieved by using the formCode URL request parameter that is required to open the form, however for some of the post-form pages (Submission Complete, Attachments, Save Confirmation) this parameter is not provided. In these cases

GTM has been setup to look for a form code within the html of the page that's been rendered, by finding a hidden element with the id of 'submission_formCode'. The pages do not automatically include this hidden element, but any TM user with the 'Portal Edit' permission can add it with a few simple steps:

1. After logging into TM, navigate to System > Form Spaces (called Portals/Modules in older versions of TM)
2. Find the space that contains the form you would like to add GTM capability to, and click on the name of the space
3. Find the page to be edited, and click on the name entry.



4. Navigate to the 'Pages' tab
5. Place the following html inside the page content. The location is not critical, but ensure it is not inside any conditional logic (with markup such as '#if ($session.emailed)':

```
<input type="hidden" name="formCode" id="submission_formCode" value="$form.formCode"/>
```

6. Save your changes (the button at the bottom of the page)

> These changes will need to be applied to the all pages that you would like to track events on.

Enabling Analytics for each Form

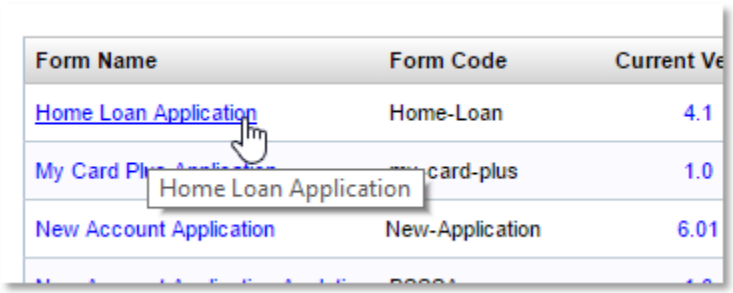
Part of the configuration exercise is to enable the creation of analytics data on a per-form basis. This allows greater control over which forms are tracked (and when) as well as tailoring the exact look of the page views that are generated. Enabling analytics for a form involves steps in both Avoka TM and GTM.

> If any of these steps are not performed, **no** analytics data will be received for the particular Avoka form – even if it's published to an environment that already has other forms sending analytics data.
>
> Having said that, it only needs to be done once for each form – thereafter, if changes to the form are made in Composer, and published to the TM environment, this setup does **not** need to be repeated.

Adding GTM container script to the form

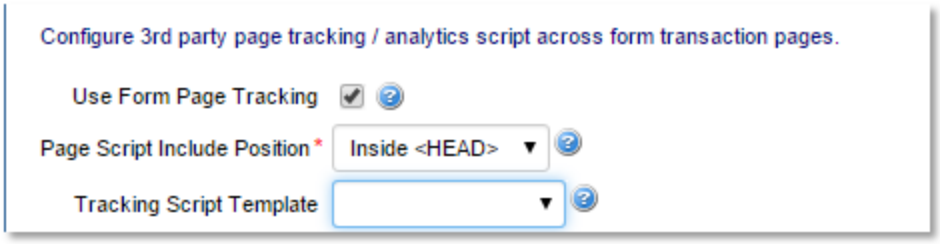
1) Navigate to Forms -> Forms

2) Click on the relevant form name to open the details



3) Navigate to the 'Page Tracking' tab

4) Select the checkbox marked 'Use Form Page Tracking' and select 'Inside <HEAD>' in the 'Page Script Include Position' dropdown



5) In the 'Form Page Tracking Script' text box, paste the GTM script that adds your GTM container to the page. This script can be obtained from GTM by navigating to Admin -> Install Google Tag Manager

6) Save your changes

Additional Forms Pages

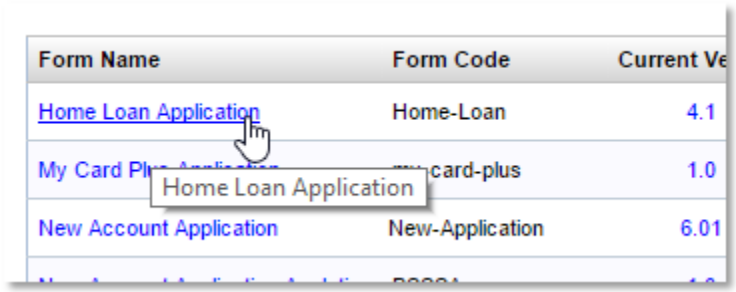
Avoka TM has the ability for a form to maintain its own version of these pages – so that different content may be displayed to the user for a particular form. Note that this setup is optional, and if not configured, the default pages referred to in section 2.2 are used.

If configured, then these custom pages also require the same changes that were applied in section 2.2 – to include a hidden field with the form code that GTM can use.

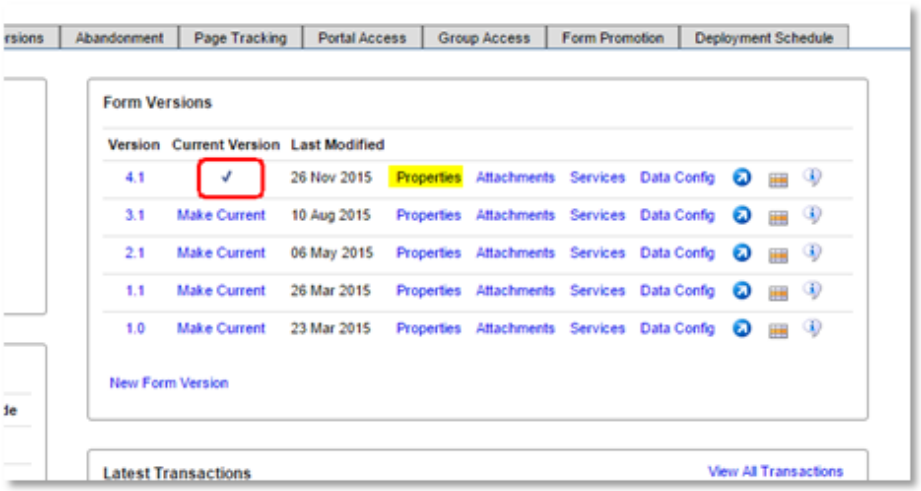
Every time a form is being enabled for GTM it is therefore necessary to check whether any custom pages have been configured, and to edit these.

1) Navigate to Forms -> Forms

2) Locate the relevant form and click on the name entry



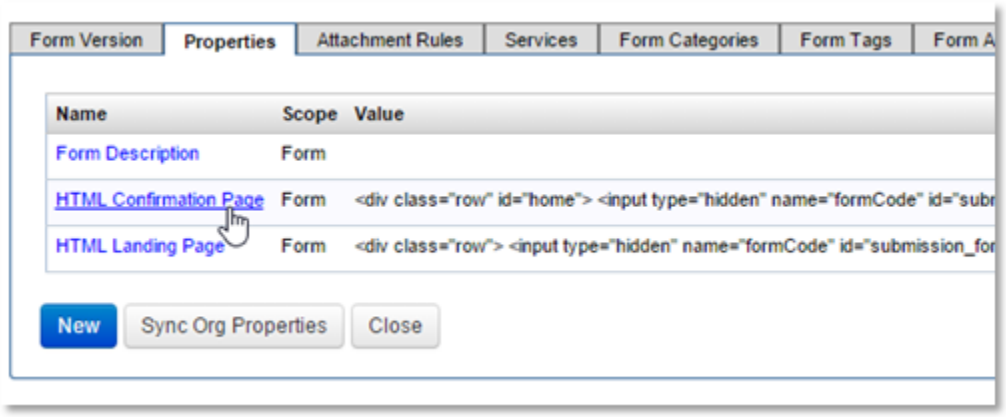
3) In the "Form Versions" box, locate the version that is ticked as the default, and click on the "Properties" link for that version



4) In the list of properties, if you see an entry for any of the relevant pages, they will need to be edited. The list of possible entries that will require editing are:

- HTML Confirmation Page
- HTML Saved Page

5) If present, click on the name of the relevant property



6) Place the following html inside the page content. The location is not critical, but ensure it is not inside any conditional logic (with markup such as '#if ($session.emailed)':

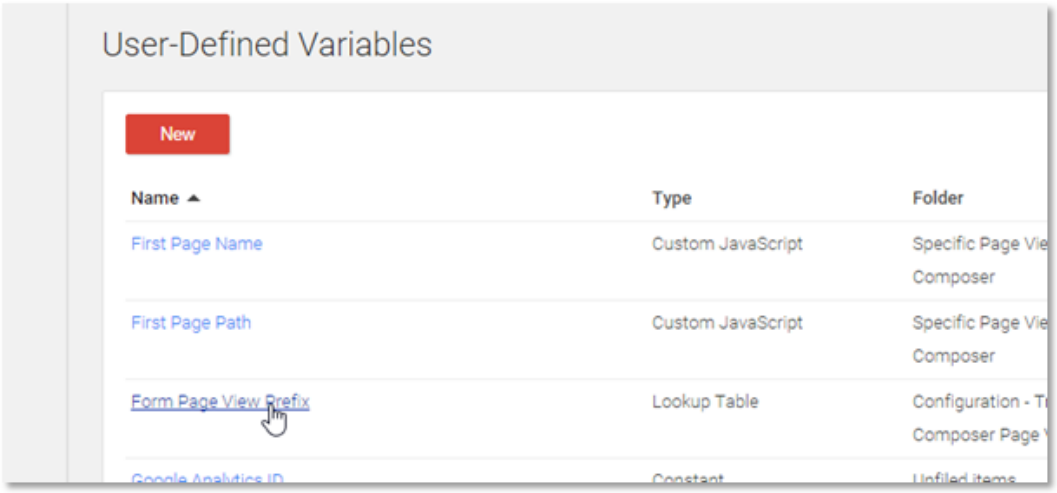
<input type="hidden" name="formCode" id="submission_formCode" value="$form.formCode"/>

```
<input type="hidden" name="formCode" id="submission_formCode" value="$form.formCode"/>
```

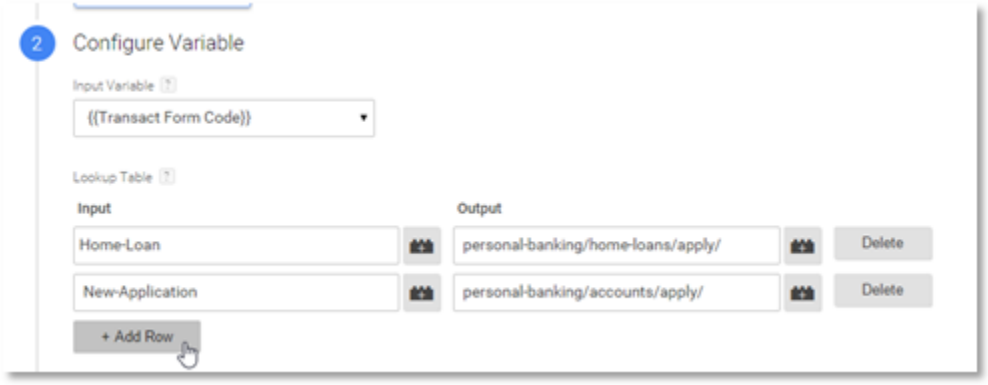
7) Save your changes (the button at the bottom of the page)

Google Tag Manager Configuration

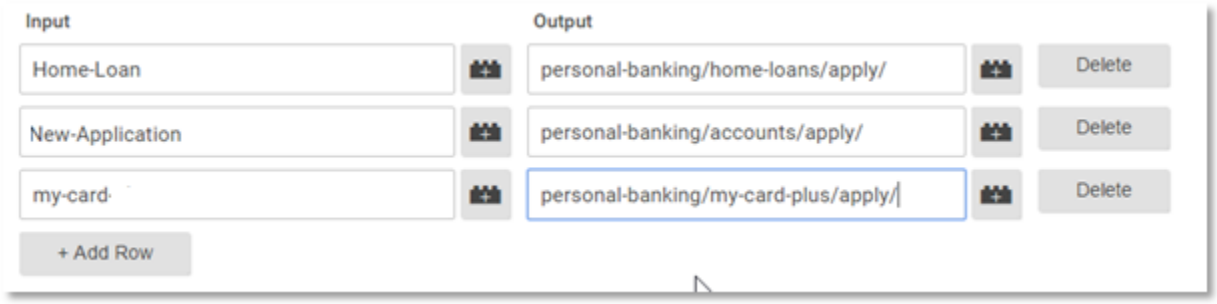
1) Navigate to Container -> Variables -> Form Page View Prefix



2) Click inside the 'Configure Variable' section to allow editing, and add a row to the Lookup Table

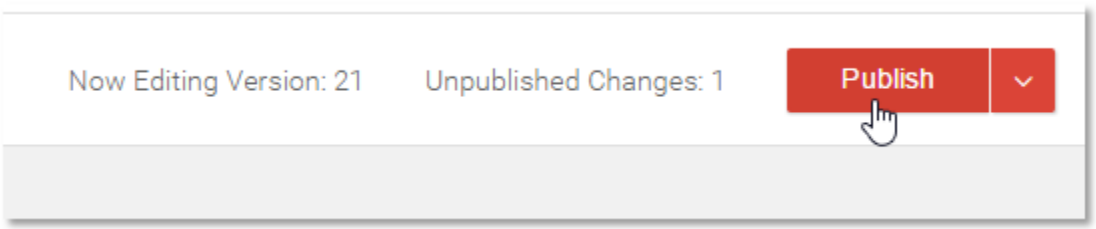


3) In the 'Input' field, type the TM Form Code (see below) and in the Output field enter the Page View prefix that will be used for this form. This prefix will be used for all Page Views generated by this form, and is a unique path that will identify the form in the Analytics engine

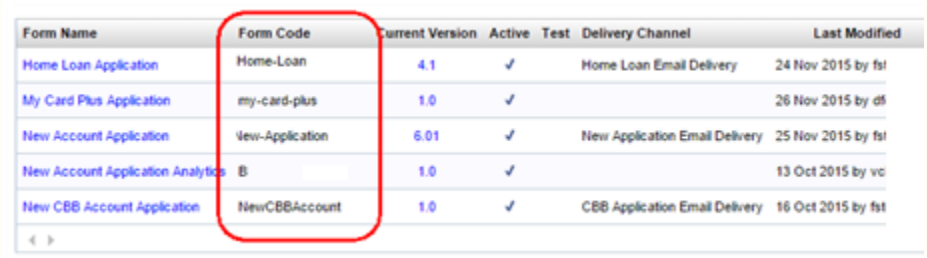


4) Save the Variable

5) Once all changes have been made, it is important to publish the changes using the Publish button in the top right corner of the GTM page. Once published, traffic on that form will generate the new page views which will be sent to Google Analytics



To find the TM Form Code, open Avoka Transact Manager, and navigate to Forms -> Forms. The resulting list will show the form codes for all published forms in that environment:



Form Name	Form Code	Current Version	Active	Test	Delivery Channel	Last Modified
Home Loan Application	Home-Loan	4.1	✓		Home Loan Email Delivery	24 Nov 2015 by fs
My Card Plus Application	my-card-plus	1.0	✓			26 Nov 2015 by df
New Account Application	iew-Application	6.01	✓		New Application Email Delivery	25 Nov 2015 by fs
New Account Application Analytics	B	1.0	✓			13 Oct 2015 by vc
New CBB Account Application	NewCBBAccount	1.0	✓		CBB Application Email Delivery	16 Oct 2015 by fst

Further Information

The following sections include some useful information for GTM administrators that may be wishing to maintain the container, or make changes going forward.

Navigating and locating items in GTM

A folder structure has been setup inside the container to group relevant items together. By navigating to the Folders section (in the left-hand menu structure) items can be located a little more easily. The folder structure consists of the following folders:

- **Configuration - Transact Composer Page Views** – contains variables that relate to the changeable configuration. In particular, the 'Form Page View Prefix' variable that an administrator needs to edit to add or change a form's analytics settings is located here.
- **General In-form Navigation Page Views - Transact Composer** – contains Variables, Triggers and Tags that are involved in generating the page views for in-form navigation – such as registering to receive custom events when a user changes sections, as well as the variables that resolve the section name from the custom event, to be used in the page view Tag.
- **Specific Page Views - Transact Composer** – contains Variables, Triggers and Tags for the page views that are to be generated for all pages outside of the actual Composer form – such as confirmation pages, attachments pages and landing pages.

General Functional Overview

In-form Page Views

To enable page views to be generated as a user navigates through the Composer form, a few key GTM elements have been configured.

- Register Page Change Form Events (Tag)
  - Custom HTML tag type
  - Injects some javascript that utilises the Composer libraries to detect when a user has changed pages, and sends a custom event to GTM – the custom event contains information relating to which section has been opened.
  - Fires on Document Ready, but only in the SmartForm.html page, and if the form being viewed has been configured with a page view prefix (see the Document Ready with Section Level Analytics Trigger)
- Section Level Page View (Tag)
  - Google Analytics (Universal) tag type
  - Actually creates the Google Analytics page view with the relevant prefix and section information
  - Fires on the custom page.change event that is generated by the user action

The other elements in this In-form Navigation folder in GTM are triggers and variables that support these two tags – mainly for extracting details from the generated custom event, as well as ensuring the page views only fire in relevant Composer form pages.

Other Page Views

For all other page views, a fairly simple paradigm has been followed of a single Trigger and single Tag. It should be noted though that these items also rely on other existing Variables, but going forward it is unlikely that these supporting variables would need to change.

As an example, the Submission Complete page views are configured as follows (and the other pages follow the same

- Submission Complete Page Opened (Trigger)
  - Page View Trigger
  - Fires on DOM Ready
  - Checks that the Page URL contains 'confirmation.htm' and that a form prefix has been configured for this form – which automatically ensures the page is a TM form confirmation page, and that the form has been set to use Google Analytics
- Submission Complete Page View (Tag)
  - Google Analytics (Universal) tag type
  - Uses the configured form prefix with a hard-coded page name – i.e. {{Form Page View Prefix}}submission
  - Fires on the Submission Complete Page Opened trigger

Related articles

- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services
- Validating Form Data Against a Schema in Groovy

How to switch address lookup provider

Unknown macro: 'redirect'

Introduction

The following instructions apply to clients that are currently using Veda as their address lookup provider and would like to change their provider to Mastersoft without having to

1. Re-publish their existing forms.
2. Upgrade their TM version
3. Upgrade their Composer version

Address Provider Types

Some address providers will return information in one call. Almost all the address information is returned in the first lookup. Mastersoft is one such provider.

Other providers require a two phase call process. Where the first call will return several partial addresses, once the user selects one of these addresses a second call is made to retrieve the detailed information for the selected address. Veda is one such provider using this method.

Switching from a two call to a one call provider (Veda to Mastersoft)

There are currently two Veda Dynamic Data Services (DDS) in TM for versions prior to 4.1.8

"Veda Address Search" - First call to Veda that returns a list of address candidates.

"Veda Address Detail" - Returns the Detailed data for the address based on an address selection from the first call candidates

The aim of the switch is to use a new provider without having to republish existing forms. This requires that the existing Veda Dynamic Data Services be modified to now call Mastersoft. The published form hasn't changed so it still expects the DDS names to remain identical and that two calls will be used.

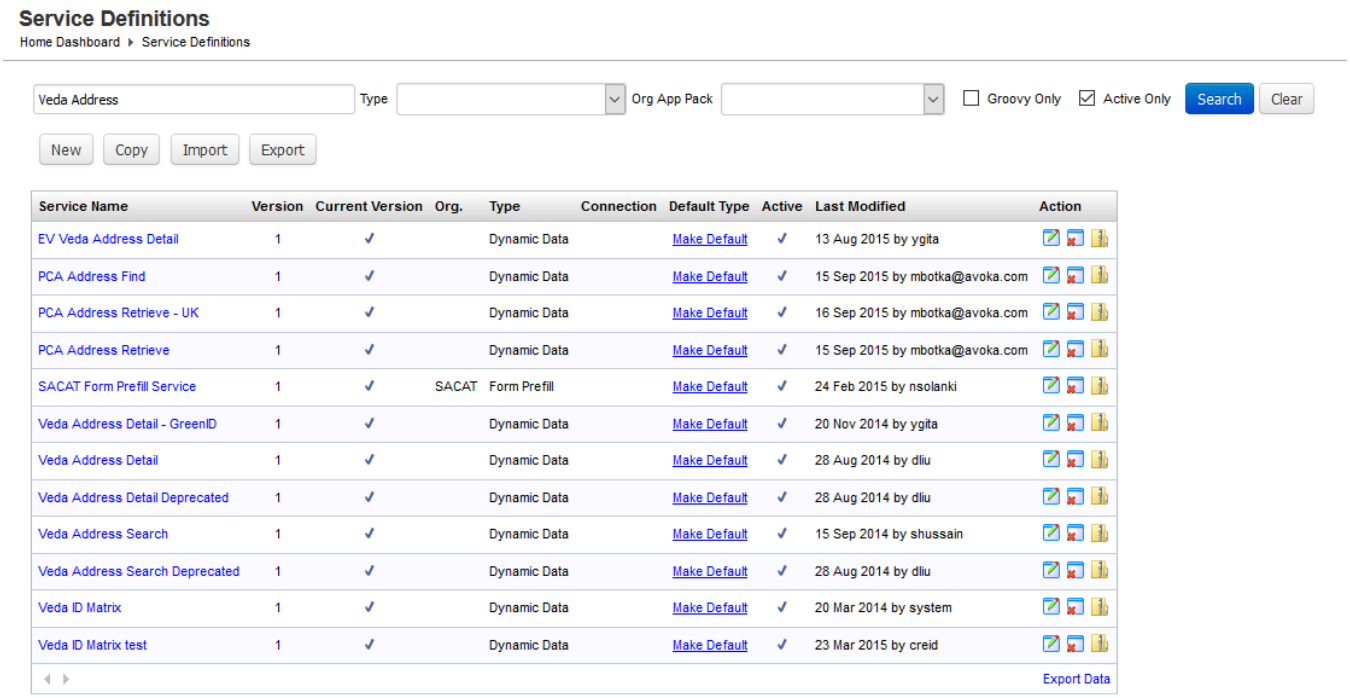
We have developed 2 new Dynamic Data Services to achieve this. These need to replace the current Veda DDS of the same name (see figure below). Once this is done the form will start to call Mastersoft for the address information. The returned data will be formatted in an identical way to how Veda formatted the data for both the first and second calls. The end user will not see any difference.

Step by step set up

1) Ensure you have the correct service definition for your version of TM. This is provided by Avoka.

2) Log into TM and open the service definitions screen. System=>Service Definitions

3) Search on Veda Address to filter down the Veda definitions below.



4) Edit the Veda Address Search Service

5) Rename the "Veda Address Search" to "Veda Address Search Deprecated" and save

6) Edit the Veda Address Detail Service

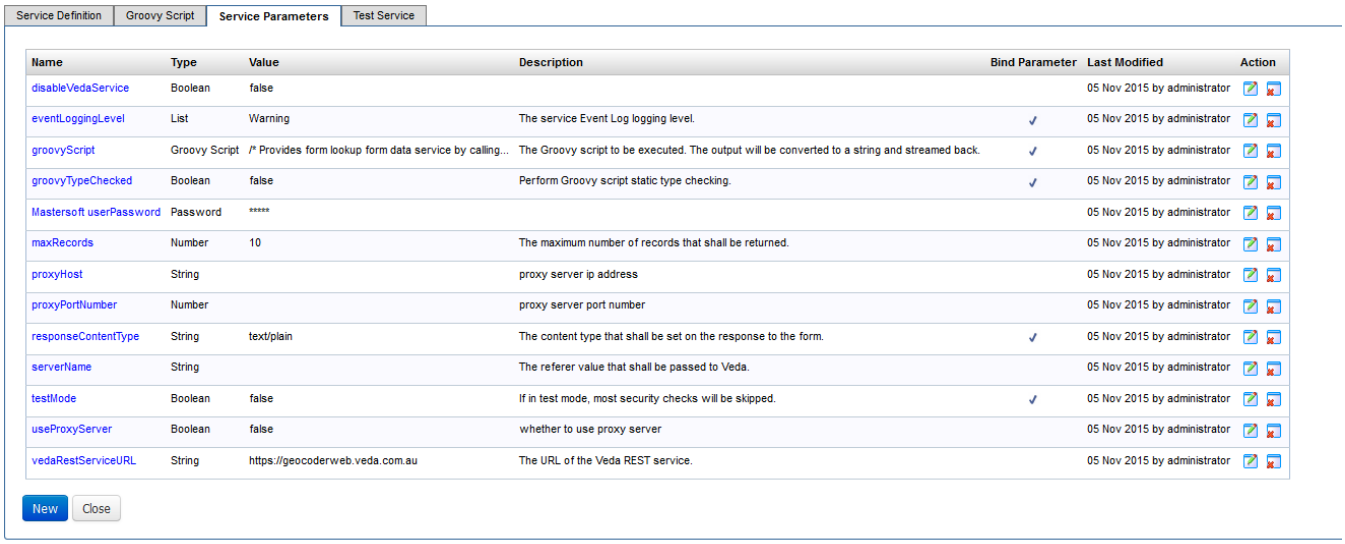
7) Rename the "Veda Address Detail" to "Veda Address Detail Deprecated" and save.

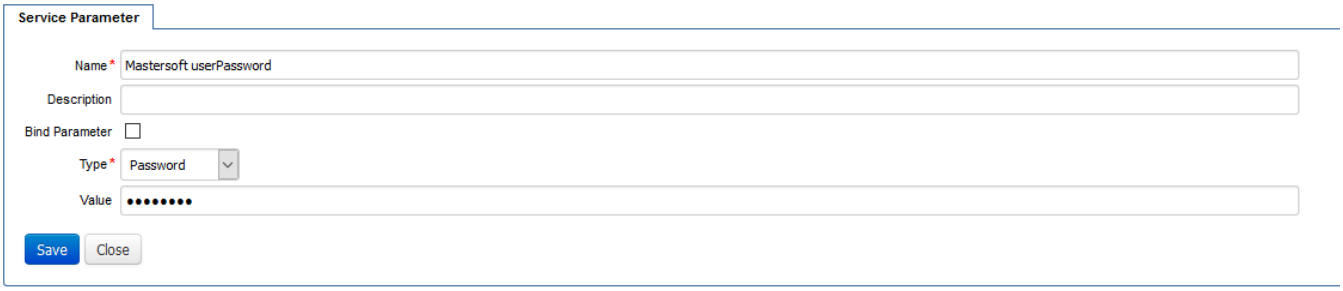
8) Return to the Service Definitions screen and select the Import button.

9) Import the two services provided by Avoka. The names of these services must be "Veda Address Search" and "Veda Address Detail"

10) After the import. Edit "Veda Address Search"

11) Select the Service Parameters tab. The userPassword parameter needs to be updated with the Mastersoft userPassword. The example below is a dummy example.





12) Repeat steps 10 and 11 for "Veda Address Detail" updating with the same user/password

13) The service is now ready. A form that uses the address validation should be used to test immediately.

Related articles

Old internal confluence link - https://support.avoka.com/confluence/pages/viewpage.action?spaceKey=SFF&title=Composer+-+Address+Lookup+Widget+-+4.1

- How to stop the mandatory marker from wrapping
- Data Validation (Composer)
- Mandatory Marker
- Maguire Error handling on different platforms V4
- Field Validation in Composer

How To Open TaskUserWebService in SOAP-UI

Unknown macro: 'redirect'

It is good to read section 4 of the transact_web_service_guide_v4.2.pdf prior to how to.

The TaskUserWebService and TaskUserWebServiceV2 WSDL in Transaction Manager are located in a secure context path. This wiki article shows the workaround steps to successfully import the WSDL's and call methods in SOAP-UI

Task Web Services V2 requires you to authenticate using a username and password, It is possible to get the wsdl from the non secure context however you will be unable to call the secured services from the unsecured WSDL

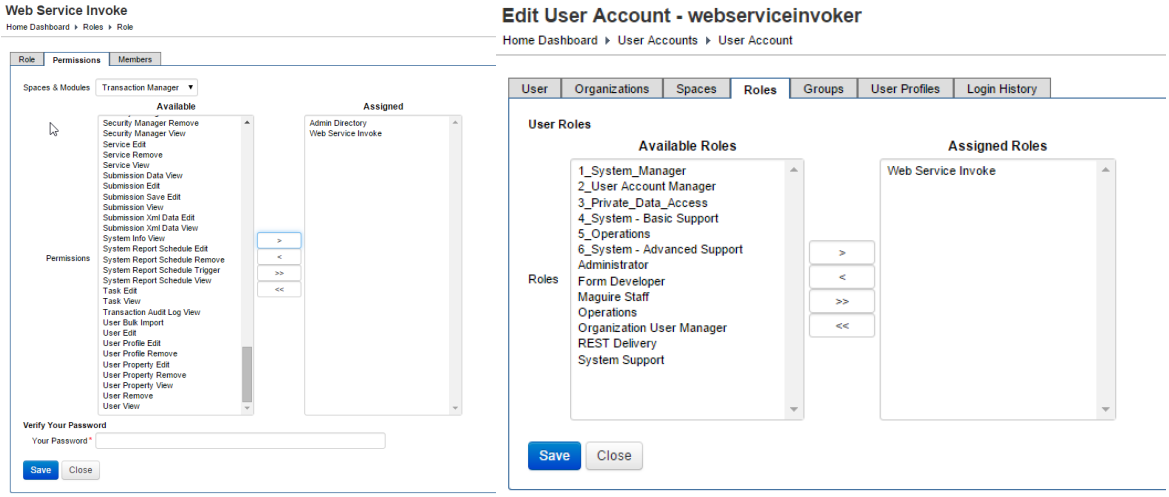
Step-by-step guide

a. WSDL URL

https://{Server Name}/manager/secure/services/TaskUserWebServiceV2?wsdl

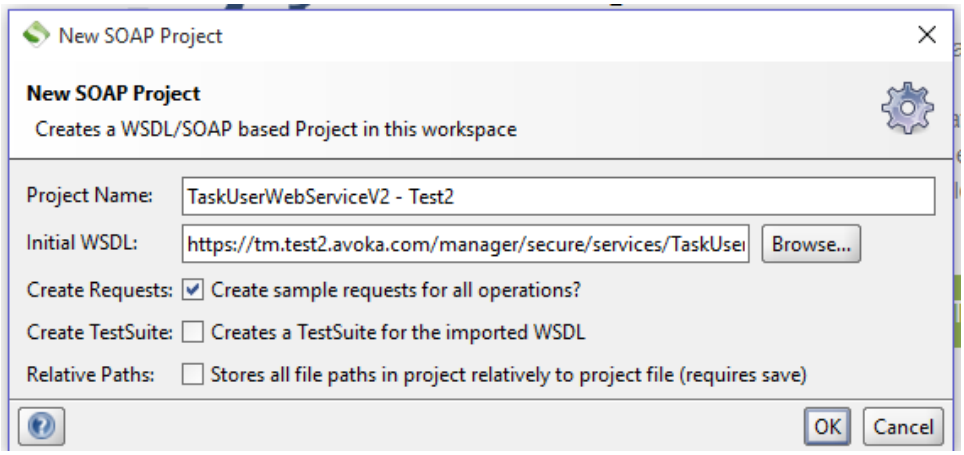
> Note: Do NOT use the unsecure URL https://{Server Name}/manager/fcservices/TaskUserWebServiceV2?wsdl

b. Create a service user that has been assigned a Role that has the Web Service Invoke permission

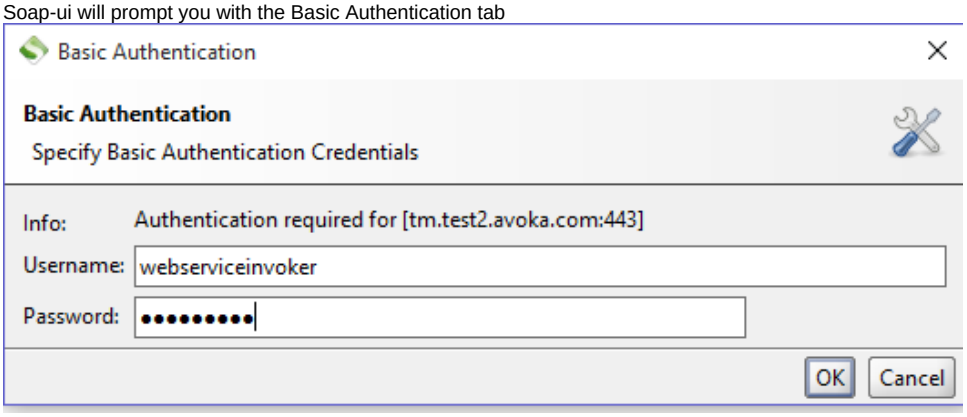


> Note: Users with the Adminitrator role have the Web Service Invoke permission

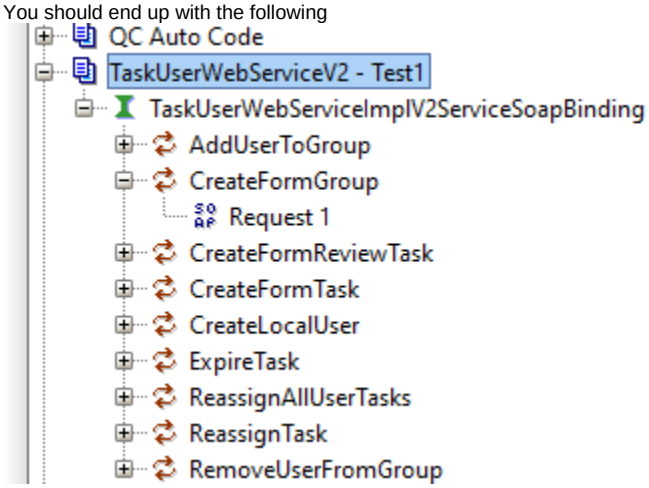
c. Create SOAP-UI project the wsdl in Step 1



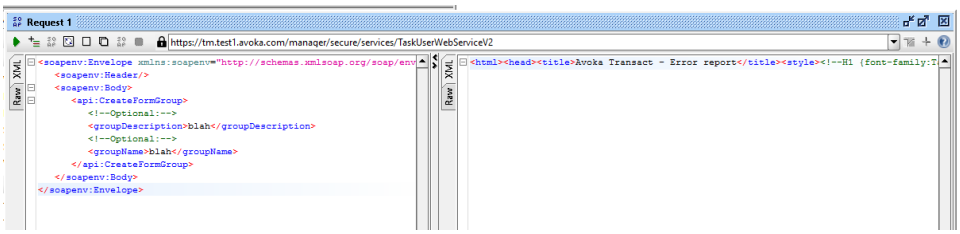
Soap-ui will prompt you with the Basic Authentication tab



You should end up with the following

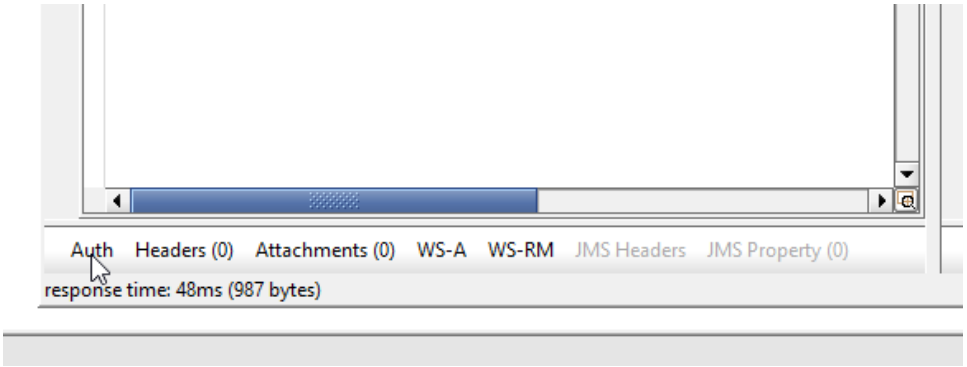


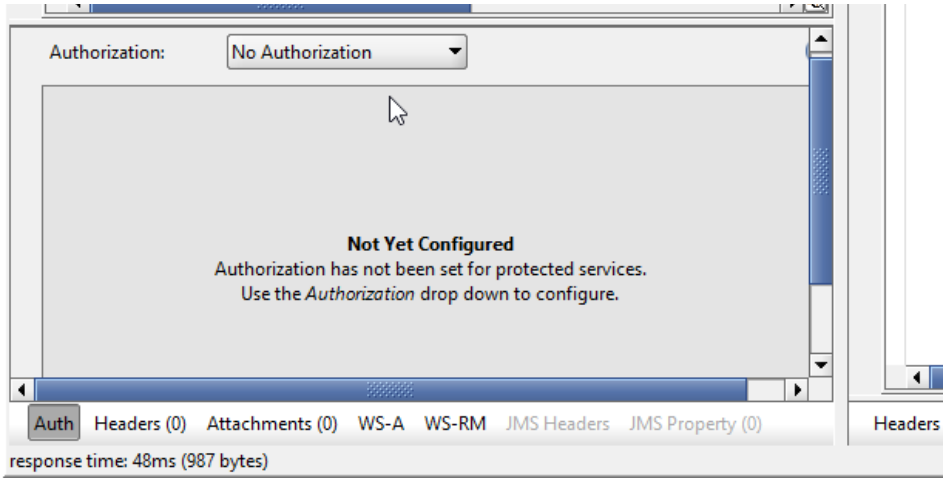
d. Try calling the web services with no authentication. Test Creating a Form Group. The response on the right tell you you are not authenticated.



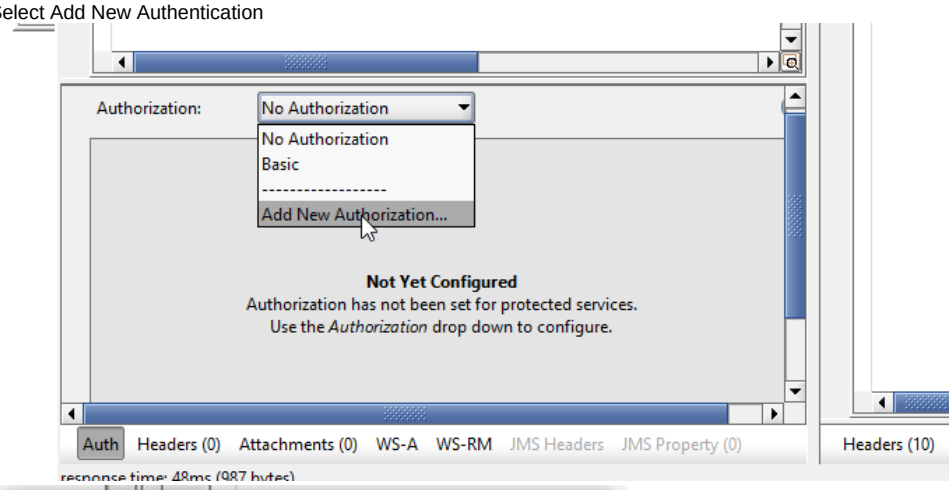
e. Setup Authentication

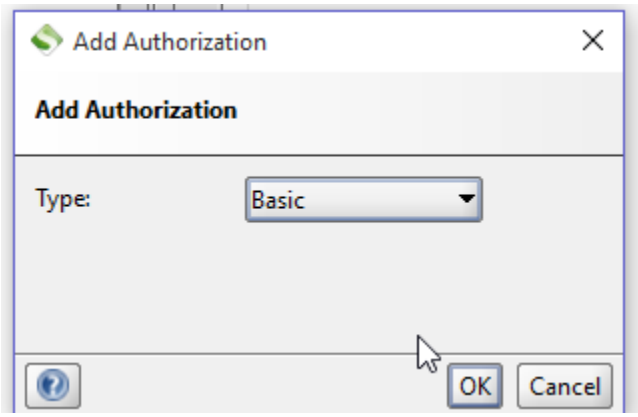
Click on the Auth buttong on the bottom left of the Request1 page.

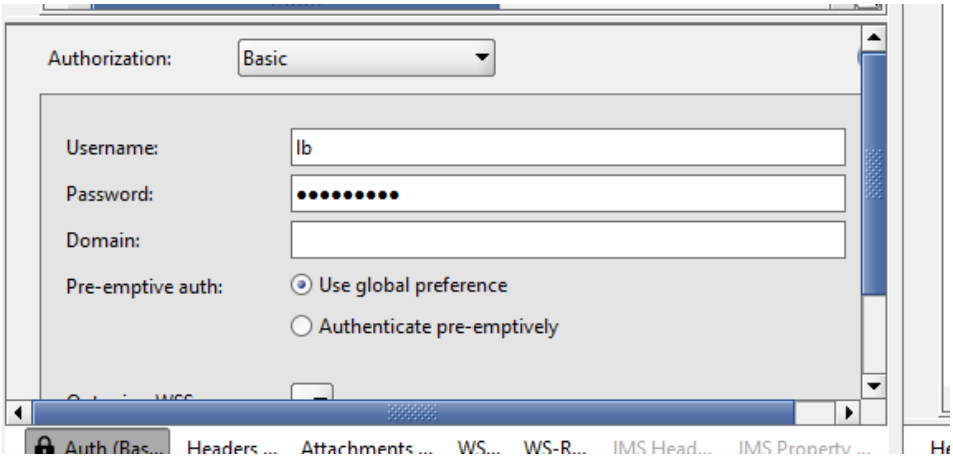




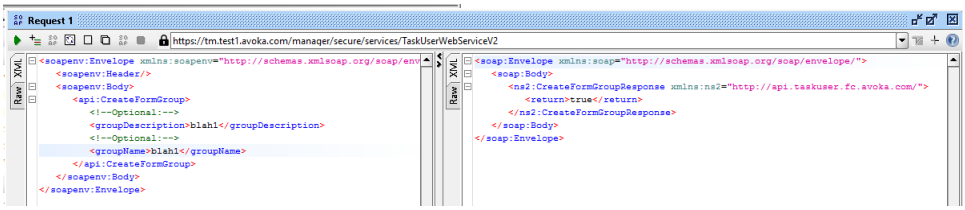
Select Add New Authentication







Next re run the request



Related articles

- Creating new form tasks in a Groovy delivery service
- Managing multiple service endpoints and credentials for external service calls
- Groovy Services
- Understanding and integrating with Docusign
- Importing a JavaScript library into a Composer form

Transaction Manager Integration with Salesforce

Unknown macro: 'redirect'

Avoka Transact for Salesforce

This article is now deprecated.

Avoka Transact for Salesforce was made available with Transact 4.2 and consists of a managed package that gets installed into the Salesforce environment to facilitate tight integration between these 2 systems. For more information please see the Transact for Salesforce space.

Introduction

As of version 3.6.5, the Transact Manager API includes the ability to interface with Salesforce.com. Salesforce objects can be created, retrieved, updated or even deleted, which can be useful in many scenarios. For example, a Data Prefill service may retrieve relevant customer data from Salesforce.com for pre-populating into a form. Alternatively, a Submission service could create or update an existing customer record with newly entered data.

The following Salesforce interfaces are supported

- SObject - postOperation (create an object)
- SObject - getOperation (retrieve an object)
- SObject - patchOperation (update an object)
- SObject - deleteOperation (delete an object)
- Query - getOperation (perform a Salesforce query to retrieve one or more items)

Security

Using the salesforce web services requires a pre-configured user to be setup in the Salesforce portal. The following credentials are required:

- **SalesForce Username**
- **SalesForce Password Security Token**
- **SalesForce Client Id**
- **SalesForce Client Secret**
- **SalesForce Login Server URL**

See the section below for instructions on obtaining these credentials.

Establishing a Salesforce connection with Groovy script

The **com.avoka.component.salesforce.SalesForceClient** class provides a client object with web service factory methods.

The Groovy script example below illustrates using the SalesForceClient object to create an SObject Salesforce web service object, which is then used to retrieve a contact object. In this example configuration Service Parameters are accessed via the params map object.

```
import groovy.json.JsonSlurper

import com.avoka.component.salesforce.SalesForceClient
import com.avoka.component.salesforce.service.SObject

def username = serviceParameters["SalesForce Username"]
def passwordToken = serviceParameters["SalesForce Password Security Token"]
def clientId = serviceParameters["SalesForce Client Id"]
def clientSecret = serviceParameters["SalesForce Client Secret"]
def loginServerUrl = serviceParameters["SalesForce Login Server URL"]

// Create a SalesForce client using the service params with the configured connection details
def salesForceClient = new SalesForceClient(username, passwordToken, clientId, clientSecret, loginServerUrl)

// Create a SObject service
def sObject = salesForceClient.getSObject()
```

```
// In this example the contact ID is provided as a service parameter. In practice it can be retrieved using a com.avoka.component.
salesforce.service.Query object
def contactId = serviceParameters["Contact ID"]

// Retrieve a contact from SalesForce. The return value is the JSON representation
def jsonResponseString = sObject.getOperation("Contact", contactId)

// Example of how to use the results to populate an Avoka Transact form in a Prefill service
Map<String, Object> responseMap = new JsonSlurper().parseText(jsonResponseString)

def xmlDoc = new XmlDoc(schemaSeed)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/FirstName", responseMap.FirstName)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/LastName", responseMap.LastName)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/Email", responseMap.Email)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/Phone", responseMap.Phone)
```

When using any of the methods in the **SObject** class, SalesForce objects are represented by JSON. For example, a new contact can be created with the following Groovy code snippet:

```
def bodyJsonString = "{
"FirstName": "First",
"LastName": "Last",
"Phone": "0999999999",
"Email": "example@avoka.com",
"AccountId": "0019000000QleXDABC"
}"

// Create a SalesForce client using the service params with the configured connection details
def salesForceClient = new SalesForceClient(username, passwordToken, clientId, clientSecret, loginServerUrl)

// Create a SObject service
def sObject = salesForceClient.getSObject()

// Create a new contact in SalesForce. The return value is the Contact ID
def newContactId = sObject.postOperation("Contact", bodyJsonString)
```

The **Query** class provides the means to perform more generic, or targeted data queries using the SalesForce SOQL query language. Relations between objects can be viewed and administered in the Schema builder within the SalesForce developer console. Data is returned in JSON format. An example of using the Query object to prefill a form can be seen below.

```
// Create a Query service
def query = salesForceClient.getQuery()

// Get a contact with Account details from SalesForce returned as a JSON string - The contact email address is used for the lookup
def jsonResponseString = query.getOperation("SELECT id,firstname,lastname,phone,email,a.name,a.industry FROM contact c, c.
account a WHERE c.email='example@avoka.com'")

// Example of how to use the results to populate an Avoka Transact form in a Prefill service
Map<String, Object> responseMap = new JsonSlurper().parseText(jsonResponseString)
List<Map<String,Object>> recordList = responseMap.records
def record = recordList.get(0)

def xmlDoc = new XmlDoc(schemaSeed)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/FirstName", record.FirstName)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/LastName", record.LastName)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/Email", record.Email)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/Phone", record.Phone)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/Account/Name", record.Account.Name)
xmlDoc.setText("/AvokaSmartForm/SalesForceData/Account/Industry", record.Account.Industry)
```

Obtaining Saleforce Credentials

The following sections will outline how to obtain the four credentials required by the API to create a connection to Salesforce.com:

- **SalesForce Username**
- **SalesForce Password Security Token**
- **SalesForce Client Id**
- **SalesForce Client Secret**

SalesForce Username

A new 'Avoka Transact' user should be created in SalesForce.com to accommodate the integration credentials. The user should be created as per below. Note the email address should be specified as your email address or your companies sales force administrator. The **username** is what is used for the Avoka Transact SalesForce Username credential.

SalesForce Password Security Token

The Password Security Token integration credential is a string concatenation of two Salesforce.com credentials - password and security token:

SalesForce Password Security Token format: <password><Security Token>

At the completion of the previous user creation step, SalesForce will send a confirmation email. This email will contain a link for the user to login, as well as the Security Token for the account. The SalesForce Password Security Token that the Transact Manager API requires can then be constructed from the user password (which is set upon first login to Salesforce.com - by following the link in the email) and the Security Token specified in the email.

It is also possible to request a new Security Token from within the SalesForce console. Login with your Transact user credentials created above and select 'Setup' from the User dropdown.

You may request a new Security Token by selecting Reset My Security Token.

SalesForce Client Id and Client Secret

These two credentials are created by configuring a connected app within Salesforce.com:

In the App Setup menu select Create > Apps

On the Connected Apps table select the New button.
Fill in the details as entered below:

Connected App Name: Avoka Transact
API Name: Avoka_Transact
Contact Email: <as required>
Enable OAuth Settings: true
Callback URL: <irrelevant - use any>
Selected OAuth Scopes: Access and manage your data (api)

When viewing the new Connected App:
The SalesForce Consumer Key (Client Id) is displayed under API (Enable OAuth Settings).
The SalesForce Consumer Secret (Client Secret) can be revealed by clicking the 'Click to reveal' link.

Related articles

- Managing multiple service endpoints and credentials for external service calls
- Groovy Services
- Understanding and integrating with Docusign
- Importing a JavaScript library into a Composer form
- XML Binding in Composer Forms

Transact Manager - Sharepoint Integration

Unknown macro: 'redirect'

Transaction Manager Integration with Sharepoint

Integration between Transaction Manager and SharePoint is available from Transaction Manager 3.5 via a Java integration component.
The following Sharepoint interfaces are supported

- Copy - copyIntoItems
- Copy - getItems
- Lists - addAttachment
- Lists - getListItems
- Lists - getListCollection
- Lists - updateListItems
- Lists - checkInFile
- UserGroup - getUserInfo
- UserProfileService - getUserProfileByName
- Webs - getWeb
- Webs - getWebCollection

Security

Before using the sharepoint web services make sure the user you use to authenticate has the "***Use Remote Interfaces - Use SOAP, Web DAV, the Client Object Model or SharePoint Designer interfaces to access the Web site.***" permission.

Establishing a SharePoint connection with Groovy script

The com.avoka.component.sharepoint.SPClient Class provides a Microsoft SharePoint Client class with web service factory methods.

The Groovy script example below illustrates using the SPClient object to create an List SharePoint web service object, which is then user to lookup a list collection. In this examples configuration Service Parameters are accessed via the params map object.

```
import com.avoka.component.sharepoint.SPClient
import com.avoka.component.sharepoint.service.List

// Create a SharePoint client using the service params map with the configured connection details
def spClient = new SPClient(params.username, params.password, params.url, params.domain)

// Create a List web service
def listService = spClient.getLists()

// Get a List collection map
def collectionMap = listService.getListCollection(params.siteContext)
```

Further documentation and examples on the interfaces relating to SharePoint Integration are available in the Avoka Transaction Manager API Javadocs.

Related articles

- Log files best practice
- Managing multiple service endpoints and credentials for external service calls
- Groovy Services
- Collaboration Jobs - Separate out emails from Task Assigns using multiple actions
- Understanding and integrating with Docusign

How to decrypt Json Web Token (JWT) in TM

Unknown macro: 'redirect'

In TM 5, there are already library built in which you can reference to decrypt JWT, here is a step by step on how to do it

1. Save the certificate string which contains the public key as a service parameter
2. Import the necessary libraries

   import groovy.json.JsonSlurper
   import java.io.InputStream;
   import java.security.PublicKey;
   import java.nio.charset.StandardCharsets;
   import java.security.cert.CertificateException;
   import java.security.cert.CertificateFactory;
   import java.security.cert.X509Certificate;
   import com.auth0.jwt.JWTVerifier;
3. **def certificate = serviceParameters?.PublicAuthorizationCert**
   InputStream is = new ByteArrayInputStream(certificate.getBytes(StandardCharsets.UTF_8))
   CertificateFactory f = CertificateFactory.getInstance("X.509")
   def cert = (X509Certificate)f.generateCertificate(is)
   PublicKey pk = cert.getPublicKey()
   JWTVerifier jwtVerifier = new JWTVerifier(pk);
   Map<String, Object> verify =jwtVerifier.verify(appToken);
4. At this point if every thing is ok, you will get a Map object which you can use to extract data in the JWT. If for any reason like the public key is incorrect, or the token has expired, etc, an error will be thrown.
5. To extract data from JWT
   def field1= verify.get("<path to the data>");

Maestro

Unknown macro: 'redirect'

Maestro Forms

Unknown macro: 'redirect'

Components

Unknown macro: 'redirect'

- Configuring Payments in Maestro
- How to update a Data-Driven Dropdown triggered by an input Text Field
- How to use Data-Driven Components in Maestro

Configuring Payments in Maestro

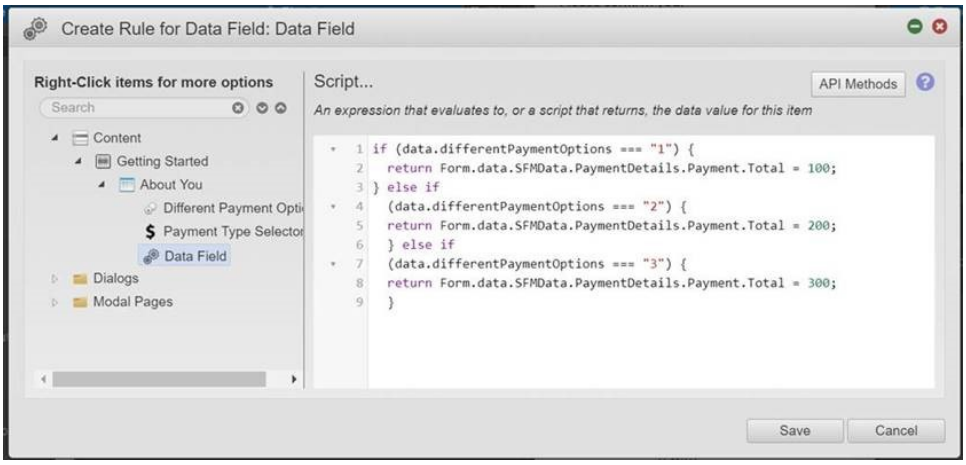
Unknown macro: 'redirect'

Businesses often require payment interface functionality within a form. The steps below detail the procedure to follow in order to configure payment interface functionality within your smart form. Note that Transact does not store, process or transmit payment card data; any payment card data interaction is directly between the end user's browser and the payment gateway or tokenisation provider. Avoka recommends that when enabling Transact's payment interface features, you ensure appropriate security, access and change control measures are in place to prevent accidental or malicious change of configurations.

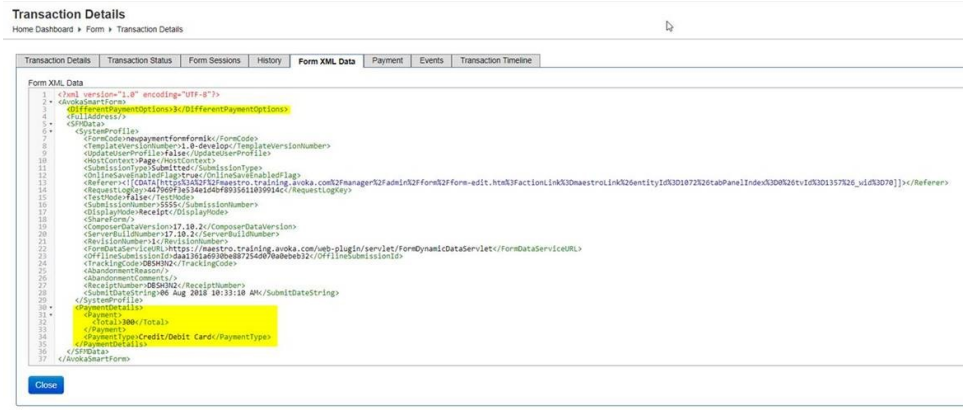
Maestro

1. Include a Payment Type Selector component in the form. **Note**: It has to be non-zero for the Payment page to appear after submission.
2. If you require a Fixed payment amount, update the Payment Total ($) property of the Payment Type Selector.

3. If you require a flexible payment amount, use a data field to update the XML element **Form.data.SFMData.PaymentDetails.Payment.Total**



4. Resulting XML values in Transact Manager will trigger the payment pages to be displayed after the form is submitted:



How to update a Data-Driven Dropdown triggered by an input Text Field

Unknown macro: 'redirect'

This article describes how to use the input from a text field to both trigger and to make a dynamic data services call. This DDS call will retrieve the data to populate a data-driven dropdown. We will use the example of a zip code text field where the user types the zip code and a zip code lookup service returns the list of cities in a zip code and populates them in a DDD.

Compatibility

Module	Maestro
Since	5.0
Deprecated	

Step-by-step guide

The general steps involved are as follows:

1. Add a Text Field to your form. This will be the zip code field. Here we presume that the field reference ID for the zip-code field is "zipcode".
2. Add a Data Field to your form. This will store the data returned by the DDS call. We presume that the field reference ID for the Data field is cityDataField.
3. Add a Data-Driven Dropdown to your form. Make the data source for the DDD point to "data. cityDataField".
4. Add an "On Change" script to the Text Field in order to map the request and response parameters for your DDS call and to make the call. It should look something like the Javascript below.

Example Javascript to map request/response fields, to make DDS request and to update the Data Field

```
// The key "ZIPCODE" is presumed to be the input parameter for the
"Zipcode Lookup Service"
var inputFieldRefs = [{
    key: "ZIPCODE",
    ref: "data.zipcode"
}];

// The key "cities" is presumed to be the output parameter returned from
the "Zipcode Lookup Service"
var outputFieldRefs = [{
    key: "cities",
    ref: "data.cityDataField"
}];

var serviceName = "Zipcode Lookup Service";
var dataParams = Form.convertToFieldDataMap(inputFieldRefs, data);

// Character input must be length of valid zipcode before making request
if(data.zipcode && data.zipcode.length === 5) {
  proceed();
}

// Lookup promise
function proceed() {
    // Make asynchronous call to DDS and when the response is received set
the field data or log an error
    DynamicData.call(serviceName, dataParams).then(function(response) {
        // Update the form field data.cityDataField with the response data
formatted for a Data-Drive Dropdown.
        Form.setFieldDataFromResponse(outputFieldRefs, response, data);
    }).catch(function(err) {
        console.log("Error: ", err);
    });
}
```

Make sure you have a service written to return return the data back as JSON in the correct format. The service in our example, "Zipcode Lookup Service", is presumed to return a JSON response format that looks like:

Example DDS JSON response format

```
{"cities":[{"label":"city1","value":"city1"},{"label":"city2","value":"
city2"}]}
```

Related articles

- Developing a Dynamic Data Lookup
- How to create a Dynamic Data lookup using test data
- Repeating and Nested data in Dynamic Data service
- Data Lookup
- How to use Data-Driven Components in Maestro

How to use Data-Driven Components in Maestro

Unknown macro: 'redirect'

This article describes how form authors can use the Data-Driven Components such as a Data-Driven Dropdown or AutoSearch component in Maestro.

Compatibility

Module	Maestro
Since	5.0.0
Deprecated	

Step-by-step guide

The following broad steps describe the process

1. Add a data field which stores your initial data (In our example called "Product List")
2. Add your Data-Driven Dropdown, and configure it to point to a JavaScript object
3. Add a script to your form which creates that object out of the string in your Data Field

Or more generically,

1. Add a script which creates a map
2. Add a component which uses the map

Adding a data field to store initial data

Your data field should contain something like this:

Data Field content

```
[
   {
      "label":"Single investor",
      "value":"singleSuper",
      "brand":"Corporate Bank"
   },
   {
      "label":"Joint investor",
      "value":"joint",
      "brand":"Corporate Bank"
   },
   {
      "label":"Single investor (non-super)",
      "value":"singleNonSuper",
      "brand":"Corporate Bank"
   },
   {
      "label":"Company investor",
      "value":"company",
      "brand":"Corporate Bank"
   },
   {
      "label":"Product A",
      "value":"p3",
      "brand":"Retail Bank"
   },
   {
      "label":"Product B",
      "value":"p4",
      "brand":"Retail Bank"
   },
   {
      "label":"Product Z",
      "value":"p5",
      "brand":"Retail Bank"
   }
]
```

Notice that the "label" & "value" keywords correspond to the Data-Driven Dropdown properties.

This example also includes additional data (brand) which may be used later. Consider it optional for now.

Tip

The following online tool is great for converting an excel file to JSON format:

http://www.csvjson.com/csv2json

When you copy and paste columns, just select a "tab" separator. Ensure that your columns have titles.

Adding a script which creates a map

JavaScript

```
data.$MccCodes = [
  {
    "MCC": 742,
    "Description": "VETERINARY SERVICES",
    "label": "742 VETERINARY SERVICES"
  },
  {
    "MCC": 763,
    "Description": "AGRICULTURAL CO-OPS",
    "label": "763 AGRICULTURAL CO-OPS"
  },
  {
    "MCC": 780,
    "Description": "HORTICULTURAL AND LANDSCAPING SERVICES",
    "label": "780 HORTICULTURAL AND LANDSCAPING SERVICES"
  },
  {
    "MCC": 1520,
    "Description": "GENERAL CONTRACTORS",
    "label": "1520 GENERAL CONTRACTORS"
  }
];
```

Adding a Data-Driven Dropdown, and configuration

After dragging on a "Data-Driven Dropdown" component, configure the following:

1. Choose a name for the Data-Source. It needs to be unique, and you can ignore the $ symbol, it's a convention, not syntactical. We will use this name later, and in this example has been set as "data.$productData"
2. The Label Field and the Value Field need to map to keys in your JSON (not required to edit anything if you have used the template above)

Adding a script which creates an object out of your Data Field

At some point in form execution (normally on page transition or shoehorned into a validation script or calculation script on an unrelated field), you need to add some code to convert your field value into an object.

Do not add the code to a click script on the Data-Driven Dropdown, as the execution order of the script & other events will trip you up.

Script to create the object

```
// data.productList is the field which contains the JSON string with keys
and values
// data.$productData is the object which the Data-Driven Dropdown uses
// brand is an optional key in the JSON string
// data.tenant is a field which contains the name of current brand, and in
this example is prefilled into the form by TM services on form render.

var tempData = JSON.parse(data.productList );
data.$productData = tempData.filter(function(arrValue,arrIndex,arr) {
  if (arrValue && arrValue.brand && arrValue.brand.toLowerCase().indexOf
(data.tenant.toLowerCase())!=0) {
    return false;
  }
  return true;
});
return undefined;
```

Related articles

- Enforcing required configuration properties in custom native components
- Loading External Javascript Libraries
- How to detect if a user is on a mobile device
- Advanced Debugging of Maestro Forms
- Creating native components for Maestro (Eclipse/ IntelliJ Developers)

Data

Unknown macro: 'redirect'

- Understanding and Accessing Prefill Data in Maestro

Understanding and Accessing Prefill Data in Maestro

Unknown macro: 'redirect'

It's useful to understand the data processing pipeline that Maestro goes through when populating a form (at least at a high level).

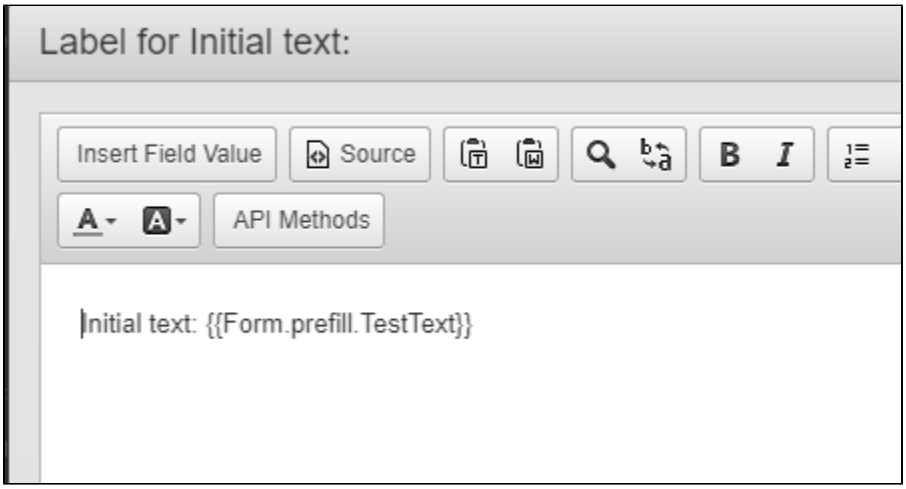
1. The prefill XML (provided by prefill services in Transact Manager) is parsed and turned into an in-memory Javascript object graph with the same structure as the XML.
2. An object graph called "data" is constructed, using the fields in the form to determine the structure of this object graph.
3. The "data" object is initialized from the prefill object.
4. The data object is further modified as calculation rules fire.

The "data" object is live, in that it is continually changing as the user interacts with the form, and calculation rules fire. The "prefill" object never changes.

In some scenarios, it may be helpful to refer to the original prefill information. For example:

- We may want to be able to reset a field to its original value.
- We may want a display field to show the original value of a prefill value, and not change if that value is changed by the user.

In this case, you can directly reference the prefill object as shown in the screenshot below.



If you would like to see the prefill data object in your browser's debugger, type: "maestro.Form.prefill" at the console prompt.

Data Validation

Unknown macro: 'redirect'

- Implementing a pre-submit validation

Implementing a pre-submit validation

Unknown macro: 'redirect'

A common requirement in Maestro forms is trying to implement a pre-submit validation call to the server.

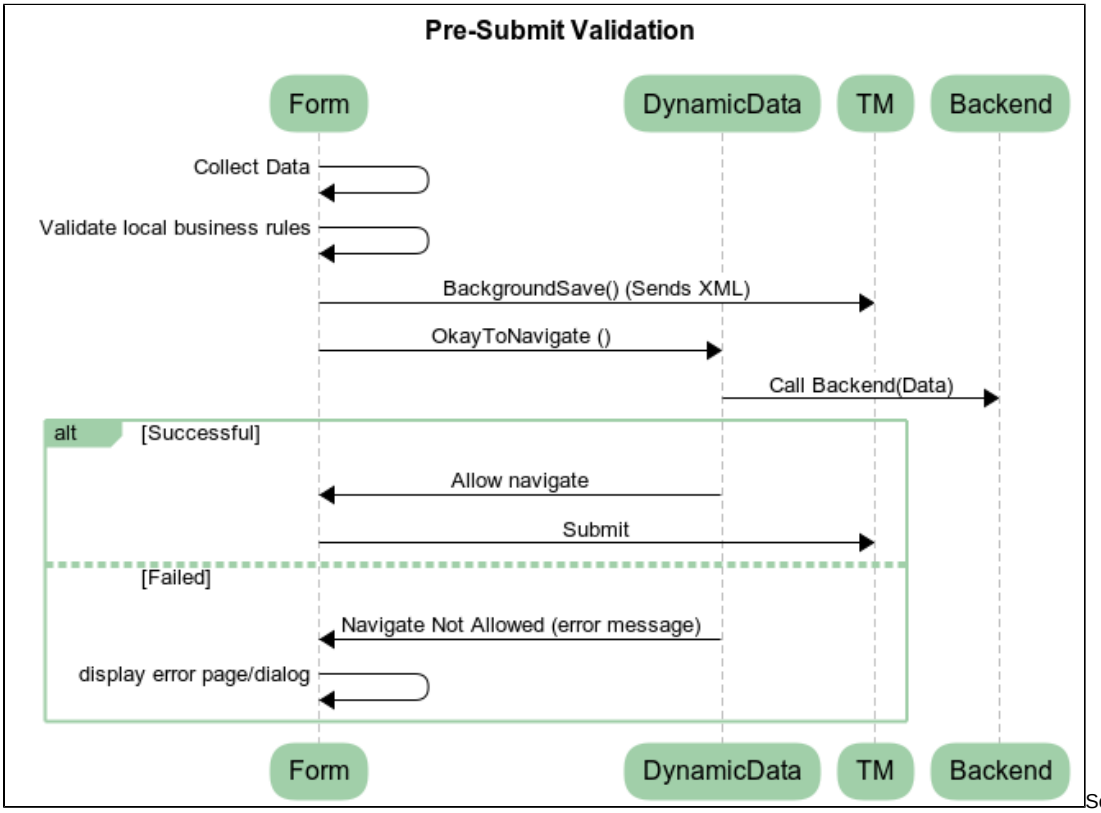
What we want to do is something like this:

1. The User clicks the submit button.
2. Instead of processing the submit immediately, call a dynamic data service, passing the data from the form.
3. The Dynamic data service will process the data, optionally call some backend service(s), and return a go/no-go decision back to the form.
4. If the decision is "go", then the submit occurs.
5. If the decision is "no-go", then an error is displayed to the user.

There are a few additional subtleties...

- We should validate that all the mandatory rules and validation rules have passed before calling the dynamic data service.
- We need a way to pass the data to the server. One common pattern for doing this is to explicitly invoke an background save - this will result in the XML being available to the Dynamic Data service. Another pattern is to collect the data as a JSON object and pass it in the Dynamic Data service.

The overall flow is shown in the following sequence diagram:



Sounds easy?

The problem with this is that each of the functions called is asynchronous. This means that it doesn't return immediately, it returns some time later. Which means that we need to understand how promises work.

Please refer to this article for a brief explanation of promises in Maestro. This will really help to understand the code.

Promises in Maestro

The code

Here is the original code for the Submit click event of a standard Maestro form:

```
Form.submit(item.properties.submissionMethod, item.properties.legacyMethod).then(function(response){
    Form.fireRule("onSuccess", item, data, response);
}, function(err) {
    Form.fireRule("onFailure", item, data, err);
});
```

Some notes:

- The Form.submit() method includes a call to validate() which does local mandatory and rule validation.
- Depending on your template, you may not have access to the Submit button. If you don't, you need to speak to your template developer.
- It's tempting to think you can implement this using the Pre Submit event at the Form level. However, this doesn't work, because there is now way to prevent the Submit from occurring.

The new code looks something like this:

```
parameters.param = data.lastName;
Form.backgroundSave()
.then(function(backgroundSaveSuccessful) {
    return Form.validate();
}).then(function(validateSuccessful){
    return DynamicData.call("MyDynamicDataService", parameters);
}).then(function(dynamicDataCallResult){
    console.log(dynamicDataCallResult.result);
    if (dynamicDataCallResult.result=="success"){
        return dynamicDataCallResult.result;
    } else {
        throw new Error(dynamicDataCallResult.result)
    }
}).then(function(dynamicResult){
    return Form.submit(item.properties.submissionMethod, item.properties.legacyMethod);
}).then(function(submitResult){
    Form.fireRule("onSuccess", item, data, response);
}).catch(function(err){
    console.log(err);
    Form.fireRule("onFailure", item, data, err);
    Form.showDialog("errordialog");
});
```

Debugging

Unknown macro: 'redirect'

- Advanced Debugging of Maestro Forms

Advanced Debugging of Maestro Forms

Unknown macro: 'redirect'

Occasionally while developing Maestro forms you may see a behavior that doesn't seem right and you can't figure out why. This usually occurs in relation to rules that you've created using the script editor.

The following sections describe some advanced debugging techniques that you can use to find out what is going on in your script and in the form data.

Topics covered:

This article is focused on functional aspects of the form - debugging of CSS and styling issues is not covered.

Recommended Skill Set

Debugging Maestro forms in the browser is a technical task and requires some understanding of web application development in general, specifically the web standards of HTML, CSS and JavaScript.

Additionally, forms produced by Maestro run on AngularJS. AngularJS is a JavaScript based open-source front-end web application framework backed by Google. There is a very strong community supporting this technology and while it is not required that you are familiar with AngularJS, it would be a benefit for you to have some basic knowledge of the framework. We would recommend you review the tutorials to develop your understanding of this technology:

- AngularJS Tutorial

Debugging Tools

Before getting started, ensure you have appropriate tools for debugging JavaScript in the browser. These include:

1. **Modern Browser**
   Most modern browsers come with good developer tools these days. We prefer to use Google Chrome, but Firefox and Microsoft Edge can also be used. This article assumes you are using Google Chrome which we recommend.

2. **Developer Tools Enabled**
   All these browsers come with developer tools out of the box and allow you to install plugins/extensions to provide additional capabilities. There are a number of ways to enable these developer tools but universally, hitting the **F12** key while viewing a page will bring up the built-in developer tools.

3. **AngularJS Browser Plugin**
   Maestro forms are built on the AngularJS framework and plugins are available to provide specific support for AngularJS web applications. In Chrome, we use AngularJS Batarang, Firefox also has AngularJS plugins that may provide the same capability.

Maestro Form Architecture

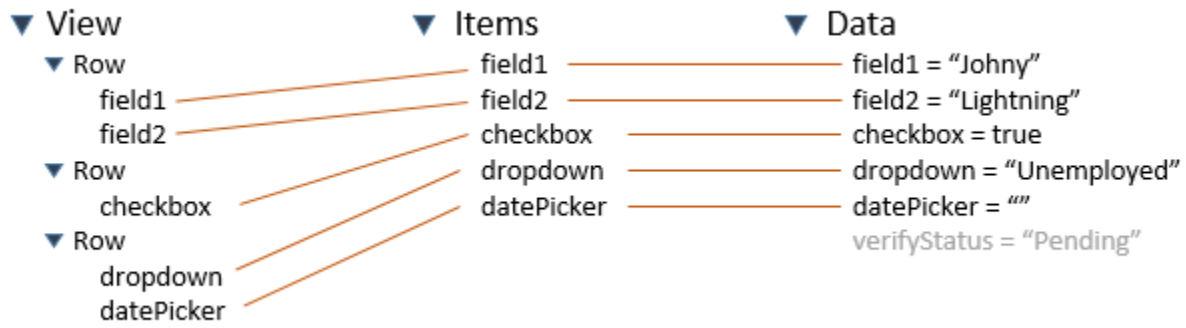
The architecture of a form consists of 3 main elements:

1. **Form View**
   A hierarchical structure that defines the visual layout of the form.

2. **Form Items**
   Each component (e.g. Field, Button) or container (e.g. Block, Section) within the form structure is considered an **item**. Form items is a list containing every item in the form.

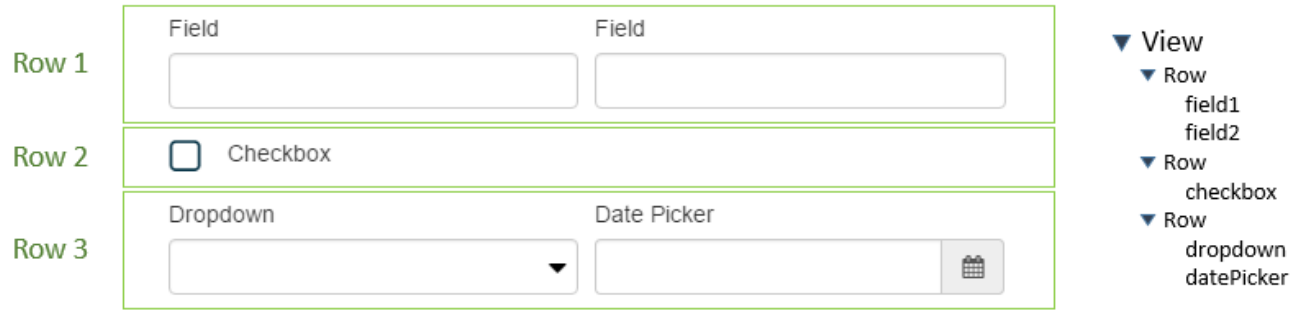
3. **Form Data**
   Think of form data simply as a map of key / value pairs. Most data elements are linked to a form item (e.g. a field) but data elements can also exist autonomously.

These 3 elements are all closely related. Form items appear in the Form View and have Data elements associated with them:



Form View

The form view is a hierarchical structure that consists primarily of horizontal rows, each containing one or more components. The hierarchical structure represents the layout of the form and all components within it.



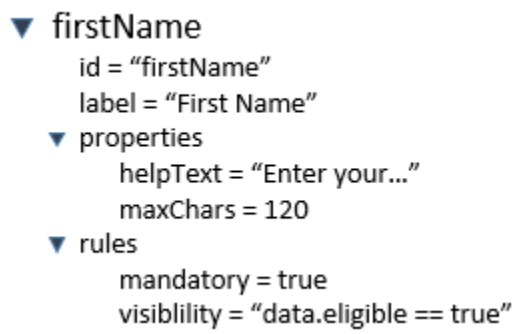
Being hierarchical, rows in the view can be containers for lists of more rows in a recursive manner.

Form Items

Each form item in Maestro is uniquely identified by a field Id. Maestro generates these field Ids based on the name of the field but also allows you to override the generated value and manually specify a field Id so long as they are unique in the context of the form.

Having guaranteed unique field Ids for all items within a form allows us to have a flat list (array) representation containing all items in the form, making access to these items much simpler than having to navigate a view hierarchy to find them.

Items can contain UI elements, but also have properties and rules associated with them. By way of demonstration, consider a mandatory first name field that has a maximum length of 120 characters and a visibility rule:



Form Data

All data in the form is stored in a single object called the Form Data object. Like form items, each form data element must have an id that is unique in the context of the form. If a data element exists with the same Id as a form item, that form item will read and write its data to that data element. Data elements can exist without being bound to a form item.

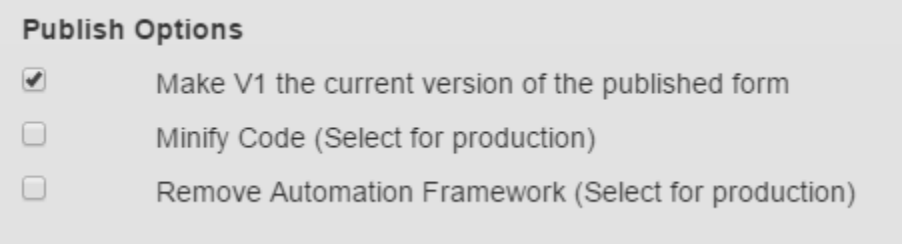
Depicted below is 4 data elements, 3 of which are bound to fields (items) and one that is un-bound (verifyStatus). The middleName data element is blank suggesting that nothing has been entered into the middleName field.

Data elements that are not bound to form items (verifyStatus in this example) will not be Saved/Submitted with the form data and so only exist for the duration of the session.

Note that the **Data Field** component type can be used to bind data elements without having to display them in the visual layout so that they get persisted with the form data.

Publishing Forms for Debug

When you publish a Maestro form there are a number of options available to you.



Some of these are very relevant to debugging, these are:

Publish Option	Description
Minify Code	**Deselect** this option. Code minification is the process of compressing code to reduce the size and therefore load time, but does not affect the operation of the code. The process removes unnecessary characters from the code,white space, new line, comments etc... The result of minification is a single line of code that is very long and difficult to read and debug so ensure that this option is deselected for debugging.
Remove Automation Framework	**Deselect** this option. The automation framework built into Maestro forms facilitates UI driven automated tests such as those that would be run from test tools like Selenium. This framework also contains some features that will assist in debugging JavaScript so it is handy to keep the automation framework for debugging.

Activating a Debug Session

in order to start a debugging session you should:

1. Load the published Maestro form in a modern browser of your choice
    a. You must access the published version of the form, **DO NOT** try and debug a form in Preview mode in the Maestro designer.

b. If you are intending to debug JavaScript (e.g. Maestro rules) then ensure you have deselected the **Minify Code** and **Remove Automation Framework** publish options as described above.

2. Once the published form is loaded in the browser you can activate the browser developer tools by hitting the **F12** key.

> **Shortcuts for Republishing Forms**
>
> While debugging forms you may need to republish your form with changes multiple times and you need to refresh the page with your published form on it each time. The following keyboard shortcuts are available for you to use to streamline this process:
>
> - CTRL-P: When used from the Maestro Design page will bring up the Publish dialog. (Command-P on Mac)
> - CTRL-ENTER: Will trigger the publish function. Remember to deselect the Minify Code publish option first.
> - CTRL-TAB: This will switch you to the last active browser tab which in most cases will be the published form.
> - CTRL-R: Will refresh the currently active browser window.
>
> Refreshing the page with the republished form on it should leave the browser developer tools active so you should not need to reactivate them each time.

Inspecting the Form Object

In the Maestro framework, the Form object contains everything else you might need to access for debugging including the View, Items and Data (see Maestro Form Architecture), so to access any of these elements you need to scope the Form object. There are a number of ways to do this from the **Console** tab in the browser developer tools:

- Using JQuery you can scope the form object in this manner:

```
$("Form").scope().Form
```

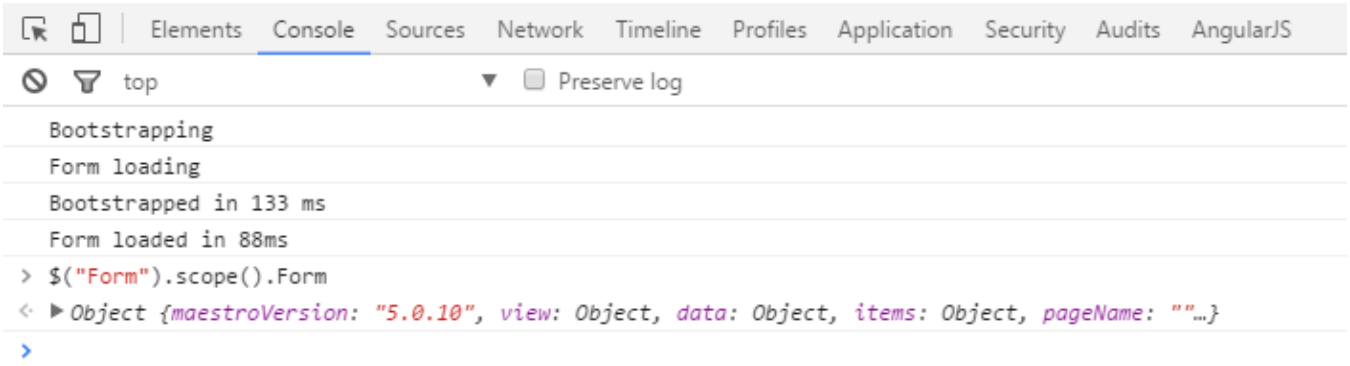
- If you have included the Automation Framework (see Publishing Options) you can short cut this access by using the **maestro** root element included with that framework:

```
maestro.Form
```

- If you have AngularJS Batarang (or similar) installed, they may also allow you to access the Form object by way of the $scope object (You need to select an element in the Elements tab first - e.g. body or form):

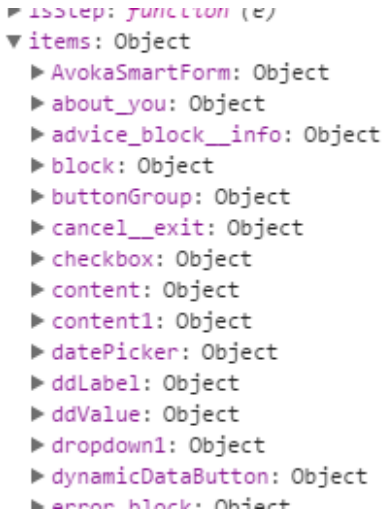
```
$scope.Form
```

Enter one of these directives into the Command prompt in the Console tab and press Enter to scope the Form object:



Opening up this object you will see a long list of sub elements but among those you will find the 3 key elements to the Maestro Form Architecture, the Form Data object,

the Form Items,



and the Form View.

Of course, if you have a JavaScript debugging session active (as described below) you will have access to the Form Data object in the function scope and can readily inspect its contents.

Logging an Item to the Console

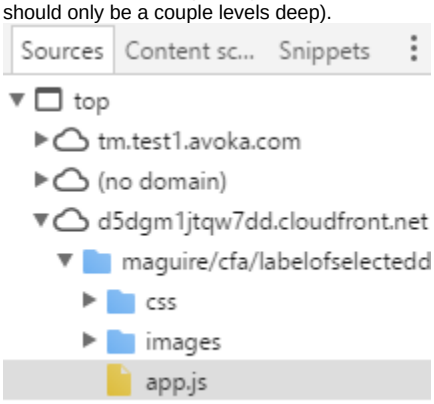
There is a handy Util function to generate a printable representation of an item and log it to the console as follows:

```
$scope.Util.logItem($scope.Form.items.personalDetailsBlock)

"personalDetailsBlock
 firstName
 middleNames
 lastName
 dateOfBirth
  day
  month
  year
  date
  daysData
 emailAddress
 mobileNumber
"
```

Locating a JavaScript Rule for Debug

All rules in Maestro are implemented as JavaScript so to locate the logic implemented in a Maestro rule you must find the JavaScript code that implements that rule. You can do this as follows:

1. In the developer tools window select the tab that lists the page source files (in Chrome this is titled **Sources**).
2. Open the **app.js** file by either using CTRL-O (CMD-O on Mac) and typing in the name of the file, or by searching in for it the file sources hierarchy then double-click to open it (the location of this file will vary depending on your installation and the existence of a CDN in your environment, but it should only be a couple levels deep).



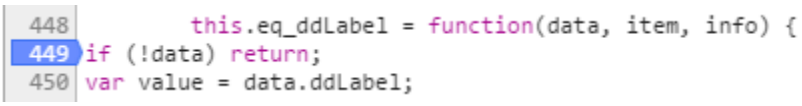
3. In the app.js file, search for and locate the JavaScript function that contains your rule in the app.js file. There are several ways to do this depending on the capabilities of the browser dev tools, but most commonly you will search by a code snippet or by the function name:

    a. To find the function using a code snippet, select a snippet of code from your rule script by clicking in the app.js file and using CTRL-F to activate the find feature. You may need to review your rule script in the Maestro designer and select a snippet that is likely to be unique. For example, below I have searched for the snippet *Form.getItemFromPath('data.buttonGroup')* and located the function **eq_ddLabel** that contains my rule script:

```
447
448         this.eq_ddLabel = function(data, item, info) {
449 if (!data) return;
450 var value = data.ddLabel;
451
452 var item = Form.getItemFromPath('data.buttonGroup');
453 var selectionObj = Util.find(item.properties.options, "value", data.buttonGroup);
454
455 return (selectionObj && selectionObj.label) || "";
456 };
457
```

    b. To search by JavaScript function name you will need to know the naming conventions for these functions (see the section on JavaScript Function Naming below). If you know the function name you can either search for the function name as a snippet (as above) or if your dev tools support it you can use the JavaScript Member search capabilities - in Chrome this can be activated using **CTRL-SHIFT-O:**

4. To debug your rule, set a break-point on one of the lines before the code you want to step through by clicking in the left column area and confirming the break-point indicator:



5. Now you are ready to debug the rule. You can return to your form window and perform the action that will trigger the rule to execute and step through your rule script using the standard JavaScript debug functions.

> A word of caution against modifying JavaScript in-line in the browser debug tools. Results of this sort of change are often unreliable. While it is less convenient to republish a form each time you want to change a rule - this is the recommended approach.

Rule Types and JavaScript Function Naming

Every rule that you put into your Maestro form will have a corresponding JavaScript function in the app.js file. The name of this function will be generated based on a standard naming convention that consists of a rule type code and the field ID with an underscore separator as follows:

```
<rule-type-code>_<field-id>
```

The following codes represent the types of rules used by the Maestro foundation widgets:

Rule Type Code	Description
sh	Short for **sh**ow, this is the code for a **visibility** rule that controls whether the field is presented on screen. The result of this rule should be a boolean value where true = visible, false = read-only.
us	Short for **us**able, this is the code for an **editability** rule that controls the read-only status of a field. The result of this rule should be a boolean value where true = editable, false = read-only.
md	Short for mandatory, this is the code for a **mandatory if** rule that controls the required state of an item. The result of this rule should be a boolean value where true = mandatory, false = optional.
ok	Interpreted as **OK**, this is the code for a **validation** rule that controls the error state of the field. The result of this rule should be a string that is either blank (for valid values) or contains the error message (for invalid values).
chok	Interpreted as Change OK, this is the code for a **validate after change** rule that augments the standard validation rule to provide support for dynamic data validations. The result of this rule can either contain the error message (like the standard validation rule) or a promise.
eq	Short for **eq**uals, this is the code for a **calculation** rule. The result of this rule should be the calculated value of the field. Warning: In Maestro, calculation rules are not triggered if the field they belong to is not visible.
click	A **click rule** is an action rule triggered by clicking on a clickable UI item.
blur	A **blur rule** is an action rule triggered when the focus leaves a field.
focus	A **focus rule** is an action rule triggered when a field acquires the focus in the browser window.
change	A **change rule** is an action rule triggered when the value of a field changes.
dc	An acronym for **d**ynamic **c**lass, this rule can be used to apply a css class to a field dynamically based on logic.
load	An **action rule** that is triggered at the time the form is loaded.

presubmit	An **action rule** that is triggered prior to form submit.
postsubmit	An **action rule** that is triggered after the form submit event.
onSuccess	This rule type is specific to the Dynamic Data Button and is an action rule that is executed upon successful completion of the dynamic data function.
onFailure	This rule type is specific to the Dynamic Data Button and is an action rule that is executed upon failure from a dynamic data call.

Note: widget developers can create additional rule types for their widgets.

Examples

Function Name	Rule
sh_previousAddress	A visibility rule on the previousAddress block
eq_totalIncome	A calculation rule on the totalIncome currency field
ok_emailAddress	A validation rule on the emailAddress field
click_verifyIdentity	A click rule on the verifyIdentity button
load_AvokaSmartForm	A load rule on the form

Misc

Unknown macro: 'redirect'

- How to detect if a user is on a mobile device
- Loading External Javascript Libraries
- Promises in Maestro
- Translating Maestro built-in messages to another language

How to detect if a user is on a mobile device

Unknown macro: 'redirect'

When building adaptive UX in Maestro you may run into a requirement to present different content to users depending on whether they are on a mobile or desktop client. For example, it is unlikely that desktop users will utilize their camera to snap a photo of their driver's license so you may want to down-play this capability for desktop users and promote it for mobile users.

Unfortunately there is no standard way to determine if a user is on a mobile device and a Google search will uncover strong debate over the best way to achieve this. This example provides one simple option that searches the user agent string for a set of known mobile/tablet identifiers.

Compatibility

Module	Maestro
Since	5.0
Deprecated	

Example 'Is Mobile' Detect

The following JavaScript snippet demonstrates a simple example on how to detect if the user is on a mobile (or tablet) device. This will not give 100% coverage but should cover most of the commonly encountered devices.

```
var isMobile = navigator.userAgent.match(
        /(iPhone|iPod|iPad|Android|webOS|BlackBerry|IEMobile|Opera Mini)/i)
```

You could add this code directly into your rule scripts, or create a reusable JavaScript library that you can distribute among your teams.

For more fine-grained detects you may want to review the Modernizr library which provides a comprehensive analysis of the HTML, CSS and JavaScript features the user's browser has to offer:

- https://modernizr.com/

Using Modernizr in Maestro is pretty easy:

1. Build your custom modernizr-custom.js file on their website and download it.
2. Add a JavaScript library to your form and upload your Modernizr JS file.
3. Now in your rule scripts you can use the Modernizr detects, e.g.:
   if (Modernizr.touchevents)

References

- http://stackoverflow.com/questions/3514784/what-is-the-best-way-to-detect-a-mobile-device-in-jquery
- https://www.abeautifulsite.net/detecting-mobile-devices-with-javascript

Related articles

- Enforcing required configuration properties in custom native components
- Loading External Javascript Libraries
- How to detect if a user is on a mobile device
- Advanced Debugging of Maestro Forms
- Creating native components for Maestro (Eclipse/ IntelliJ Developers)

Loading External Javascript Libraries

Unknown macro: 'redirect'

Compatibility

Since	
Deprecated	

There are several ways to include or load external Javascript libraries for use in a Maestro form. They include the following:

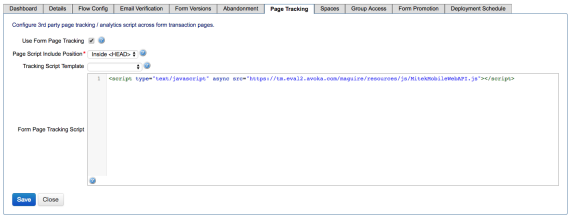
1. Using the JavaScript Library widget in the Maestro palette. This feature allows the developer to Upload a library in Maestro. Note that these libraries will get loaded on Form Load and the source will be included in the app.js file.
2. Using the Page Tracking feature of a form configuration in Transact Manager.
3. Lazy loading the libraries using jQuery.

This article covers the last two approaches in the list above.

Using the Page Tracking feature in Transact Manager

Although the feature is called Page Tracking, it can also be used to add custom Javascript and/or HTML content into the application.

1. Navigate to the Page Tracking panel for your form configuration.
2. Toggle Use Form Page Tracking to ON.
3. Set the Page Script Include Position to Inside <HEAD>.
4. Add the Javascript library reference. Example show below.
5. Note that the async attribute can be used to allow asynchronous running of the script. This technique generally allows the Maestro form to load and render faster. Some Javascript libraries cannot be used this way.

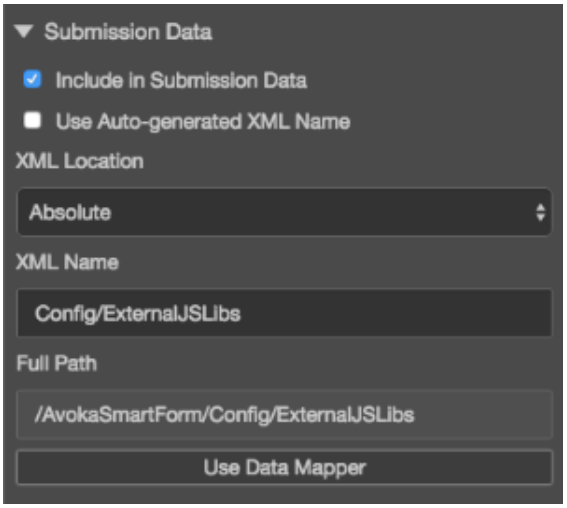


Loading libraries with jQuery

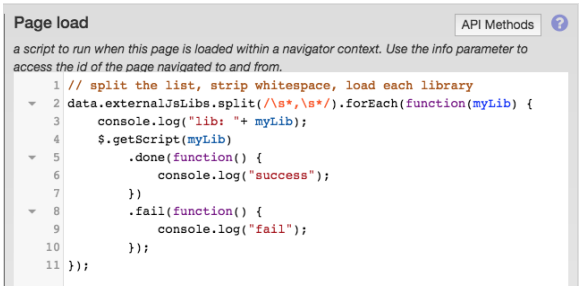
One benefit of loading Javascript libraries with jQuery, the developer has complete control over when the library is loaded. Specifically, the library can be loaded when it is required. Libraries that are related to a feature that is not used in the form do not ever need to be loaded. This obviously decreases the size of the Maestro application footprint.

Another benefit of using this approach is that the path and names of the Javascript libraries can be environment-specific. Many supporting libraries have different production and non-production URLs.

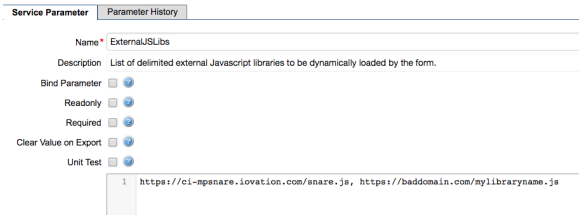
1. Add a data field into the page content of your Maestro form.
2. Go to the Data tab and toggle Include in Submission Data to ON.
3. Set the XML Location and the XML Name. Example shown below.



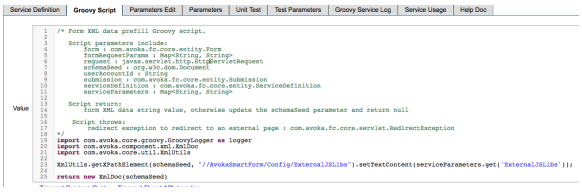
4. Create a rule to dynamically load external Javascript libraries. An example would be a Page load rule shown below.



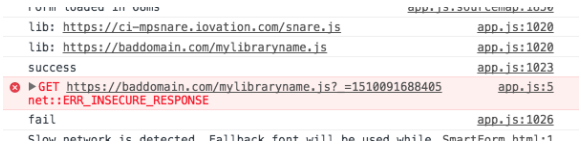
5. Note that loading libraries with jQuery provides success and failure handlers. Success handler can be used to invoke a function in the loaded script. Failure can be used to notify the user.
6. Log in to Transact Manager and create a Form Prefill service.
7. Create a service parameter called ExternalJSLibs. Set its Type to String, and provide a delimited list of Javascript resources. Example below.



8. Write a Groovy script that reads this service parameter and writes it into the Form XML location identified in step 3 above.



9. Save the script. Associate this Prefill Service with the Maestro form configuration in Transact Manager.
10. Render the form. Inspect the Console and the Network for the libraries loaded with Page Tracking and jQuery.



11. All done. Note the success and fail messages in the console. One library loaded; the other did not.

Related articles

- How to create and share a library V4
- Enforcing required configuration properties in custom native components
- Loading External Javascript Libraries
- How to detect if a user is on a mobile device
- Advanced Debugging of Maestro Forms

Promises in Maestro

Unknown macro: 'redirect'

Maestro is used to create user experiences that often interact with the Transact server. Whenever you invoke a remote service, it runs asynchronously - meaning that it will take time to execute, and you will be notified when (or in some cases, if) it responds. Occasionally internal methods within Maestro also execute asynchronously.

An interesting question is how do we structure our code?

1. We need to make sure that we call the invocations in the right order.
2. We need to write our code in such a way that it's easy to understand what is going on.

The basic mechanism for handling asynchronous events is to use callback functions. This is a function that is called when (or if) the asynchronous invocation returns. This works, but it leads to some difficult to understand code. You can read this article if you want more information about callbacks and why they cause problems. http://callbackhell.com/ And you can read this article if you want to find out a bit more about functions in JavaScript: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

A more modern approach is to use a promise. Maestro uses promises internally. There are lots of articles available if you want to understand more about promises. I like this one: https://coligo.io/javascript-promises-plain-simple/ This article does show how to create a new promise, but generally, you don't need to in Maestro, because asynchronous calls already return promises - you just need to know how to chain them. All you need to do is wrap your invocations inside a series of "then(...) method calls.

But even once I'd read that article, I didn't really understand exactly how to use promises. I found this article to really get down to the essence of what you practically should and shouldn't do, and provides one simple trick to help you understand promises.

https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html

Here is my summary of what you should do.

Chaining

You should always use chaining with your promises. Nesting your promises will result in the code that is very difficult to read and understand, resulting in the "Pyramid of Doom".

Your code should look like this:

```
invoke1(...).then(function (resultOfInvoke1) {
  return invoke2(...);
}).then(function (resultOfInvoke2) {
  return invoke3(...);
}).then(function (resultOfInvoke3) {
  return invoke4(...);
}).catch(function (err) {
  console.log(err);
});
```

Return a value

Each function should return a value. This is the key takeaway from the above article.

You can return 3 different things:

- Another promise. The "then()" clause will execute once the promise is resolved.
- A synchronous value. The "then()" clause will execute immediately the function returns.
- Throw an error, as shown below.

```
throw new Error('Something Bad'); // throwing a synchronous error!
```

Be careful of just invoking an invocation rather than returning its value. See Rookie Mistake #5 in the article.

Important: the parameter of the next function will be the result of the previous one. Or put the other way - the result that you return in the current function will be the input parameter of the next one. This is shown clearly in the naming convention above - the function is called "invoke2", and the next function takes as its parameter "resultOfInvoke2". Using this naming convention really helps, because promises are a little un-obvious unless you're familiar them. Using the naming convention makes it easy for the developer who comes after you.

Extra Steps

If you need to add an extra step into the sequence, you just need to copy and paste these two lines:

```
}).then(function (resultOfInvoke2) {
  return invoke3(...);
```

Catch exceptions

Always have a catch at the end. You never know when something unexpected will happen.

See also

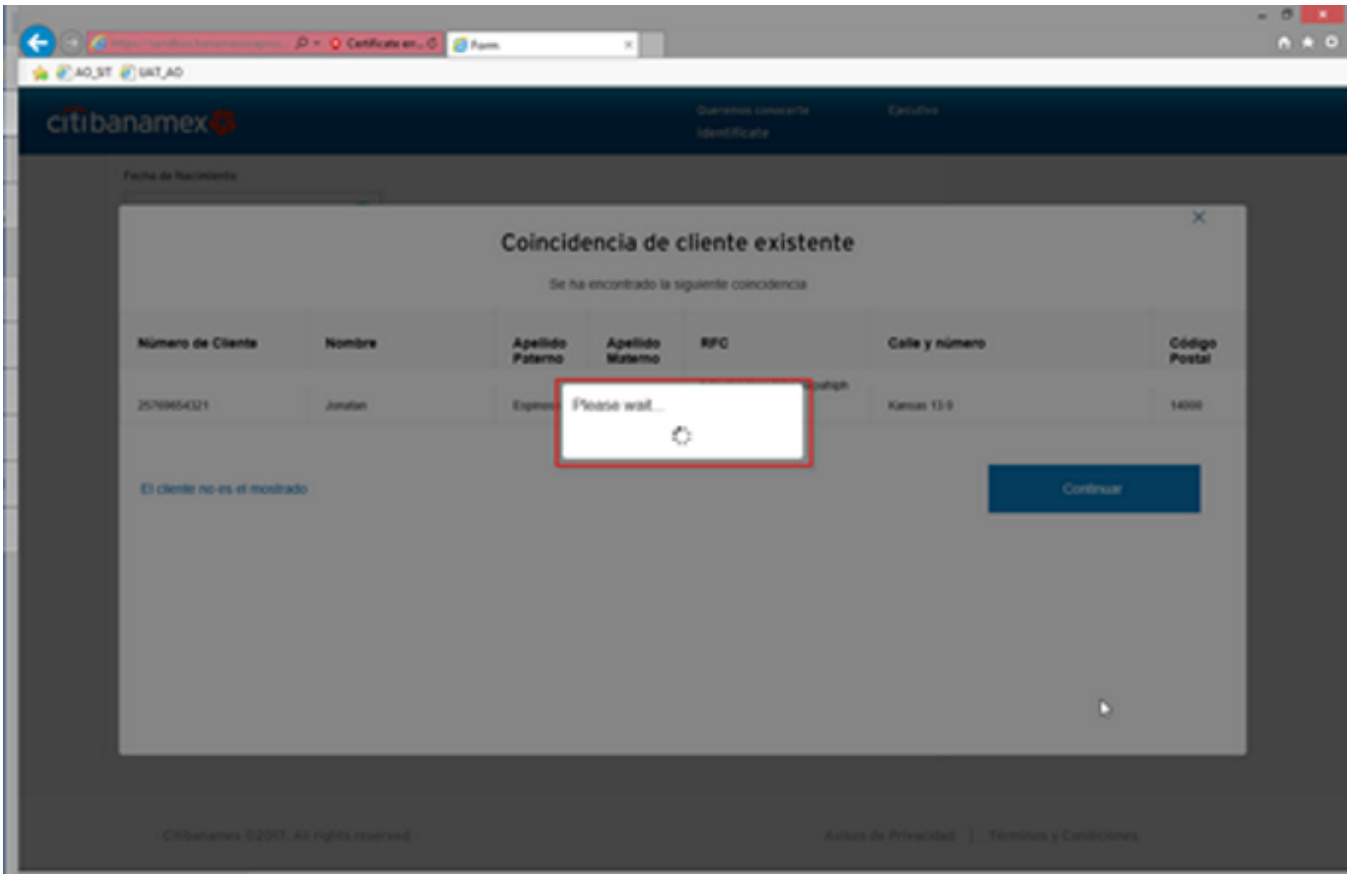
For an example of promises being using in Maestro, see this example: Implementing a pre-submit validation

Translating Maestro built-in messages to another language

Unknown macro: 'redirect'

Most of your Maestro form is built by you, and you can build it in the language of your choice. Maestro also has capabilities to author a single form in multiple languages.

But some parts of a Maestro form are built-in. One such example is the "Please wait..." message shown below:



In order to modify these built-in messages to a language other than English, you need to use the translation capabilities in Maestro.

That loading message is activated via a Form.showProgress call. e.g. Form.showProgress("Loading...")

You can right click the string in the rule and turn it into a translatable key. For the save, cancel and submit calls you can change this value on the root form item and it would also be a translatable key.

Native Components

Unknown macro: 'redirect'

- Creating native components for Maestro (Eclipse/ IntelliJ Developers)
- Enforcing required configuration properties in custom native components

Creating native components for Maestro (Eclipse/ IntelliJ Developers)

Unknown macro: 'redirect'

Compatibility

Since	5.1.2 (Maestro)
Deprecated	

Introduction

This guide is for Eclipse or IntelliJ developers who would like to create a native component library for Maestro.

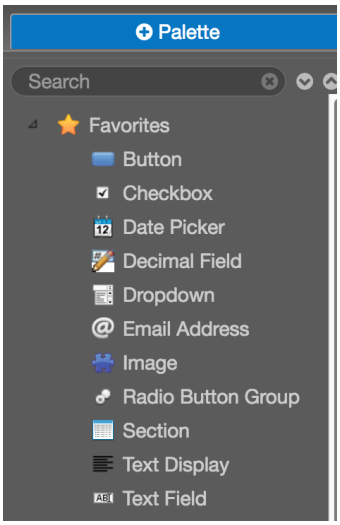
Pre-requisites

It is assumed that the audience already knows:

a. How to use an IDE such as Eclipse or IntelliJ
b. Is familiar with Maestro, related features and terminology
c. Can navigate around HTML, CSS and Javascript

Background

Maestro comes with a rich set of components that are contained in core libraries assigned at the organisation and project level. What this means is that when you create forms, your palette is complete with all of the usual features that you would expect to find in web development such as text fields, radios etc.



There may however be scenarios in which components do not perform the function you need and it is impossible to override the behaviour of an existing component in order to get the result you need.

In this case you may need a **native component and this document deals with the creation**

Current Material

There are three primary documents found across Confluence and in the knowledge base (see ***Related Articles***) which touch on the parts of the process, but none which I think attack this from a viewpoint of assembling a basic component and understanding the processes involved in a step by step fashion within an IDE. This guide will cover this and enable the developers to get up and running quickly.

Scope

This document will provide details on how to create components which:

a. Are natively built outside of Maestro
b. Which enable best use of the Angular framework that's provided within Maestro AND
c. Which tap into the core benefits of Angular to provide rich two-way data binding

This document does not deal with the rules and regulations around component creation.

Nor does it dive into details which are covered in other guides - it is intended to get development teams up and running.

Step-by-step guide

Exercise A - Custom button

In this exercise we are going to create a button with some custom styling and add it to a library in Maestro.

Step 1 - Eclipse Setup

Create a new Web Project in Eclipse with the following directories

`buildfiles >`

`<Name>Library >`

`<Name>Library > widgets >`

`zip >`

`Where <Name> is the name of the library`

Step 2 - Create the ANT build file

Now create the ANT build file in buildfiles, and call it build<Name>.xml

Use this source and make sure that the project-name and folder-to-zip are the same names as your <Name>Library >

```
<project name="maestro-components" default="zip" basedir=".">
 <property name="project-name" value="TestLibrary" />
 <property name="folder-to-zip" value="TestLibrary" />
 <property name="dir" value="${basedir}\.."/>
 <property name="target-zip-folder" value="zip" />
 <target name="clean">
  <delete file="${project-name}.zip" />
  <delete dir="${target-zip-folder}" />
 </target>
 <target name="zip" depends="clean">

  <zip destfile="/${dir}/${target-zip-folder}/${project-name}.zip" basedir="
/${dir}/${folder-to-zip}" excludes="*.DS_Store,*.project" />
 </target>
</project>
```

Note that this build file is responsible not just for zipping the correct structure, but ensuring that certain files which will cause the import to fail are excluded as well.

Step 3 - Create the custom component structure

In your <Name>Library > directory create this structure:

> *\ widgets \ <elementname>*

Step 4 - Baseline default files

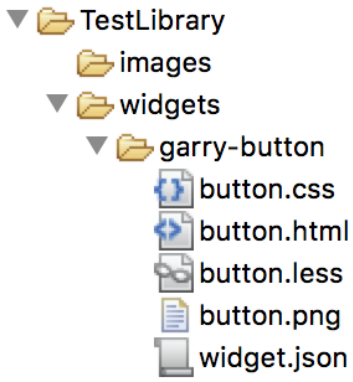
And copy in the default files from the pack resources (css, html, less, png, json) to \ widgets \ <elementname>

Copy in the library config into the root of

> *<Name>Library >*

Add in a small image that will serve as the icon in Maestro palette

You should end up with a structure similar to this:



Step 5 - Create JSON Widget configuration

Modify widget.json to ensure that it references the resources in the pack, e.g:

```
{
 "category": "Buttons",
    "label": "Garry Button",
    "html": "button.html",
    "icon": "button.png",
 "css": "button.css",
    "propDefs": {
        "icon": {
            "label": "Icon",
            "type": "icon",
            "imageStyleTarget": "Button icon",
            "value": "",
            "helpText": "Select the button icon"
        },
        "iconPos": {
            "label": "Icon Position",
            "type": "option",
```

```
            "value": "left",
            "options" : [{"label":"Left","value":"left"},
{"label":"Right","value":"right"}],
            "helpText": "Select the button icon position"
        }
    },
    "defaultStyleTarget": "wdg-button",
    "span": {"xs": 12, "sm": 12, "md": 4, "lg": 4},
    "noData": true
}
```

The .less can be ignored for this exercise.

Step 6 - Create Widget HTML

Now modify the button.html to serve as your angular component.

```
<div>
 <button type="button" class="btn wdg-button">
  <i class="wdg-button-icon av-icon-button"
     data-ng-if="item.properties.icon && item.properties.iconPos == 'left'"
     data-ng-class="item.properties.icon"></i>
  <span data-av-compile class="wdg-button-label"
     data-ng-bind-html="item.htmlLabel || item.label"></span>
  <i class="wdg-button-icon av-icon-button"
     data-ng-if="item.properties.icon && item.properties.iconPos == 'right'"
     data-ng-class="item.properties.icon"></i>
 </button>
</div>
```

Step 7 - Create Widget Style (CSS)

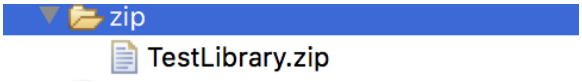
Make sure button.css is as below:

```
.wdg-button {
  width: 100%;
}
.wdg-button .wdg-button-label {
  padding-right: 4px;
}
.wdg-button .wdg-button-icon {
  display: inline-block;
  width: 28px;
  height: 28px;
  line-height: 28px;
  text-align: center;
```

```
  font-size: 18px;
}
.wdg-button .wdg-button-icon + .wdg-button-label {
  display: inline-block;
  padding: 4px 4px 0 5px;
  vertical-align: top;
}
```

Step 8 - Build the library

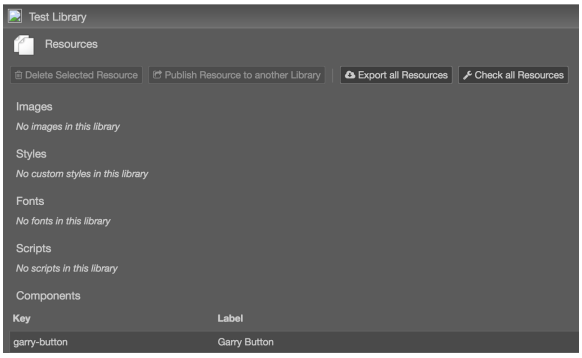
Run the ANT Build, and you should see a ZIP file created within the zip directory



Step 9 - Import the library to Maestro

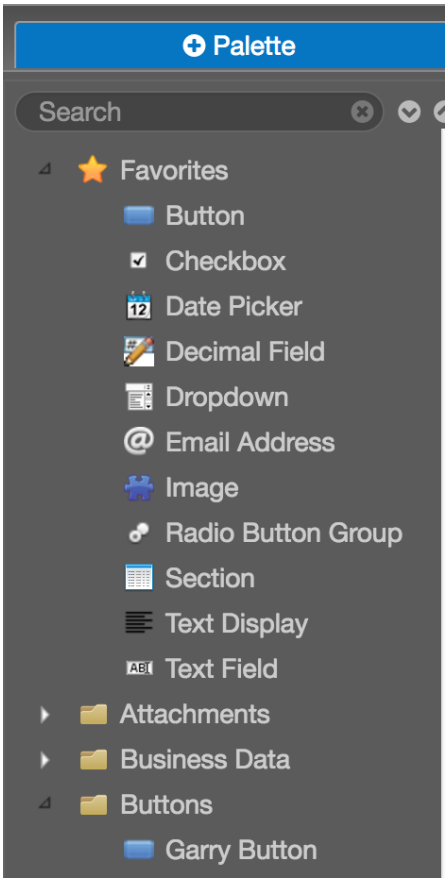
Access Maestro and go to ***Libraries***

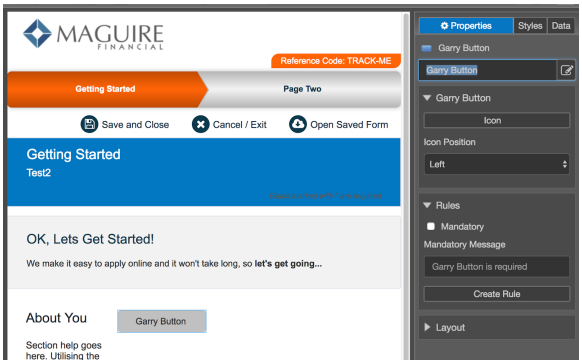
Now use the ***Import*** button and choose the ZIP file and once it's imported you should see your component listed



Step 10 - Drag onto your form

Access Maestro and drag the component onto your form





Exercise B - Angular dynamic button

In this next example, we will be providing a native component which uses angular to provide some additional functionality and data binding in the form of a count.

Follow the above exercise to step 4, but use the following ***button.html***

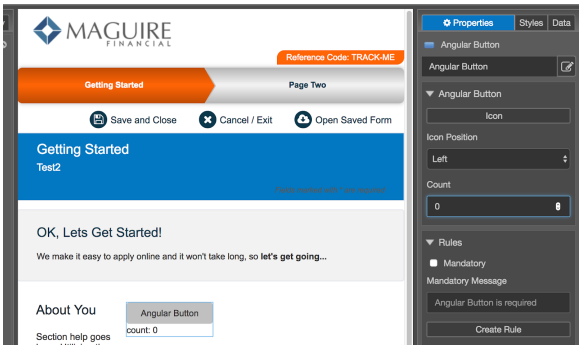
```
<button ng-click="count = count + 1" ng-init="count=item.properties.count"
class="btn wdg-button">

  {{item.label}}

</button>

<span>

  count: {{count}}

</span>
```

Modify the original ***widget.json*** and use the one below. Note that you are changing the ***category, label*** and also adding in a ***count*** parameter which is referenced by the ***button.html*** binding to ***item.properties.count*** above. Use the other files associated with exercise A.

```
{
 "category": "Buttons",
    "label": "Angular Button",
    "html": "button.html",
    "icon": "button.png",
 "css": "button.css",

    "propDefs": {
        "icon": {
            "label": "Icon",
            "type": "icon",
            "imageStyleTarget": "Button icon",
            "value": "",
            "helpText": "Select the button icon"
        },
        "iconPos": {
            "label": "Icon Position",
            "type": "option",
            "value": "left",
            "options" : [{"label":"Left","value":"left"},{"label":"Right","
value":"right"}],
            "helpText": "Select the button icon position"
        },




         "count":{
           "label": "Count",
             "type": "integer",
             "value": 0,
             "helpText":"Define the default value for the count"
         }

    },
    "defaultStyleTarget": "wdg-button",
    "span": {"xs": 12, "sm": 12, "md": 4, "lg": 4},
    "noData": true
}
```

Now build the ZIP and import to Maestro.

From the buttons category drag across the Angular Button



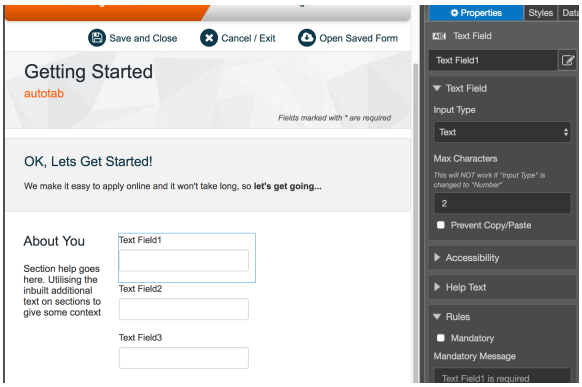
Now configure the count at 10 and preview the form:

Press the button and watch the dynamic count binding

here. Utilising

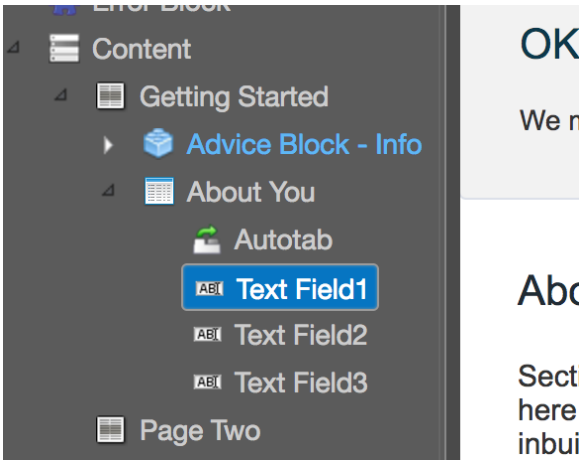
Exercise C - Auto-tabbing component (autotab.zip)

The auto-tab component allows you to ensure that the cursor automatically moves between fields in Maestro once the maximum number of characters has been reached. In the example form below, this means that after 2 characters it will tab from text field 1 to 2, then 3 for you.

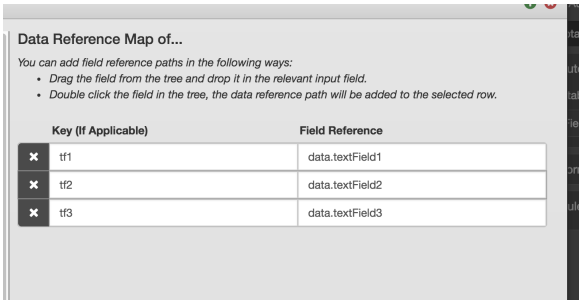


Once you have imported this component, set the max characters setting appropriately for each input field on the form.

Then drag in the **autotab** component



Now configure the auto tab component making sure that the **autotab targets** are set up with the fields in question, and that the data reference map has a field reference which correctly prefixes the data element (example below)



Pack Resources

Resource	Description
component.css	Default css file for component
component.html	Default html file for component
component.less	Additional header CSS
component.png	Default image for icon in Maestro for the component
widget.json	Configuration for the component
.library-config.xml	Default library configuration
MaestroComponents2.zip	**Contains all files as above**

Related articles

Creating Native Maestro Components

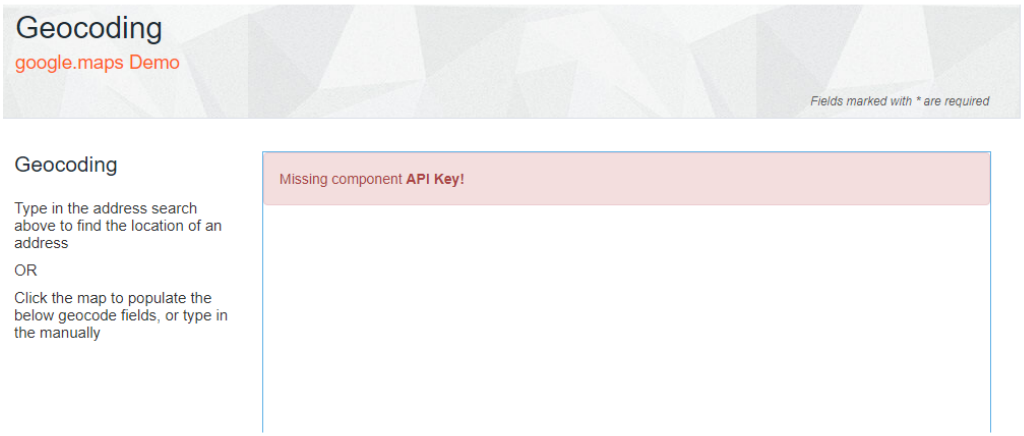
https://support.avoka.com/kb/display/MD51/Creating+Maestro+Native+Components

- Enforcing required configuration properties in custom native components
- Loading External Javascript Libraries
- How to detect if a user is on a mobile device
- Advanced Debugging of Maestro Forms
- Creating native components for Maestro (Eclipse/ IntelliJ Developers)

Enforcing required configuration properties in custom native components

Unknown macro: 'redirect'

Sometimes, as a component designer, we want to define a component property that is required to be set. Since we don't have a built-in Maestro function to mark a property as "mandatory" field, we can still show a warning message like below directly in Maestro design view to remind the form designer to set this value. "Missing component API Key!" message will disappear as soon as "Google Maps API Key" property has been set.

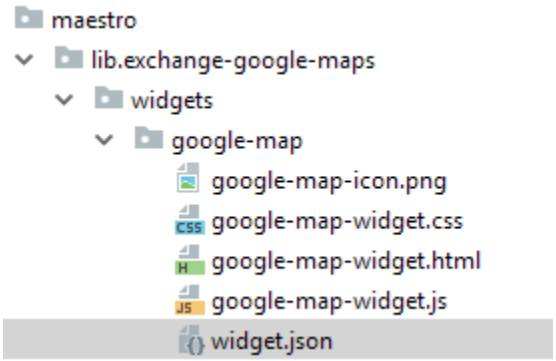


Assumption

This technique can only be used when you develop a native component.

How To

Suppose you have native component called "Google Map" which structure looks like this.



In the widget.json, it defines a property "apiKey", then what you can do is to add a in **google-map-widget.html** like below

```
<div class="av-form-field google-map" data-ng-style="{height: item.properties.height+ 'px'}"
    <div ng-if="!item.properties.apiKey" class="alert alert-danger">
        Missing component <strong>API Key!</strong>
    </div>
    <div id="google-map-widget-container">
    </div>
</div>
```

The logic is actually pretty simple. If **item.properties.apiKey** value is blank, then show this "**Missing component API Key**" message with "**alert alert-danger**" class. The ** ** will be hidden when form designer set a api key to the property.

Composer

Unknown macro: 'redirect'

Attachments

Unknown macro: 'redirect'

- Attachment Definition
- Attachment Restrictions
- Attachments in Repeats
- Attachment Time
- Customising Attachment Field Widget
- Programmatic Attachments
- Understanding the Attachment Field V4.1

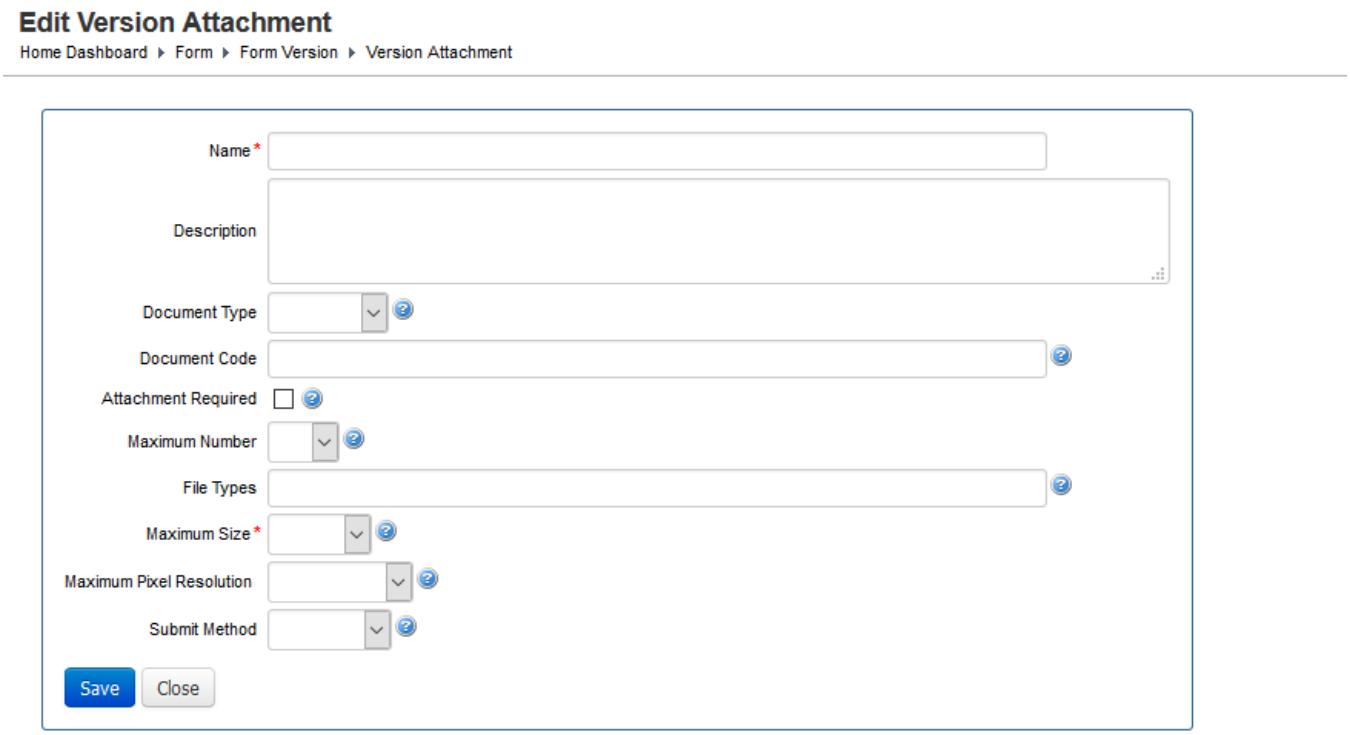
Attachment Definition

Unknown macro: 'redirect'

Attachments can be defined either in Transaction Manager, or in the form itself via Composer. Both approaches allow Attachment specific restrictions to be specified.
The following sample displays attachments defined in Transaction Manager and Composer.
Sample form: attachment form

Transaction Manager

Attachments can be defined against a Form Version.



Attachment defined in Transaction Manager

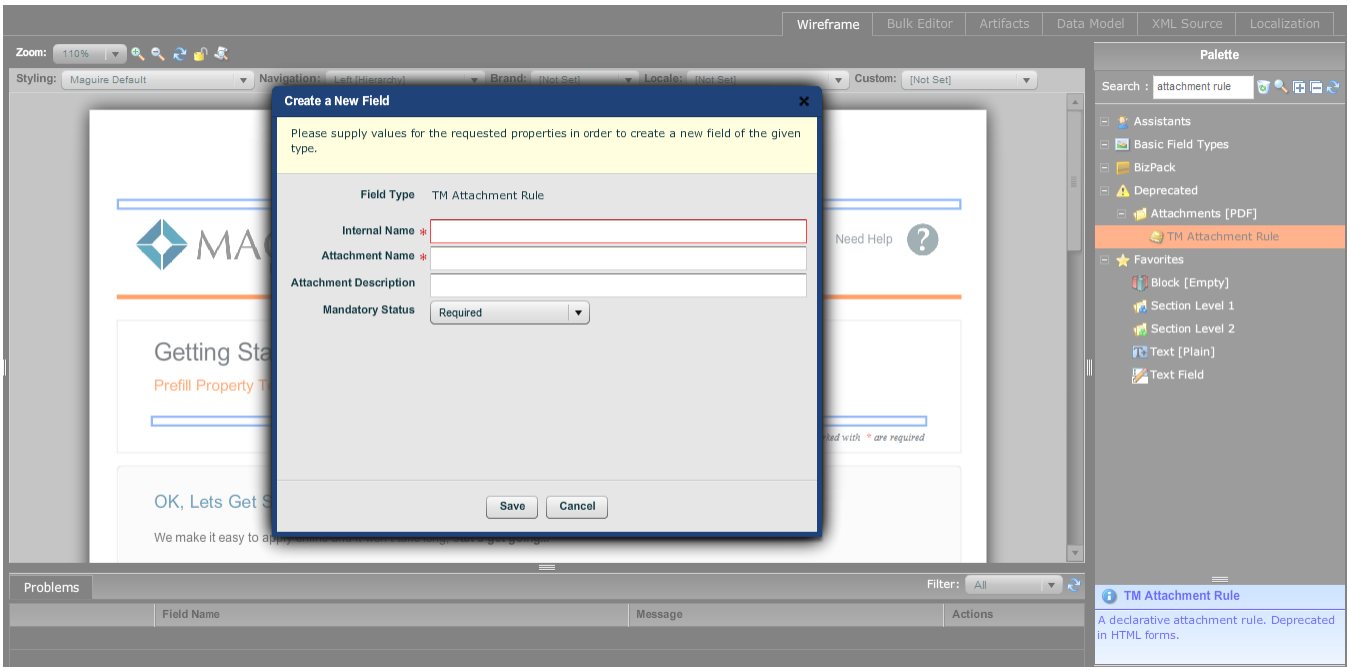
Pros

- Attachments can be modified without needing to redeploy the Form

Composer Form

TM Attachment Rule

Attachments can be added and specified by adding a TM Attachment Rule widget to a form.



Attachment Rule in Composer

Pros

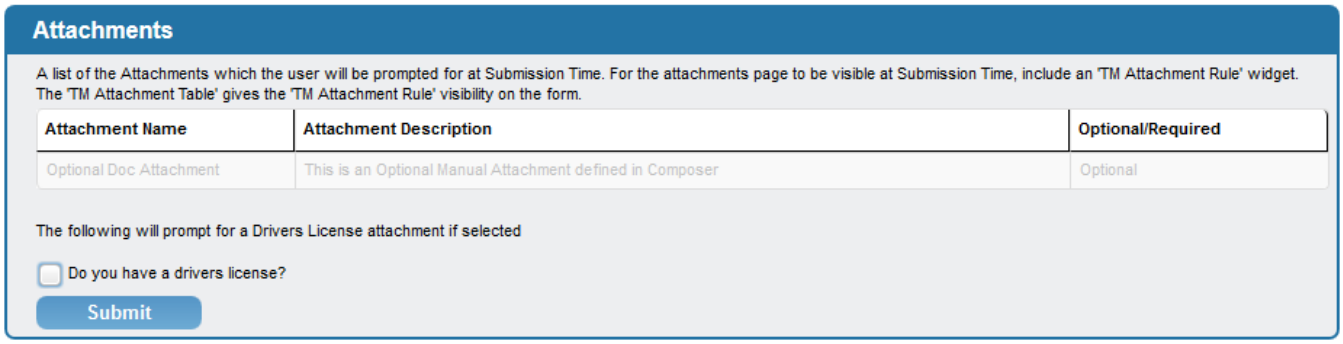
- Composer Attachments can have inclusion rules defined (So they can be dependent on other form elements/conditions)

Cons

- Attachment modifications need form redeployment

TM Attachment Table

The TM Attachment Table will display all of the Transaction Manager Attachments (Including TM Attachment Rules defined within Composer) currently applicable to the form. Any TM Attachment Rules with inclusion rules defined need to be true to be visible in this list.



TM Attachment Table rendered in a form

Related articles

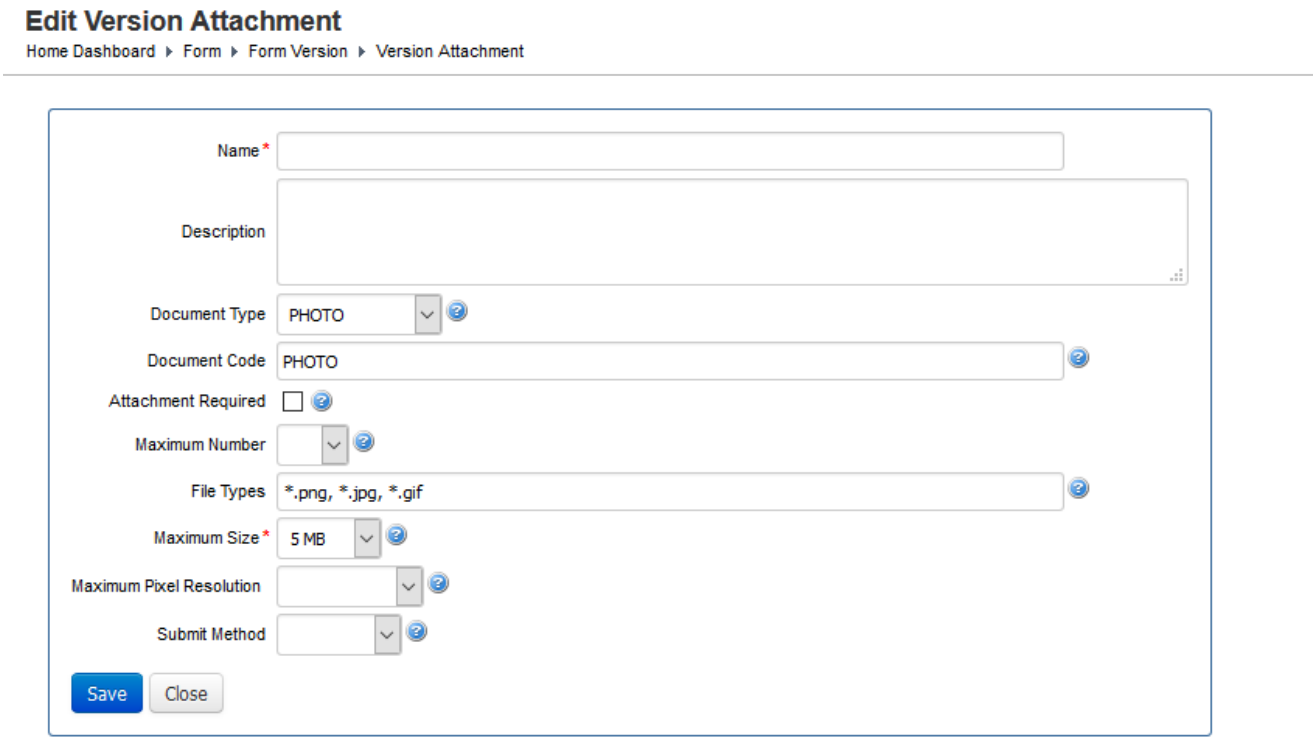
- Understanding the Attachment Field V4.1
- Attachment Definition
- Programmatic Attachments
- Attachment Time
- Customising Attachment Field Widget

Attachment Restrictions

Unknown macro: 'redirect'

Attachments configured in Transaction Manager or Composer can be defined with certain restrictions to improve data integrity or constrain the submitted document. The following restrictions can be configured:

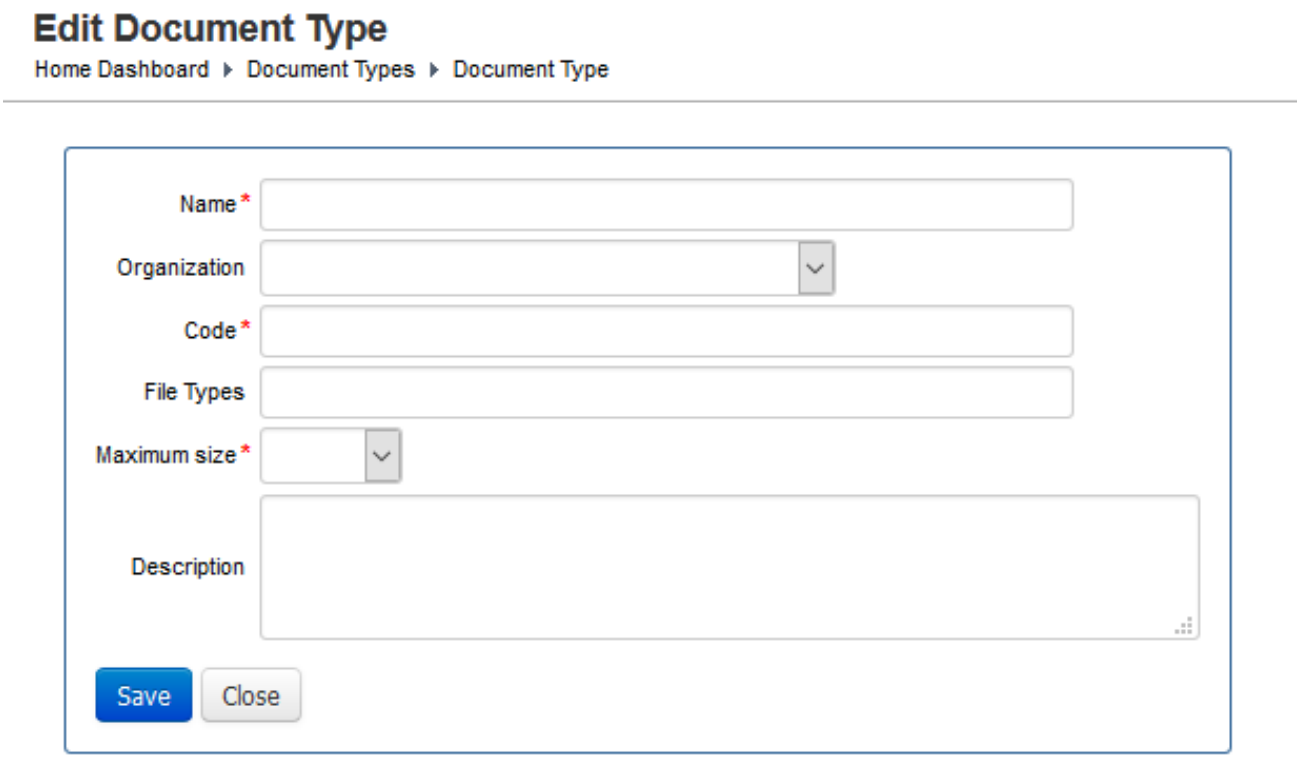
- Mandatory/Optional
- Document Type - Predefined Rules that specify File Type, Size, Description
- Document Code - Code associated with attachment
- Mime Type/File Ext - Comma separated list of file extensions
- File Size - Maximum File Size
- Pixel Resolution - Maximum Image Pixel Resolution
- Submit Method - Electronic (Upload a document) / Manual (Upload a document)



Example of an Attachment configured in Transaction Manager using a Predefined Photo Document Type which reuses its configured File Type, Size and Pixel Resolution Requirements

Transaction Manager - Document Types

Generic Document Types to be used across multiple Attachments can be predefined in Transaction Manager.
This allows file type, size, description, code to be preconfigured. Document types can also be specific to an Organisation.



Document Type Editor in Transaction Manager

Related articles

- Understanding the Attachment Field V4.1
- Attachment Definition
- Programmatic Attachments
- Attachment Time
- Customising Attachment Field Widget

Attachments in Repeats

Unknown macro: 'redirect'

Attachments in repeats or table-rows can be a little problematic, because Attachments are (by definition) global to the submission, whereas attachments in repeats are (again, by definition) repeating. The problem is that every attachment needs to have a unique name, and by default all the attachments in a repeat will have the same name.

There is a fairly easy way to work around this:

1. In the hierarchy tree, click the toolbar "gear" to show hidden fields.
2. Locate the appropriate attachment.
3. Expand it, and find the sub-field named Attachment Name. Open the properties dialog for Attachment Name.
4. You will notice that the Initial Value of this field is set to "$../{attachment.name}" - this will evaluate to the "Attachment Name" property at the Attachment Rule level.
5. What we need to do is use a Calculation Rule to make the value of this sub-field a unique value.

There are several ways to do this, but probably the easiest way is to use a calculation rule based on concatenating a string with the current repeating row index. You will need to drop a "Repeatable Block Index" field into your repeat in order to use this in a calculation.

Related articles

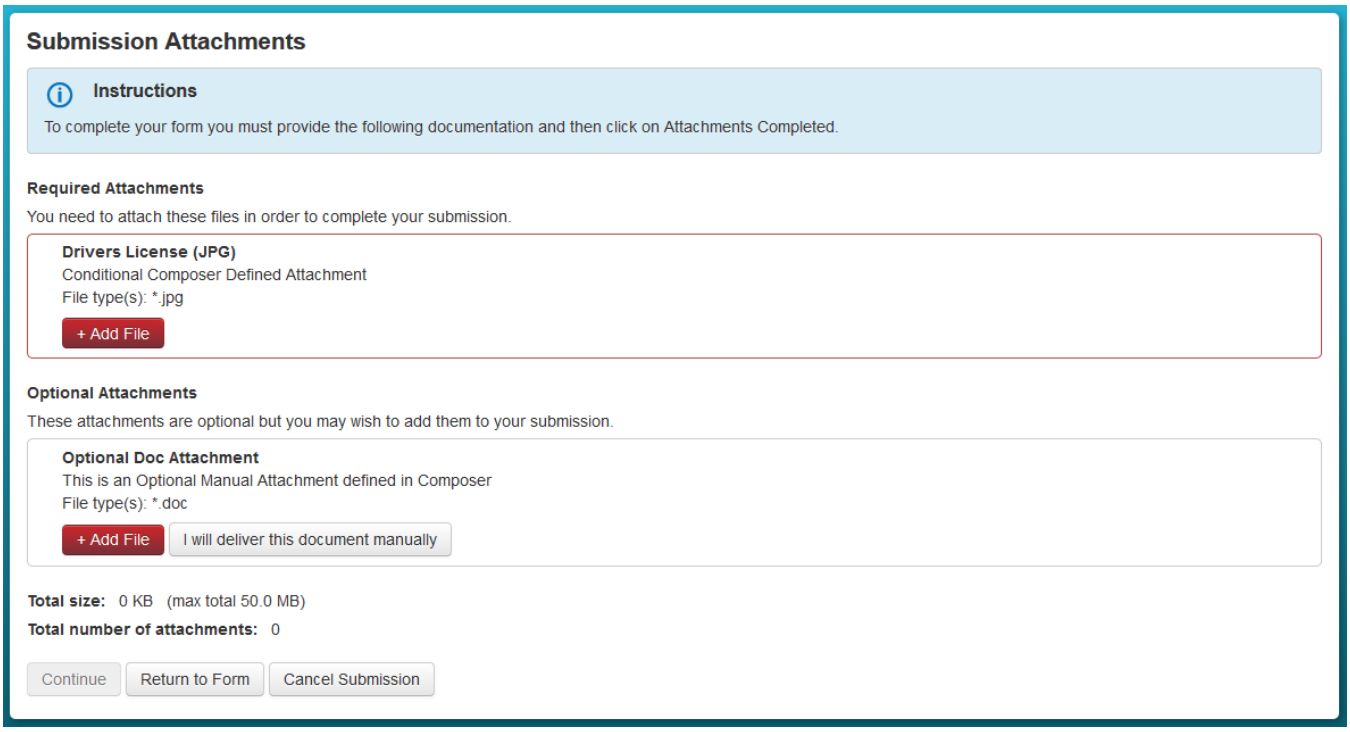
- Understanding the Attachment Field V4.1
- Attachment Definition
- Programmatic Attachments
- Attachment Time
- Customising Attachment Field Widget

Attachment Time

Unknown macro: 'redirect'

Attachments can be added at different times in the overall flow.

Attachments defined within Transaction Manager or as TM Attachment Rules in Composer will prompt the user after submission. The user will also be informed of any applicable File Type or Size restrictions applicable to each attachment.



Form Prompting User after Submission for attachments

The following sample displays TM Attachments defined in Transaction Manager and Composer. The user is prompted to provide the attachments after form submission.
Sample form: https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=attachments6

Related articles

- Understanding the Attachment Field V4.1
- Attachment Definition
- Programmatic Attachments
- Attachment Time
- Customising Attachment Field Widget

Customising Attachment Field Widget

Unknown macro: 'redirect'

Dynamic TField Attachment Source

When used in TransactField app the out-of-the-box attachment field widget allows for attachments to be added from either the devices camera or gallery storage. This setting is set in Composer during form design and allows for either one or the other. It is possible to customise a form to allow for dynamic user choice between uploading attachments from the devices camera or gallery in-form. The steps to add this functionality is outlined below.

- In your form add a **radio button group** with two **radio buttons**. Name one **camera** and the other **gallery.** Also add an **attachment field** widget to the form. The radio buttons are what the form user will use to select the source for the attachment.

> Note: It is a good idea to add a visibility rule to these radio buttons to hide them on none TField forms.

- Next you need to add some script to the **attachment field manager** to set the TField attachment source based on the status of the radio buttons.

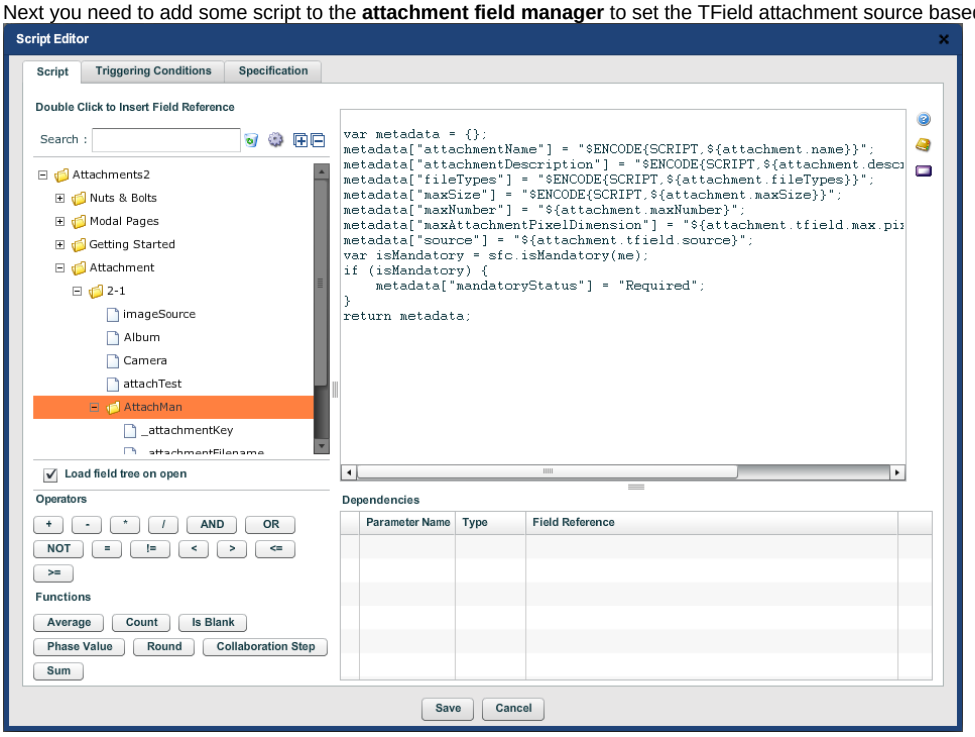


Figure: Attachment script for composer Attachment Manager Widget

This script sets the metadata properties for attachments made to **attachment field** associated with this **attachment manager**. Here we want to change the source field based on the radio buttons in the form.

- Add the following code to the **attachment manager's** attachment Script.

```
//Set the default source to camera
var source = "camera";

//Set the source based on inform data
if({Album})
    source = "album";
if({Camera})
    source = "camera";
```

And change the following line of code.

```
metadata["source"] = ~~"${attachment.tfield.source}"~~ source;
```

- The **attachment field** will now source attachments based on the inform radio buttons settings.

Creating Dynamic Unique Attachment Names

Attachment fields set the name of attachments when they are loaded into the form. This name is set in the attachment field widget properties. Sometimes it is required to have a unique name set for attachments, for example if the attachment field is contained in a repeating section all the repeated instances of the field will use the same attachment name. So it is sometimes desired to make the name unique based on information like, a timestamp or repeating index. The steps to add this functionality is outlined below.

- Similarly to the previous section Dynamic TField Attachment Source, this functionality will be added by editing the **attachment field manager** atta chment script.
- For this example we will be adding either a timestamp to the attachment name or the index of repeat the attachment field is contained in.

```
//Get the repeat manager
var im = sfc.findInstanceManager(me);
//Get the attachment name set in the attachment field widget properties
var fileName = "$ENCODE{SCRIPT,${attachment.name}}";

//If the repeat index is not null add the index to the file name
if (im != null){
    var n = im.findInstanceIndex(me);
    fileName = fileName + "_" + n;

//Otherwise add the current date/time
} else {
    var currentdate = new Date();
    var datetime = "attachName" + currentdate.getDate() + "/"
                + (currentdate.getMonth()+1) + "/"
                + currentdate.getFullYear() + " @ "
                + currentdate.getHours() + ":"
                + currentdate.getMinutes() + ":"
                + currentdate.getSeconds();
    fileName = fileName + "_" + datetime;
}
```

- Each attachment uploade will now have a its attachment name appended with either the current date or it's repeating index value.

Programmatic Attachments

Unknown macro: 'redirect'

This article provides hints and sample code to programmatically add attachments to your composer form.

- Add a Business Rule - General purpose (anywhere in the form, but preferably in a place that is easy to find)
- Set the "Additional Event Listeners" to preSubmit (Advanced property)
- Set the Business Rule to script, and use a script something like the example script below.
- In order to see the values that you can manipulate, see Nuts & Bolts / Transaction Manager Support / SFM Attachment Controller / Attachments
  - RowId must be unique. It can just be set to the instance counter.
  - DocumentCode must be unique. It should be set to something moderately meaningful.
  - All other fields are optional.
- You must add at least one standard attachment rule (otherwise the entire attachment section will be purged for efficiency reasons). Do not set any other properties other than the “internal name”.
- You can mix and match visual attachment rules with these programmatic ones.
- **Note*:*** that the attachment table will not display properly. If you want it to display properly, then you may want the same code to be on a button that allows you to preview the table manually.

Example script:

```
if (evt.evtName != 'preSubmit') {
return;
}

var im = {Attachments}; // im is of type InstanceManager. Point this to Attachments under Nuts&Bolts/ Transaction Manager Support / SFM Attachment Controller
// Repeat everything below here for each attachment
// You can also have some conditional logic to decide whether to add this one or not
// Attachment 1
var newRow = im.addInstance(); // newRow is of type NodeInfo
var count = im.getInstanceCount();

var nodeArr = [
{nodename:'DocumentCode',value:'SomeDocCode'},
{nodename:'RowId',value:count},
{nodename:'AttachmentName',value:'An Attachment'},
{nodename:'AttachmentDescription',value:'Please add a pretty attachment'},
{nodename:'MandatoryStatus',value:'Optional'},
{nodename:'SubmitMethod',value:'Electronic'}
];
sfc.updateChildNodeInformation(newRow,nodeArr);
// Attachment 2
newRow = im.addInstance(); // newRow is of type NodeInfo
count = im.getInstanceCount();

nodeArr = [
{nodename:'DocumentCode',value:'AnotherDocCode'},
{nodename:'RowId',value:count},
{nodename:'AttachmentName',value:'Another Attachment'},
{nodename:'AttachmentDescription',value:'Please add another attachment'},
{nodename:'MandatoryStatus',value:'Required'},
{nodename:'SubmitMethod',value:'Manual'}
];
sfc.updateChildNodeInformation(newRow,nodeArr);
```

Note: You must add at least one instance of the "classic" visual business rule, even if it just returns false. Otherwise the Business rule controller will be optimized out.

Sample form:

composer production > Attachments > Programmatic Attachments

Transction Manager: Programmatic Attachment

Related articles

- Understanding the Attachment Field V4.1
- Attachment Definition
- Programmatic Attachments
- Attachment Time

- Customising Attachment Field Widget

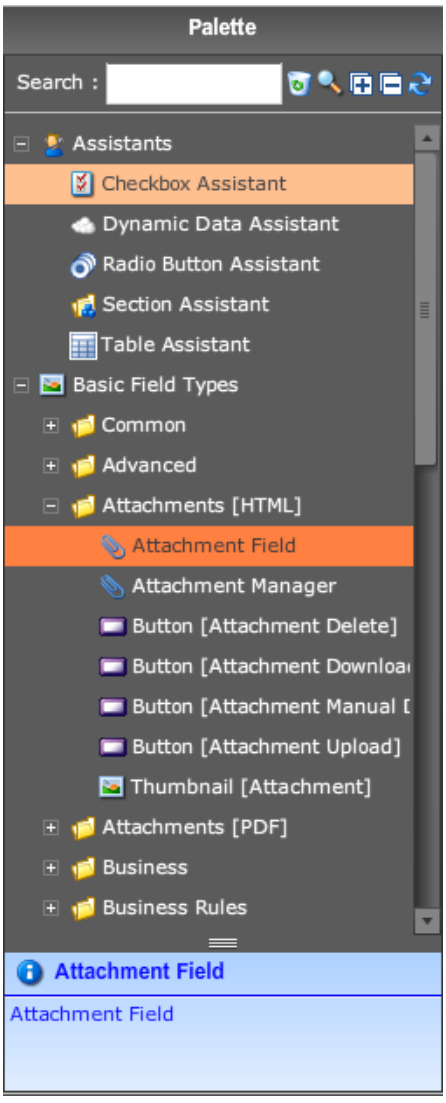
Understanding the Attachment Field V4.1

Unknown macro: 'redirect'

Prior to Avoka Transact Version 4.1, capturing attachments was only facilitated by the Self-Service Portal once data entry was complete and the 'Submit' button had been clicked in the form. With this approach attachment capture is an Independent step in the transaction process, completely separate to the form, and this sometimes makes the transaction a little disjointed. While this flow is still supported in 4.1, an alternative has been introduced to enable form designers to include file capture inside the form at the time that it is relevant. This is facilitated by a new Composer widget called the Attachment Field.

Finding the Attachment Field in the Palette

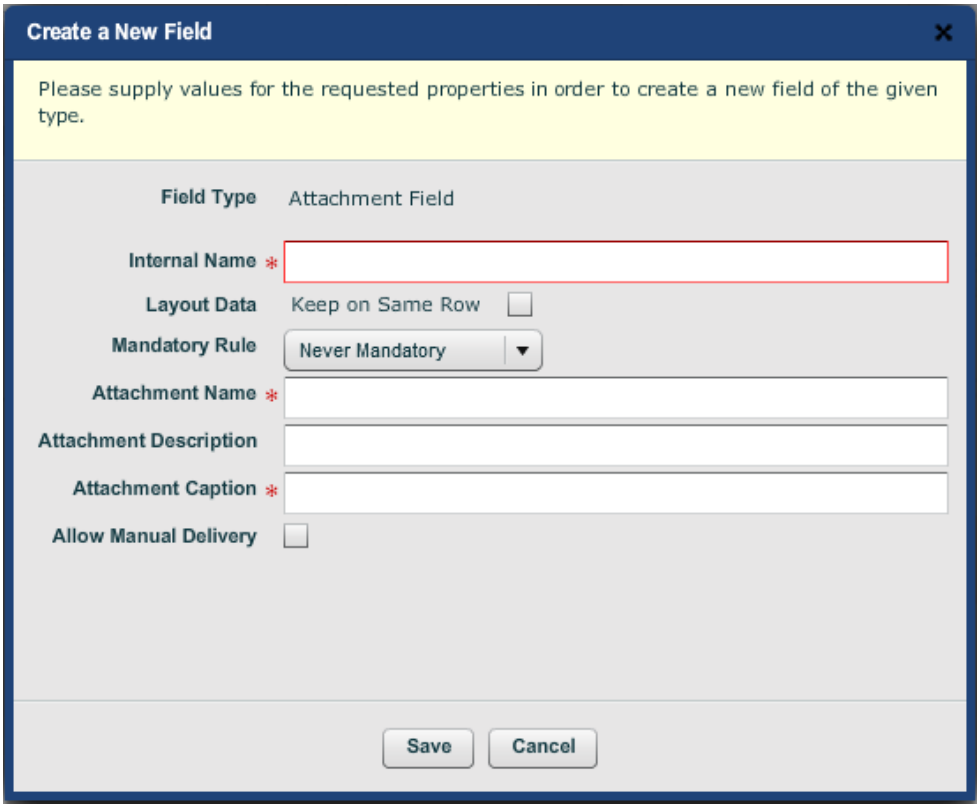
You can find the "Attachment Field" under the "Attachments" category.



Configuring the Attachment Field

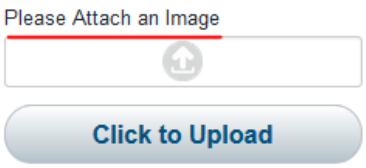
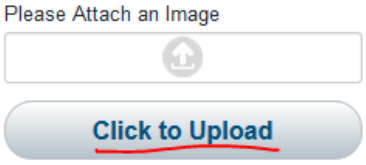
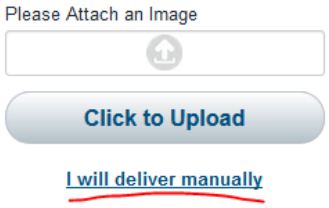
Initial Config Options

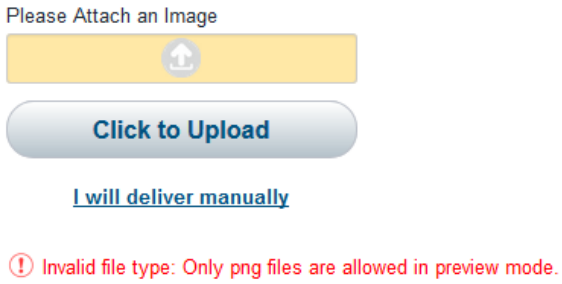
After you drop the field in a form you will get the following initial options, three of these are mandatory (with the red asterisk).

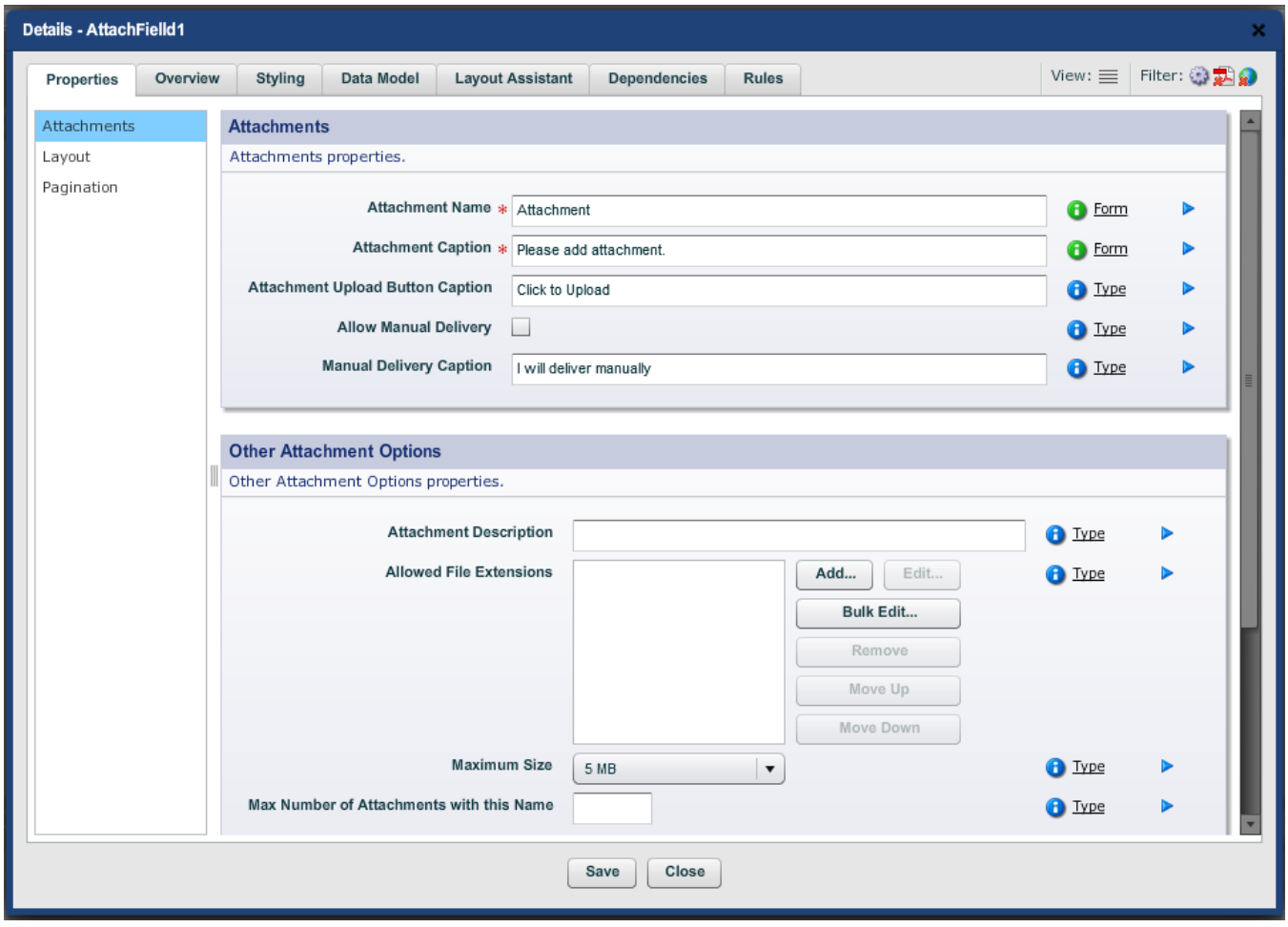


Basic Mode Config Options

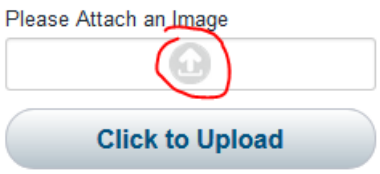
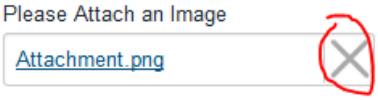
Within the field properties tab there is an "Attachments" category which contains the following basic mode options:

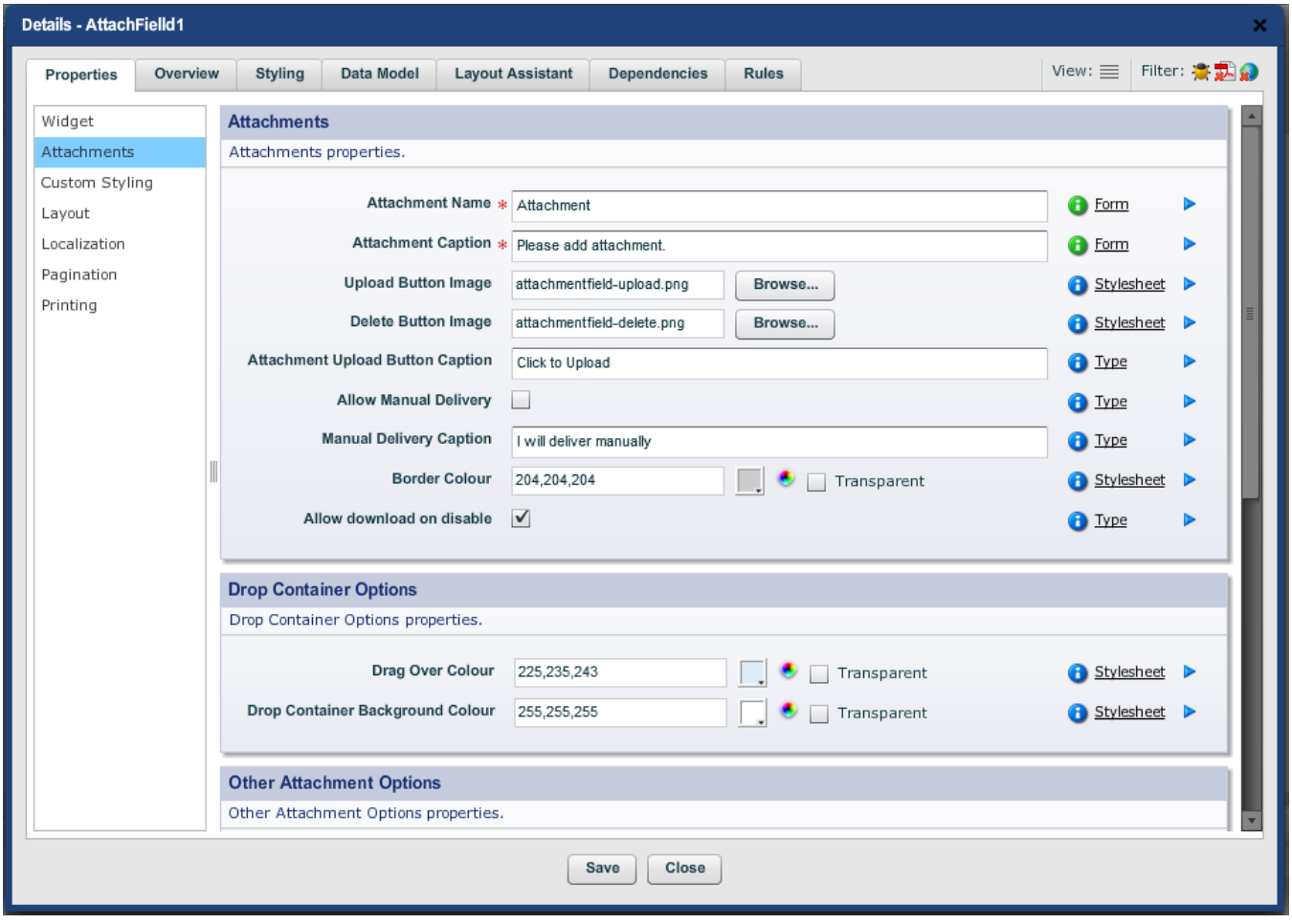
Name	Description	Image
Attachment Name	This is the attachment name that will be sent to TM.	
Attachment Caption	This is the caption text that will display at the top of the field.	
Attachment Upload Button Caption	This is the caption text that will display in the upload button.	
Allow Manual Delivery	If set to true, a link button will appear below the upload button which allows the user to delivery the attachment manually.	See images for "Manual Delivery Caption"
Manual Delivery Caption	This is the caption text that will show in the manual delivery link button.	

Attachment Description	This is the attachment description that will be sent to TM.	
Allowed File Extensions	This is where you can set the allowed file extensions. If set, an error message will return if the user uploads a file with an extension not in the list.	 ^ Will only allow .png files.
Maximum Size	This property allows you to set the maximum file size allowed. Ranging from 0.5MB to 25MB. An error message will return if the user uploads a file with a size beyond the limit set.	
The Max Number of Attachments with this Name and use Default Progress Indicator	These properties should be left as default for the 4.1 release.	



Advanced Mode Config Options

Name	Description	Image
Upload Button Image	Allows the user to change the upload image which is shown in the initial field state	
Delete Button Image	Allows the user to change the delete image which is shown in the completed field state	
Border Colour	Allows the user to change the border colour for the attachment container, this can be done in a style sheet using the fields type style.	
Allow download on disable	Set to true by default, this allows the attachment download button to still be active when the field is in read-only mode.	
Drag Over Colour	Allows the user to change the colour used when a file is dragged over the container , this can be done in a style sheet using the fields type style.	
Drop Container Background Colour	Allows the user to change the Drop Container background Colour , this can be done in a style sheet using the fields type style.	



Testing in Preview Mode

Only zip files are allowed in preview mode.

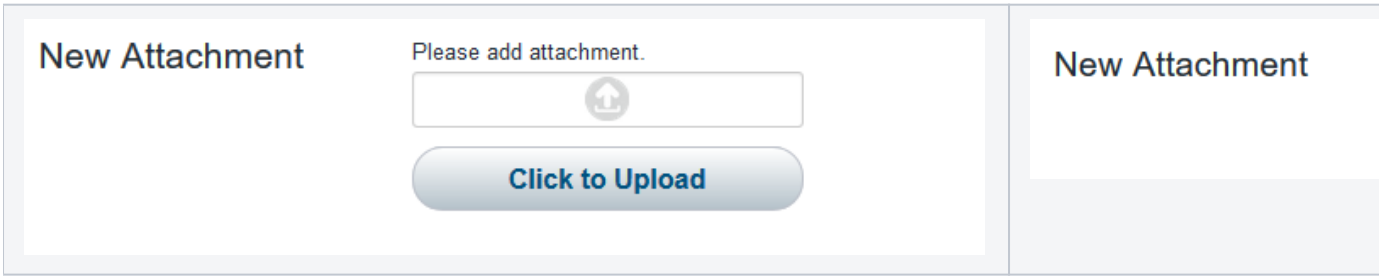
The User Experience

Interacting with the Attachment Field

To capture an attachment, add the Attachment Field to the form in the location that you wish to capture the attachments. Attachment fields support visibility, editability and mandatory rules like standard fields.

In its empty state, the attachment field shows the 'Click to Upload' button which launches the standard browser file dialog. When an attachment that complies is provided the upload button disappears and the file name is populated into the field as a clickable link that will trigger a download if clicked.

This link will be active, even if the field itself is disabled. The form can be cleared again by clicking the delete ('X') button.

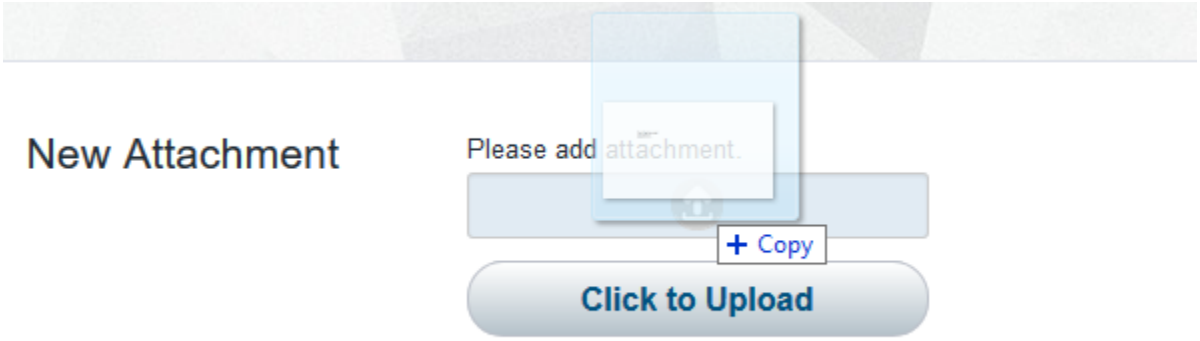


Form designers can configure the attachment field to support the manual delivery option where users indicate they will provide the attachment by conventional means.

> ***NOTE**: As of version 4.1, the attachment field is not yet supported in TransactField although it is intended that this capability will be included in a later release.*

Drag and drop support

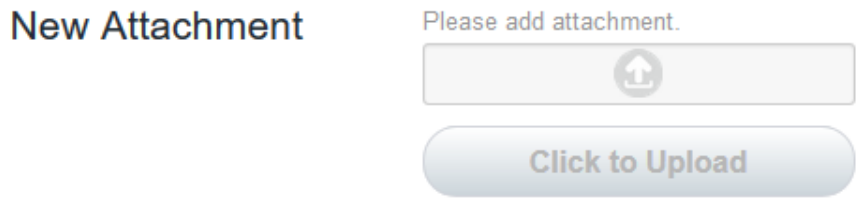
Drag and drop is supported in most browsers (IE8 and IE9 are not supported)



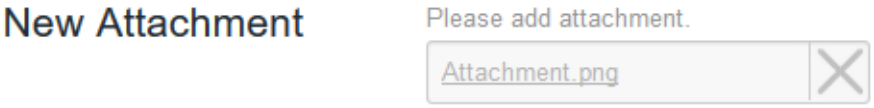
Read-Only Behavior

In read-only mode when the property "Allow download on disable" is set to true (it is true by default) the delete button is hidden and the download attachment button is clickable.

"Allow download on disable" set to true



"Allow download on disable" set to false



Exposing Attachments in Review & Approval Style Collaboration Jobs

Collaboration Jobs runs over a number of steps. The step name is passed by Transaction Manager to the form. Editability rules can be applied to attachment fields based upon the step name.

The initial applicant filling out a blank form usually wants full write access to an attachment field (upload / download / delete). However a reviewer checking the form may be required to view the attachment but not edit it. By configuring the editability rule to only disallow edits in subsequent steps, this scenario can be easily accommodated.

`NOTE: your Attachment Field must have "Allow download on disable" enabled (default).`

Support Notes

In TransactField, the attachment field displays as in receipt mode. In the future the attachment field will include the capability to upload attachments within TransactField.

Related articles

- Understanding the Attachment Field V4.1
- Attachment Definition
- Programmatic Attachments
- Attachment Time
- Customising Attachment Field Widget

Composer Administration

Unknown macro: 'redirect'

- Allowing Composer Cloud to publish forms to Transaction Manager
- Best practice guide to managing your organisation within Composer
- Composer Security V4.2
- Form Revisions V4
- How to create a Library in V4
- How to create and share a library V4
- Importing/Exporting Organizations from Composer
- Updating the name of a Composer form V4

Allowing Composer Cloud to publish forms to Transaction Manager

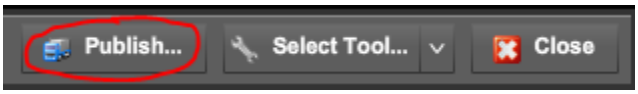
Unknown macro: 'redirect'

Composer can publish form packages to Transaction Manager, either directly or using ZIP file exports.

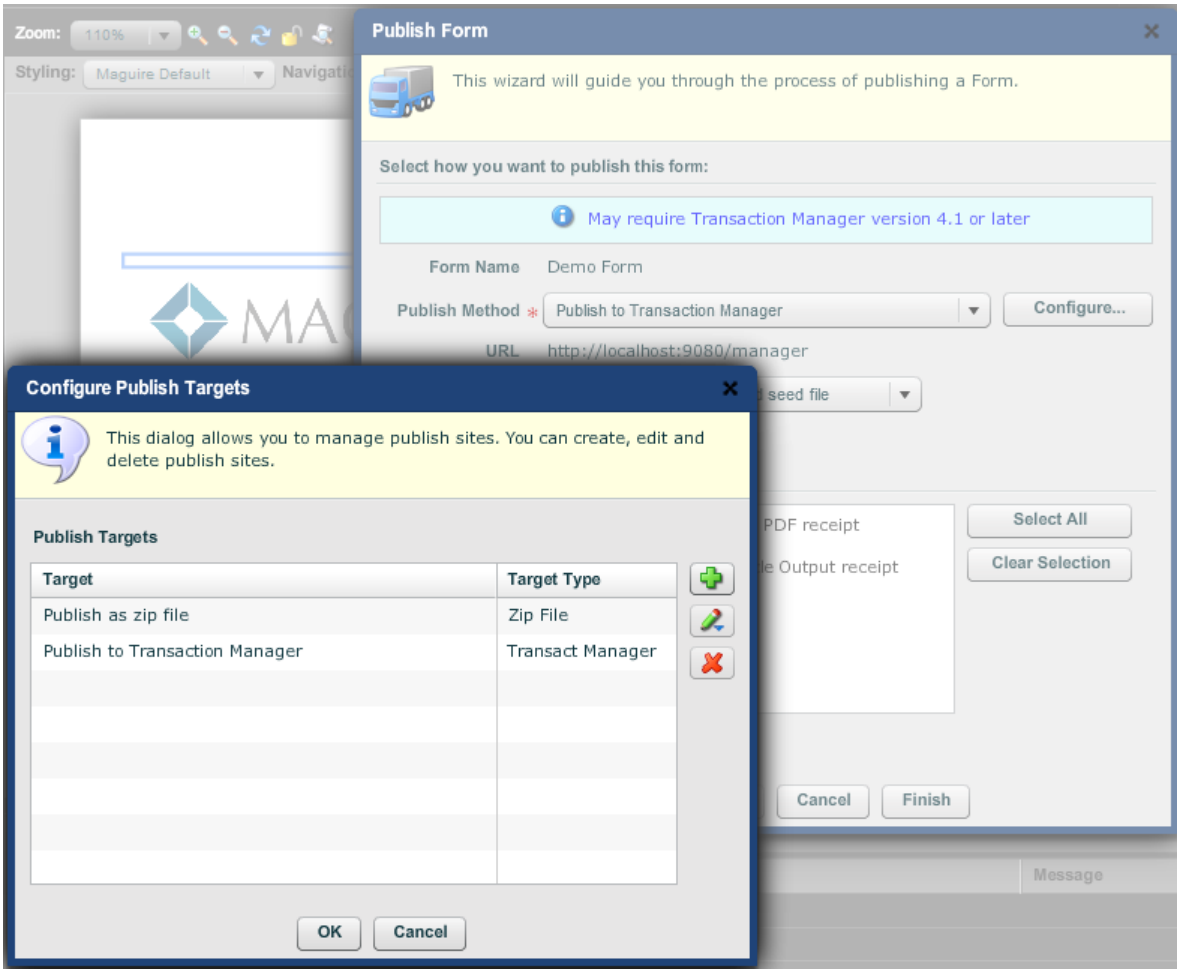
To publish forms directly from Composer to Transaction Manager, your instance of Composer must have the deployment property "External URL" set to the URL of Composer as visible to Transaction Manager. If this is not possible, you can still publish forms by exporting a ZIP file and importing it into Transaction Manager.

To publish forms directly from Composer:

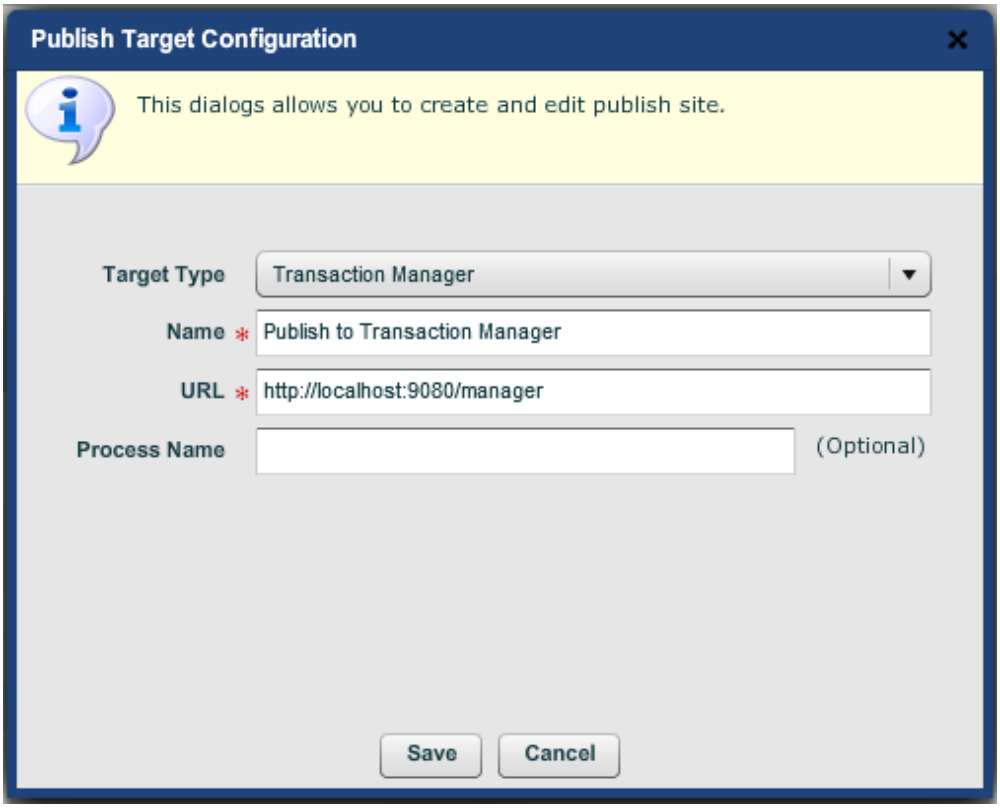
1. Click the Publish Button at the top right hand corner of your form



2. To modify the URL of the Transaction Manager instance that will be published to, click "Configure..."



3. Edit the entry associated with publishing to Transaction Manager
(alternatively, you can add a new entry with a different URL). Adjust the URL and save your changes.



4. Once you are satisfied with your publishing target, select the form types you want to include and click "Finish".

If you access Composer in the Cloud this is already configured but if you are experiencing issues it may be due to a Firewall blocking the access.

Still not able to access TM?

Most companies are very strict on Security policies. To enable you to publish forms directly from Composer to your on premise Transact Manager you will need to do the following things.

If you are having trouble getting to Transact Manager there are 2 options you have to resolve

1. Contact your IT department to allow Composer to push to TM within the Firewall

2. Contact Avoka Support with a support ticket and we will assist you to add Composer's public IP addresses the your Apache/TM whitelist. This will restrict access / manager. To do this please provide us with all IP addresses that you want to whitelist.

Related articles

- Task Assignment
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer
- How to create a Library in V4

Best practice guide to managing your organisation within Composer

Unknown macro: 'redirect'

This article gives an overview of best practices, including practical considerations for configuring your SmartForms (old, new and in forms in development) within Composer. These are guidelines rather than strict rules.

A good place to start is the formation of a series of guiding principles for your organization. These do not need to be complex and can be done in a simple way.

Composer Organization Administration Suggestions

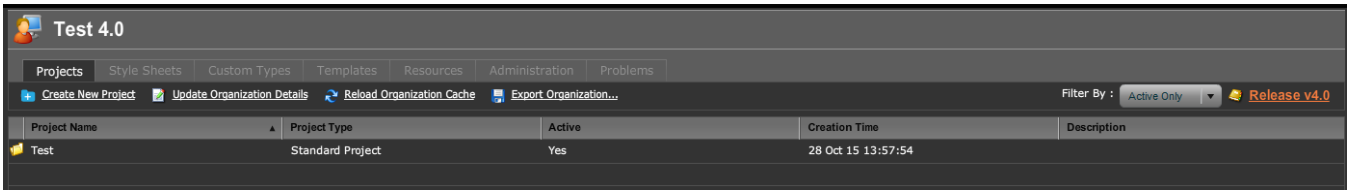
- Choose a logical and consistent way to organize your SmartForms to allow others to locate and use them. Group forms into Projects that are easily identified within your organization. This could be done a number of ways including
    - Department or agency
    - Type of form e.g. HR, Admin, Travel, IT
    - Status of form (Archive, Live, Dev)
    - Dates Created
- Think about a basic naming convention for your SmartForms at the start of a project.
- Project Naming. Projects within Composer should be named with clear meaning. Do not create names which are meaningless (or only mean something to you), are excessively long, or relate to individuals.
- Current and completed work – it may help to separate current and completed work or versions or files/documents e.g. where a document will have many versions and multiple contributors consider a “Current version” folder.
- Review what you have – don’t keep pointless multiple copies of data, and consider carefully what you need to retain, for how long, and what can (and can’t) be destroyed/deleted. Consider this at intervals and at the end of a project.
- Version control
- When making changes to live forms, create a copy first so you always have the latest deployed form saved.
- Archive and remove older forms from your Composer Organization to a safe place within your organization so you don’t have multiple versions of the same form and are not sure which was the last one deployed.
- Consistency – agree what works and stick to it

While growing your Smartform library you do not want to get into an administration nightmare so it is sensible to plan in advance.

Your Organization

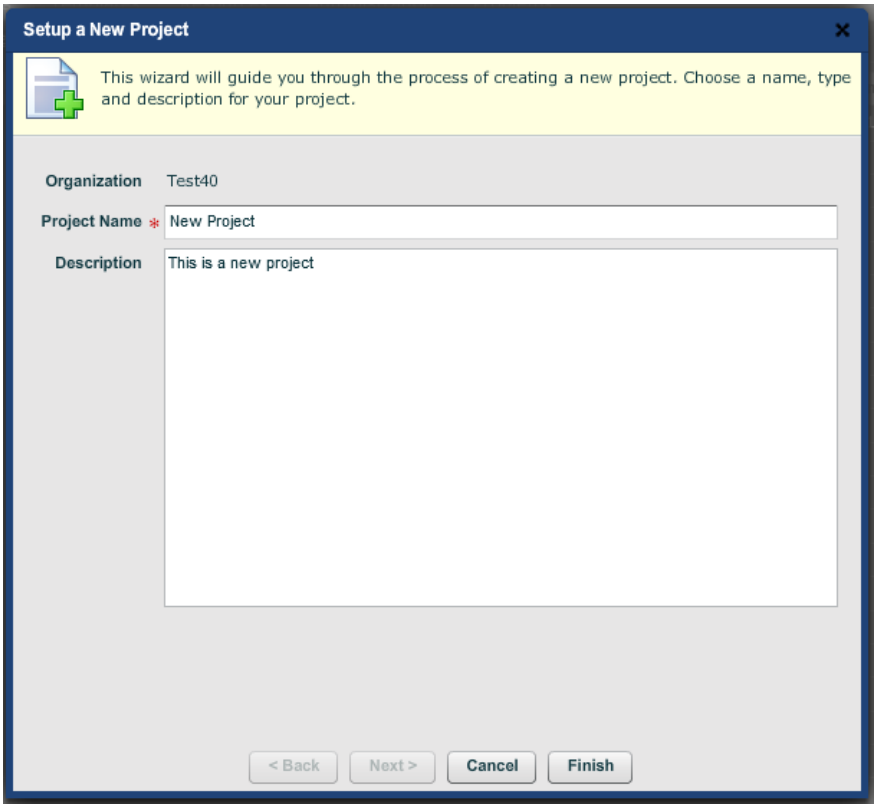
Forms are owned by organizations within Composer. Normally, you will be a member of just one organization, though you may see more than one depending on your security profile. Most users will also see at least one organization containing sample forms.

Your organization displays as the top level item in the Projects panel, as shown below. Projects are displayed under the organization.



Projects

A project is a convenient container for a number of forms - usually forms get created in groups or batches, and a project is a simple way to group them. You can also restrict which users have access to each project. Click on the Create New Project button at the organization level to create a project, and follow the wizard steps.



Saving a large amount of files in one Project can make it difficult to find a form and there is no search capability within Composer so please plan this carefully in advance. However if you click on the 'Creation Date' or 'Last Modified Date' column heading in the project view, you can reorder the form listing in ascending or descending order.

Key Features you can utilise in Composer include:

- Forms are automatically checked out to you when you open a form. This means that when the form is being worked on, no one else can access or make changes/edit the form. Therefore remember to close the form when you have finished using it.
- Manage revisions or versions of the form and, for example, revert to a previous version

Suggestions for Smartform and Project naming conventions

- Keep file and folder names short, but meaningful.
- Avoid using non-alphanumeric characters in file name.
- Avoid unnecessary repetition and redundant words in file names and file paths.
- Use capital letters to delimit words, not spaces.
- When including a number in a file name always give it as a two-digit number rather than one, i.e. 01, 02 … 99, unless it is a year or another number with more than two digits.
- If using a date in the file name always state the date ‘back to front’, and use four digit years, two digit months and two digit days: YYYYMMDD or YYYYMM or YYYY or YYYY-YYYY.
- Order the elements in a file name in the most appropriate way to retrieve the record.

Suggestions for Composer form Versioning

- Give all forms unique names
- It is advisable to append the forms current version number e.g. ‘Banking Form v1.2’ to the form name.
- When making changes to a form create a new copy of the current version (using the ‘Save As’ functionality) and update the version number of the form name with the new version.
- Additionally, if multiple people maybe editing the same form over its lifetime, when a form version is in an unstable state (due to incomplete development) it can be worth appending some identifier to its name to indicate that it’s a work-in-progress, such as the initials of the person making changes e.g. ‘Banking Form v1.3 – JS’. Once that person has completed their work they can rename the form to remove their initials.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Task Assignment
- Collaborative form completion with anonymous users (form sharing)

- Addresses

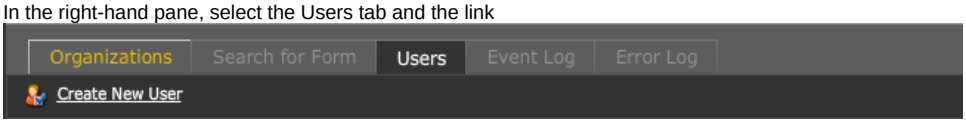
Composer Security V4.2

Unknown macro: 'redirect'

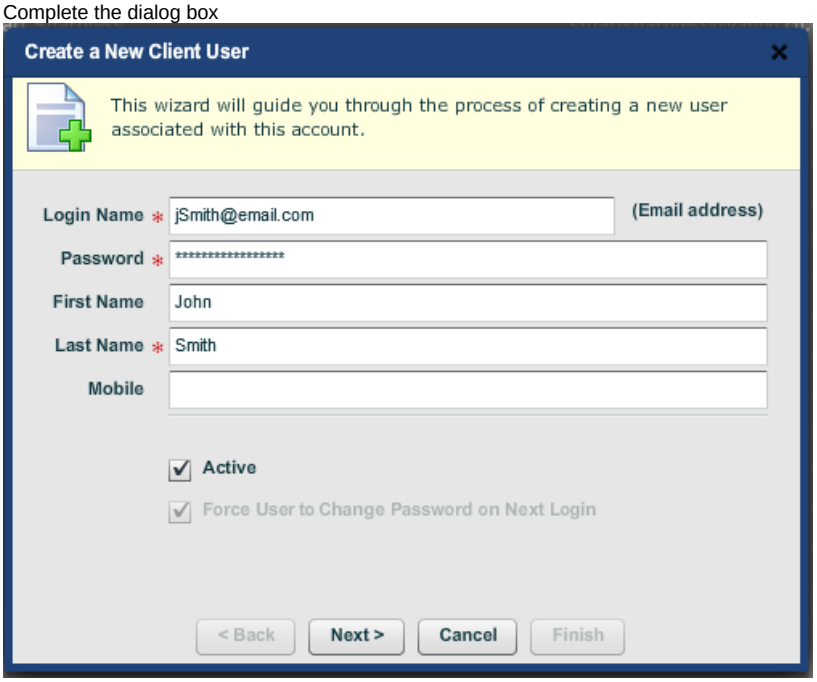
- In Composer, there are 2 standard roles by default i.e. Account Administrator and Developer.
- In order to create new developers or update the details of existing users, you will need to be an Account Administrator.

Process to create new developers

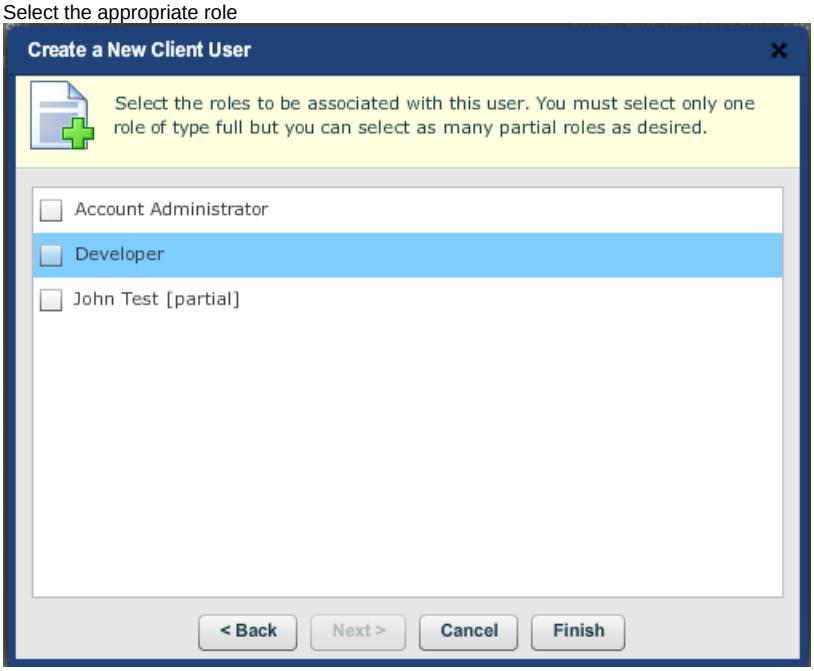
- Select your Account.
- In the right-hand pane, select the Users tab and the link



- Complete the dialog box

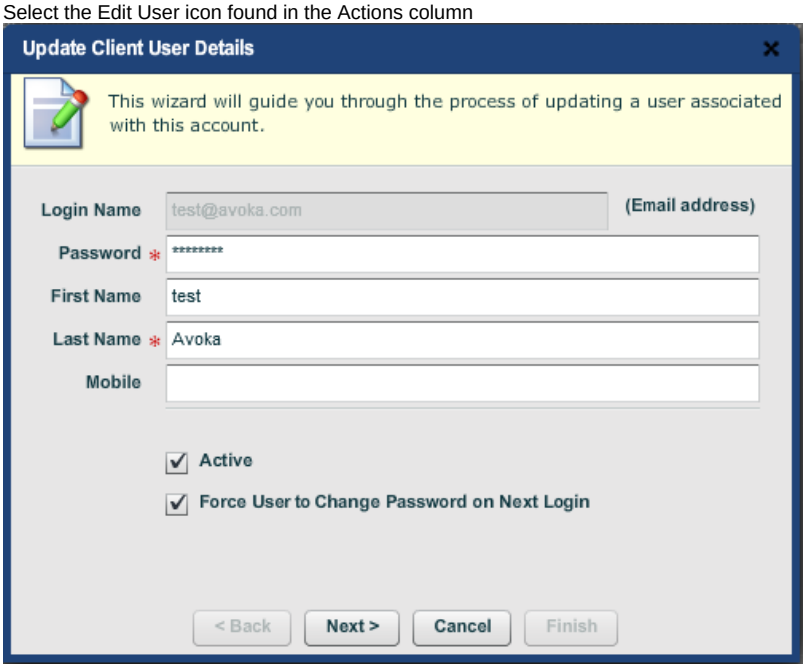


- Select the appropriate role



Process to change a password or role

- Select your Account.
- Select the Users tab and select the user.
- Select the Edit User icon found in the Actions column



- Update the relevant details.

For more information, refer to the Account Administration Guide.

Related articles

- Adding a security question in Maguire forms
- Composer Security V4.2
- How to setup a new Role in Transaction Manager
- Managing Content Security Policy (CSP) Settings

Form Revisions V4

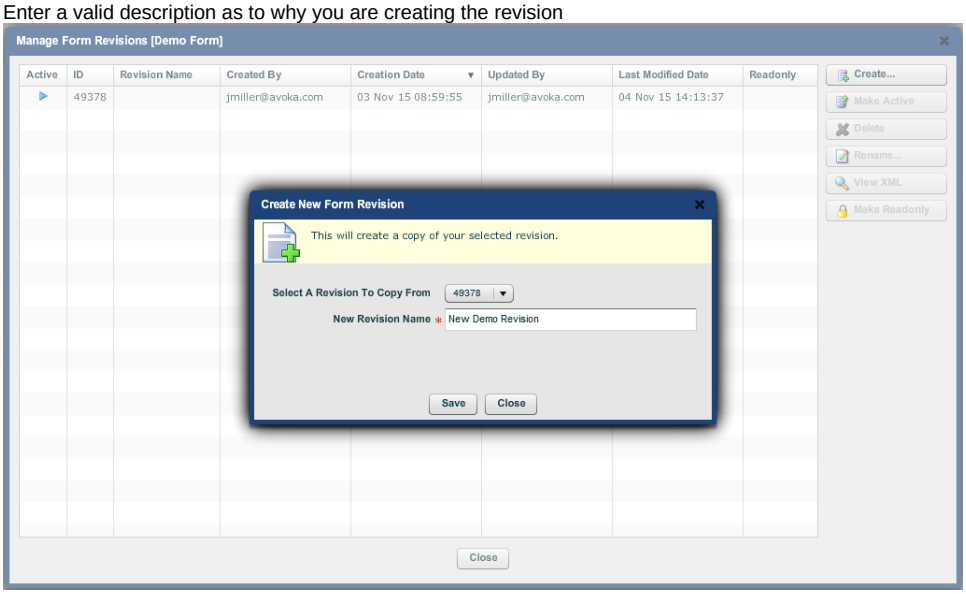
Unknown macro: 'redirect'

At various times during form development, it may be prudent to make a revision of the form.

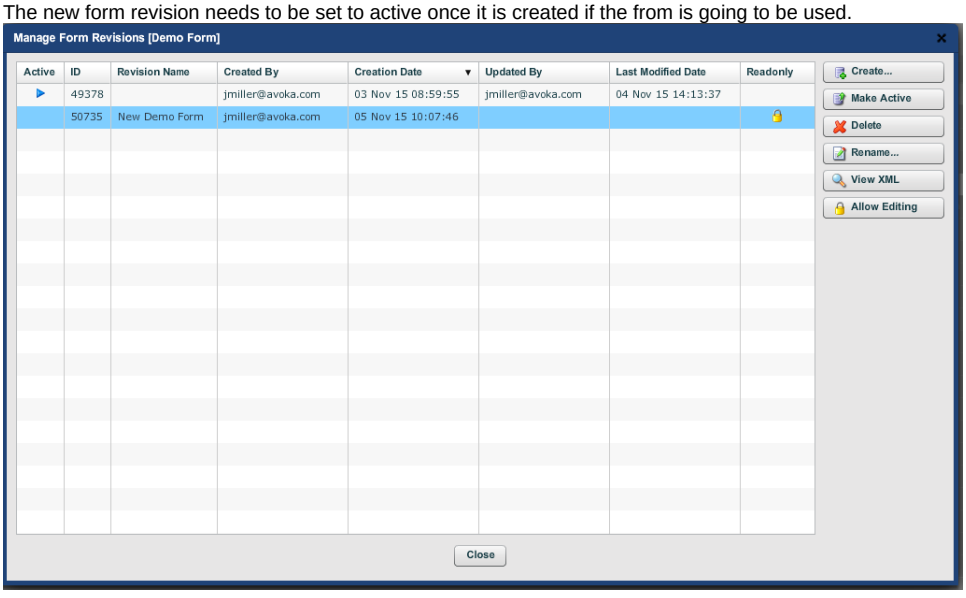
Form Revision

Method 1:

- To create a form revision, select the form in the project and select the 'Manage Form Revisions' option.
- Enter a valid description as to why you are creating the revision



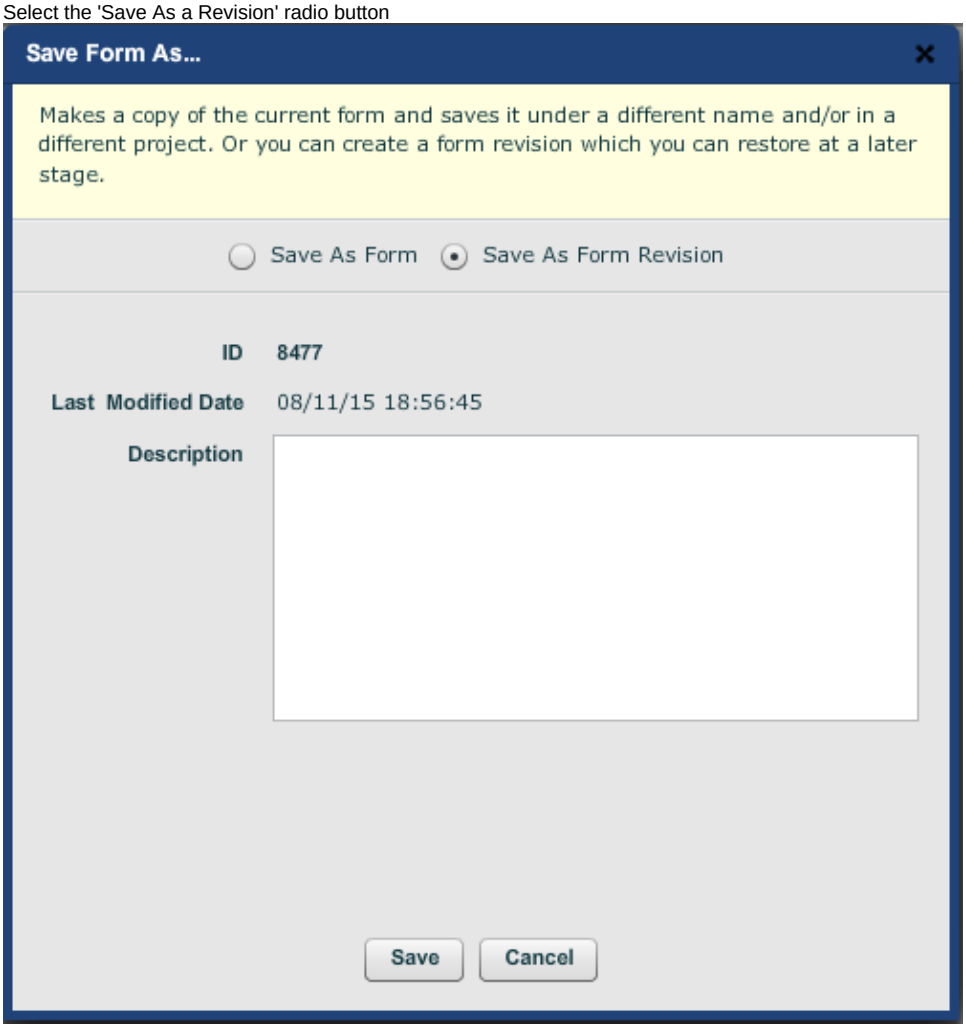
- The new form revision needs to be set to active once it is created if the from is going to be used.



Method 2:

- Select the form in the project listing and select the 'Save as' link.

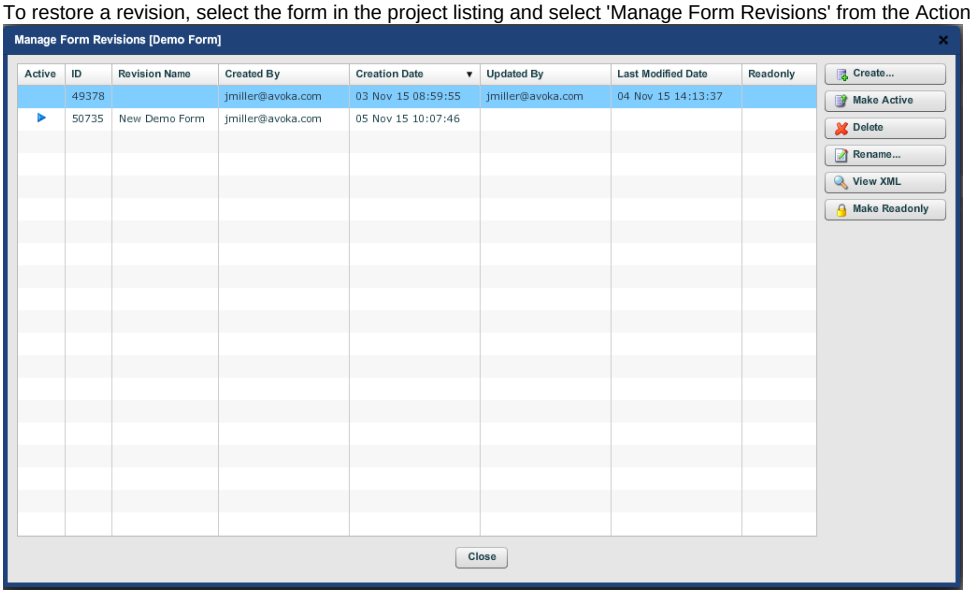
- Select the 'Save As a Revision' radio button



- Complete the Description property.

Restoring a Revision

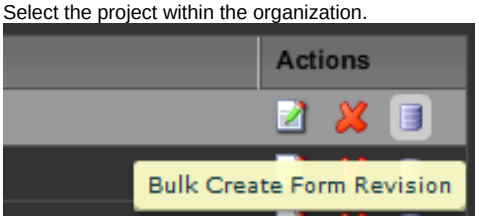
- To restore a revision, select the form in the project listing and select 'Manage Form Revisions' from the Action palette.



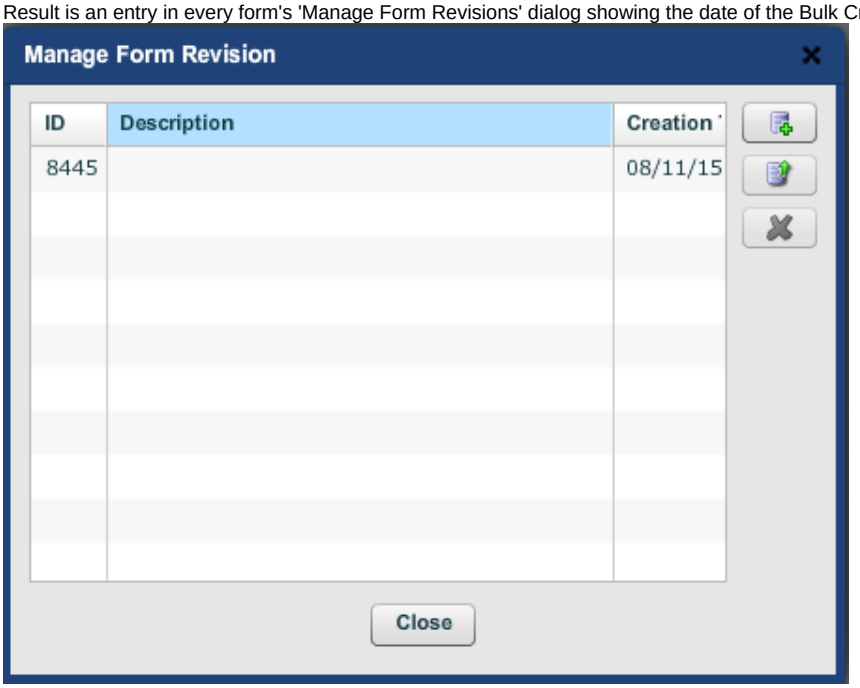
- Select the 'Restore' button.
- You will be required to enter a description of your restoration.

Bulk Revision

It is now possible to automatically create a revision for every single form within a project.

- Select the project within the organization.
- 

- Select the 'Bulk Create Revisions' options.
- You will need to confirm that you wish to proceed with the bulk revision.
- Result is an entry in every form's 'Manage Form Revisions' dialog showing the date of the Bulk Create run



Related articles

- Task Assignment
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer
- How to create a Library in V4

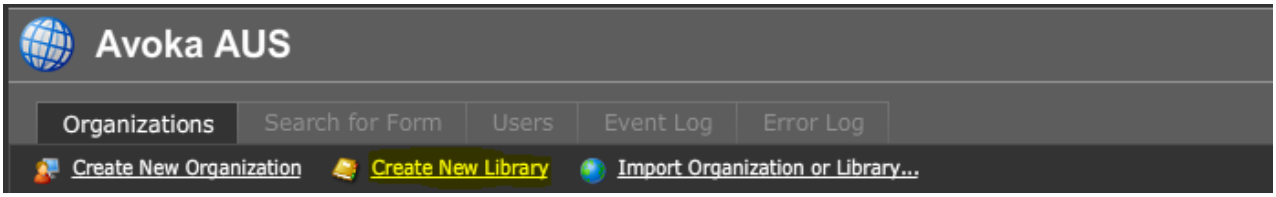
How to create a Library in V4

Unknown macro: 'redirect'

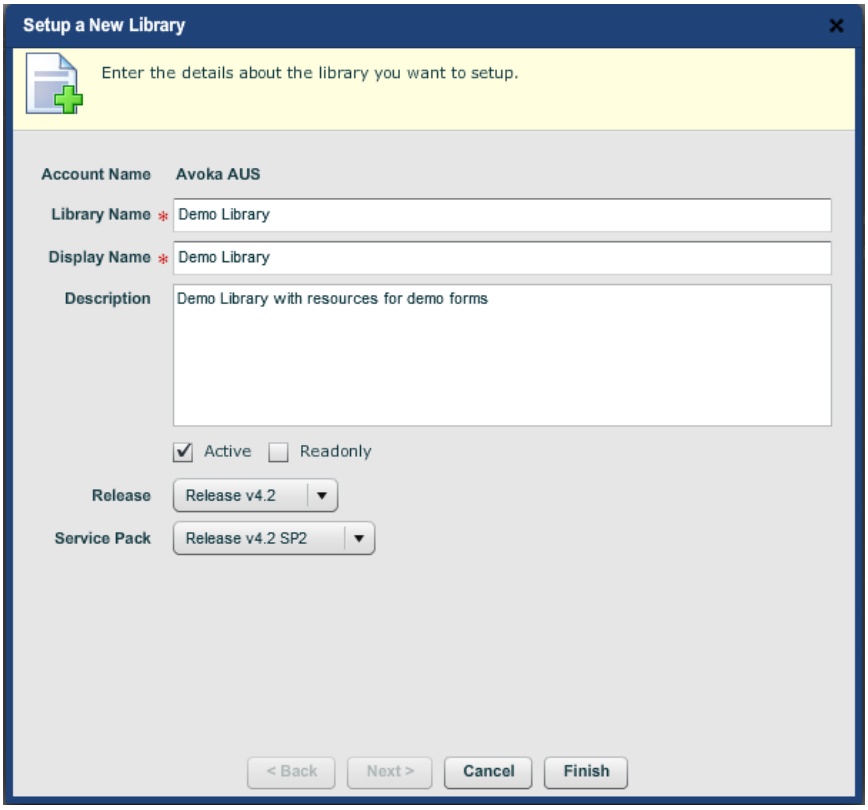
Libraries are a mechanism used to share information e.g. templates, style sheets, custom types and resources among organizations in your account.

To create a library:

Select the account and click the 'Create New Library' link.



Complete the 'Setup' screen.



Once you create a library, you are able to populate it with customized style sheets, templates and resources.

Related articles

- Task Assignment
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer
- How to create a Library in V4

How to create and share a library V4

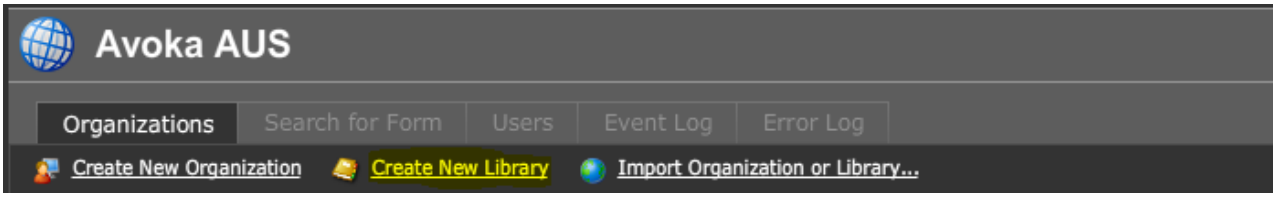
Unknown macro: 'redirect'

Create Library:

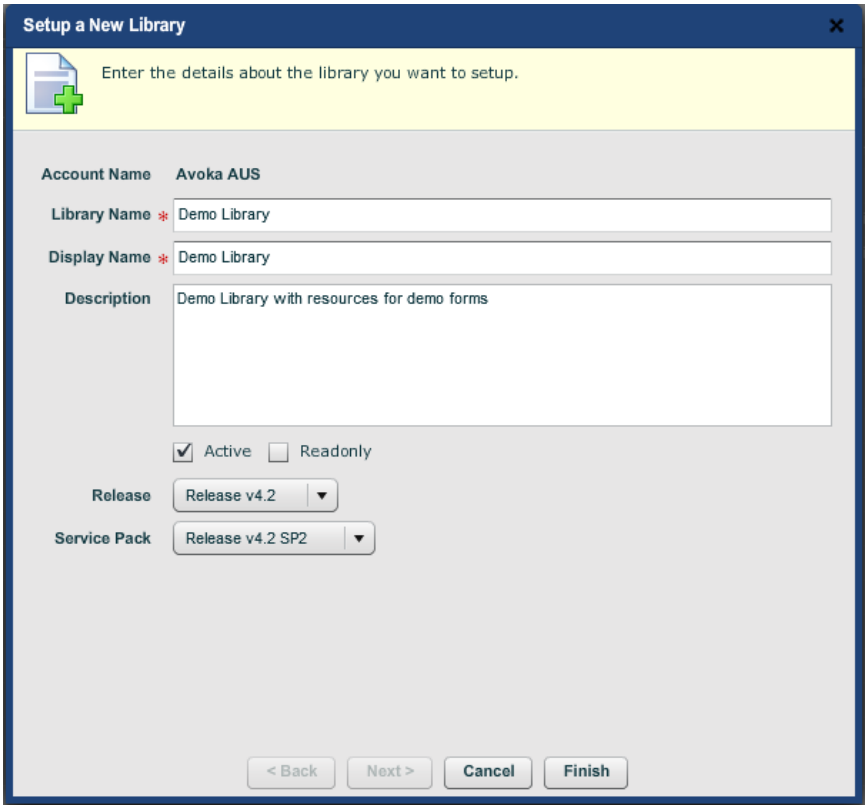
Libraries are a mechanism used to share information e.g. templates, style sheets, custom types and resources among organizations in your account.

To create a library:

Select the account and click the 'Create New Library' link.



Complete the 'Setup' screen.

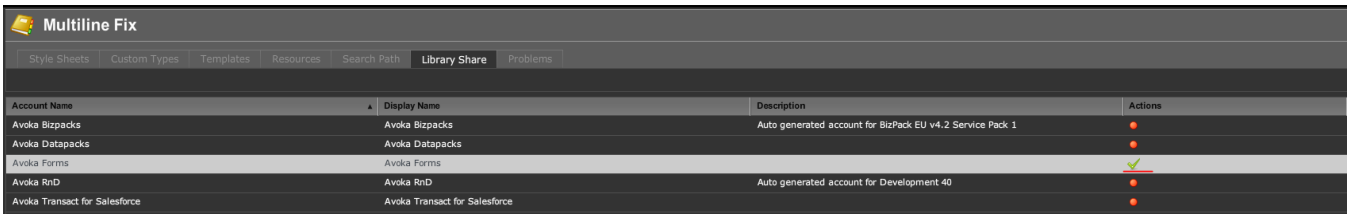


Once you create a library, you are able to populate it with customized style sheets, templates and resources.

Share Library:

Select the library name > Administration tab > Library Share - you will see a list of all the accounts you have access to.

In this example, we are sharing a library with account 'Avoka Forms' so under the Actions column, click the  icon.

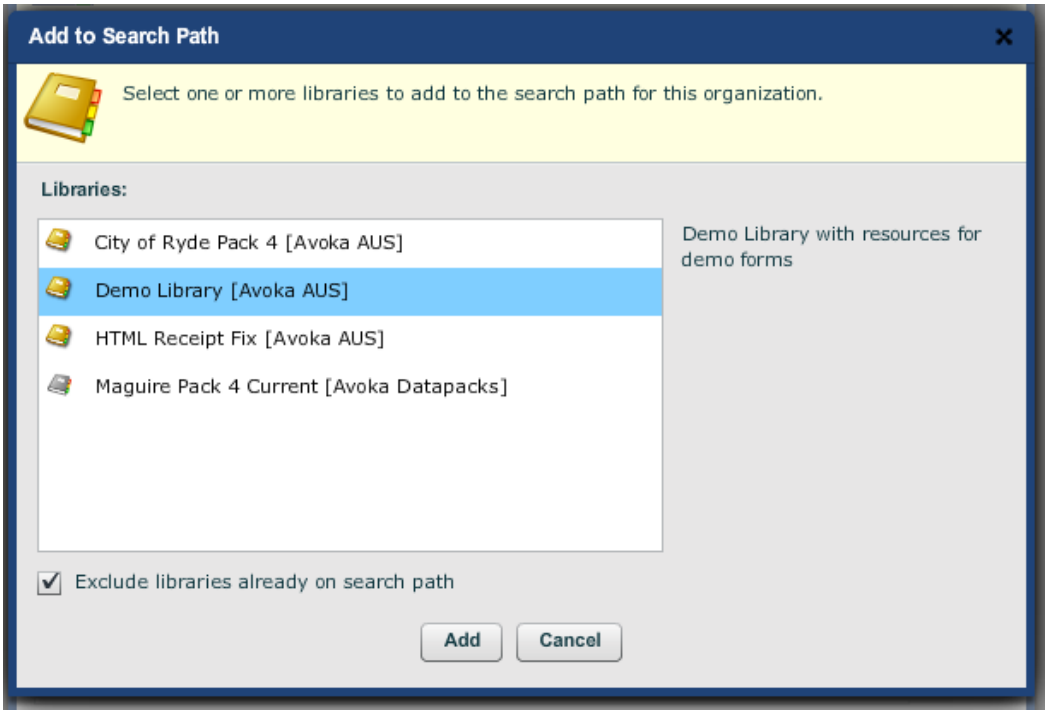


For an organization to be able to see the library, you need to update it's Search Path.

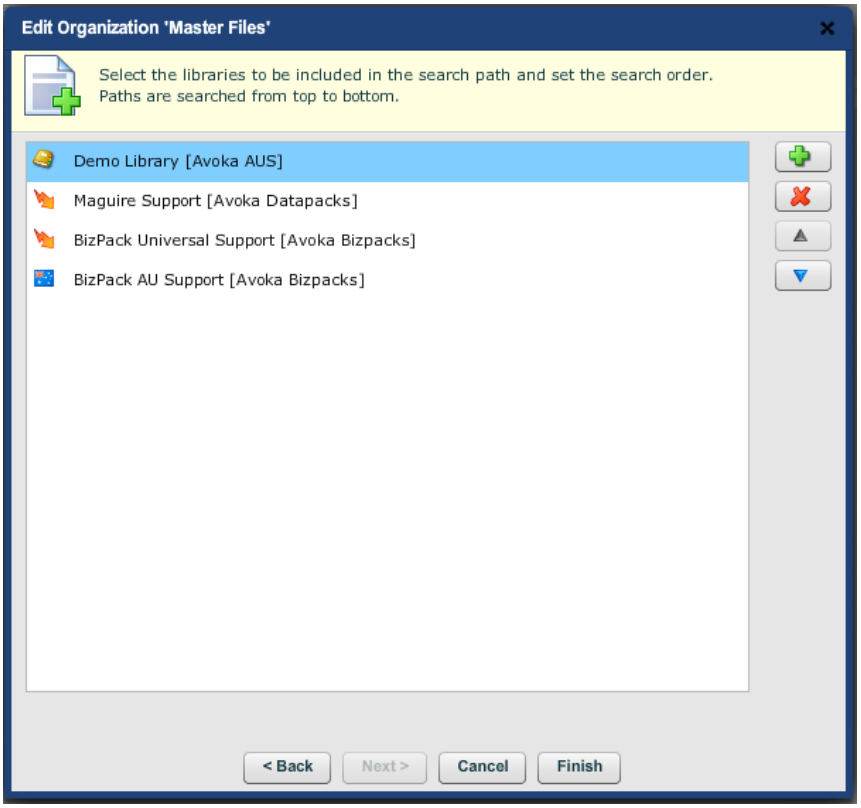
In the example, we are adding the Demo library to the organization 'Master Files' search path.

Select the target account > Master Files and select the link 'Update Organization Details' from the Action Palette.

Select the  icon and select 'Demo library'.



Result:



Related articles

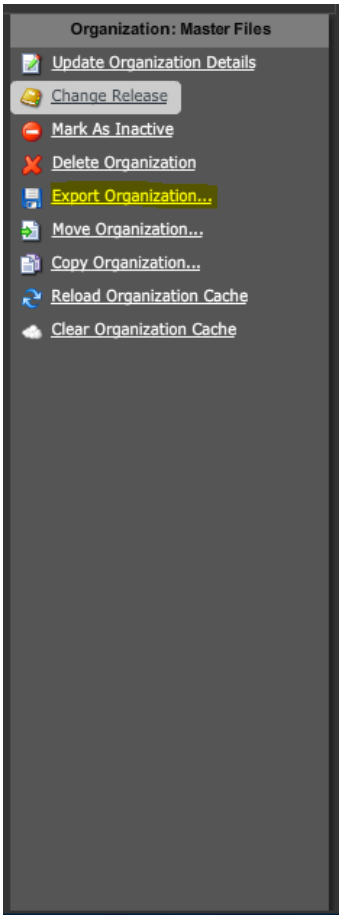
- Task Assignment
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer
- How to create a Library in V4

Importing/Exporting Organizations from Composer

Unknown macro: 'redirect'

Exporting an organisation

To export an organisation in Composer, navigate to the organisation in ‘Projects’ on the left panel, select the project and click the Export Organization link in the right hand menu,



Save the resulting zip file, and extract it by selecting ‘Extract here’.

Importing an organisation

To import an organisation, zip the folder containing the organisation files. It is important to zip at the right level. If you extracted the exported organisation zip using ‘Extract to (organisation name)\’ you will have the organisation files inside two levels of folders with the same (or similar) name. The file to zip is the one directly containing the organisation.xml file.

Once the correct folder is zipped, left click in the ‘Projects’ tab on the left panel in Composer, and select ‘Import…’. Select the zipped organisation file.

Importing into an existing organisation

When importing an organisation that already exists, any files in the import organisation that already exist will be overwritten. Any files that are in the current organisation, which are not in the import organisation will not be affected by the import. For this reason, the safest practice when importing files into an organisation is to export the organisation first, then delete everything except the organisation.xml file, then add the new files you wish to import, re-zip the organisation, and import it. This way you won’t overwrite any changes that were made to the organisation by other people in between the time you did the export and re-import.

Related articles

- Task Assignment

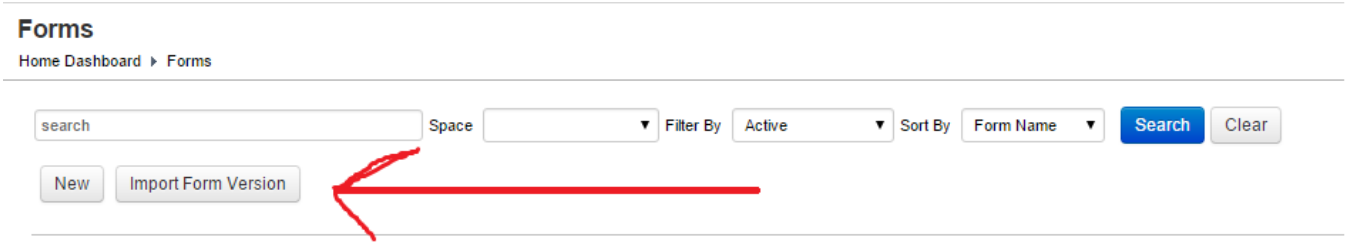
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer
- How to create a Library in V4

Updating the name of a Composer form V4

Unknown macro: 'redirect'

Prior to version 4...

... if you changed the name of a Composer form once it had been published to Transaction Manager, it was no longer recognized as being the same form. If you wanted to upload the form as another form version, you would have to publish it as a zip file, extract the .far file and upload it using the 'Upload new form version' on the form dashboard.

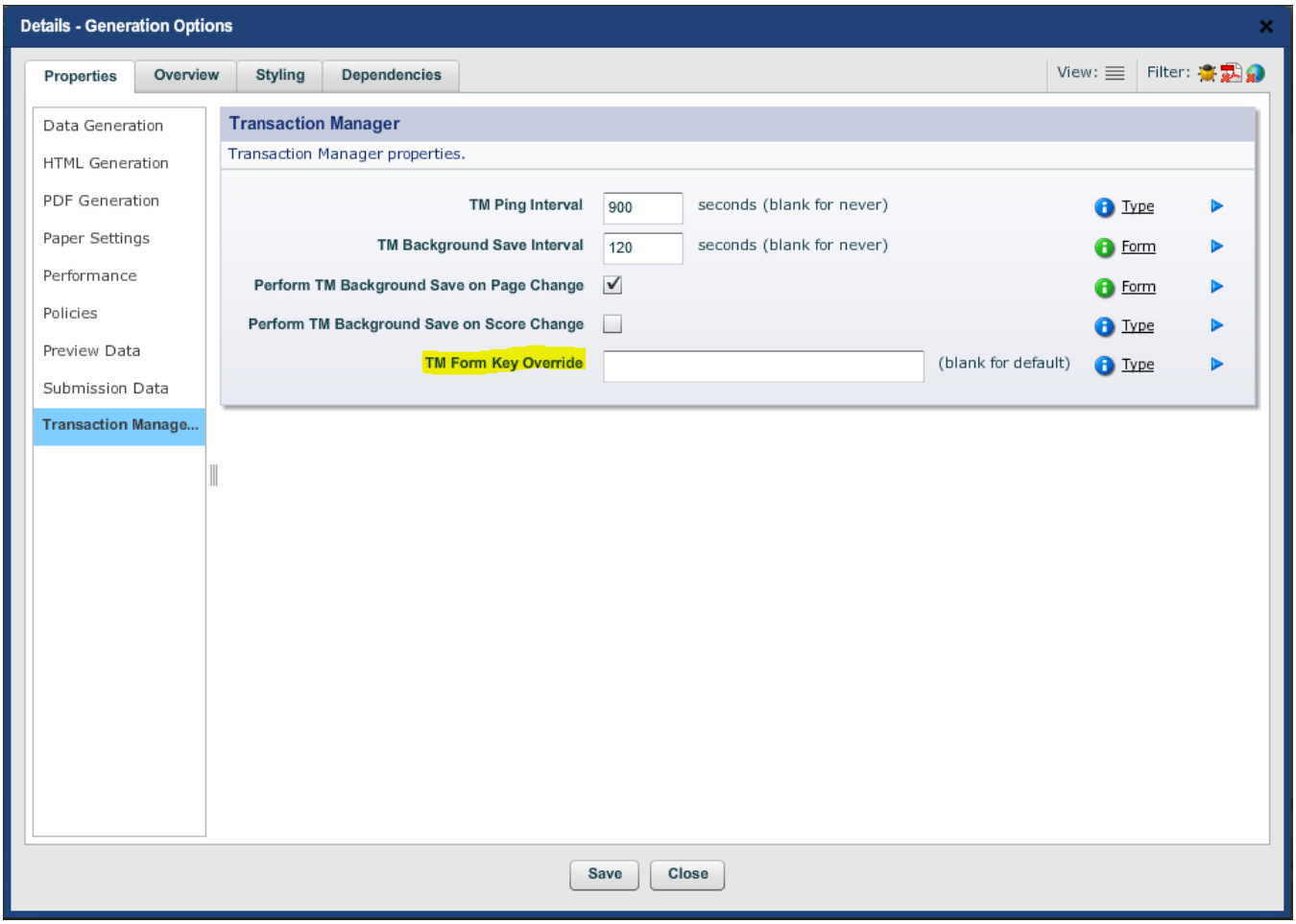


Version 4

In version 4, you can now give your form a unique identifier that will not be overridden if the form name is updated.

In the Structure pane > Nuts & Bolts > Generation Options > Properties > Transaction Manager, enter a value in the property 'TM Form Key Override'.

Note: this has to be done BEFORE the form is published to Transaction Manager for the first time.



Related articles

- Setting up 'Leave Page Confirmation' in Composer using JavaScript
- Understanding transaction management in Maguire forms
- Adding a security question in Maguire forms
- Blocking users with unsupported browsers
- Tuning the dependencies on the 'Triggering Conditions' tab in the Script Editor

Composer Development Tips

Unknown macro: 'redirect'

Enabling online save in Composer forms

Unknown macro: 'redirect'

This article describes how you can enable your anonymous or authenticated form users to save a partially completed form and return to it at a later time. This feature can be important for customer convenience, particularly for longer forms.

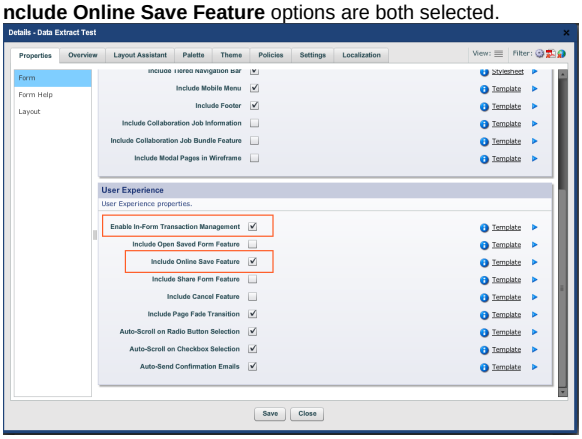
Compatibility

Since	4.1
Deprecated	

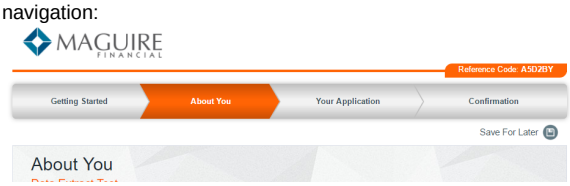
Support for online save in the Maguire Template

The Maguire template supports this feature out of the box and allows you to enable it very quickly.

1. Open the form properties and ensure that the **Enable In-Form Transaction Management** and **Include Online Save Feature** options are both selected.



2. Preview the form and note that the Save button is available under the form header and navigation:

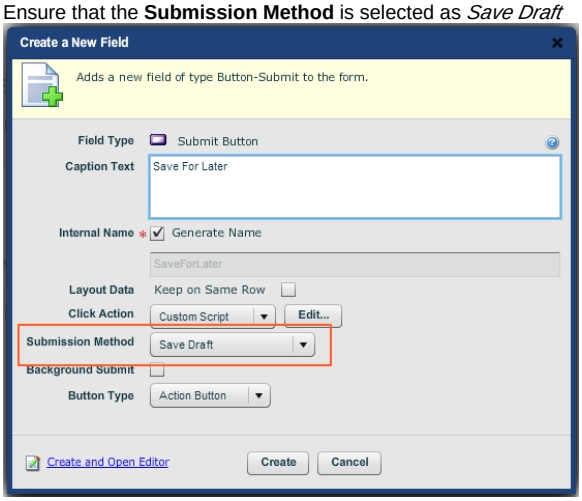


Manually adding a save button to a form

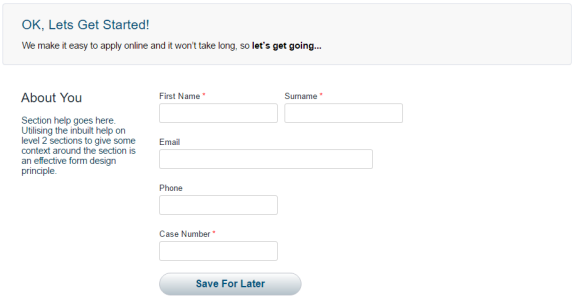
If you'd like to manually add a Save button to your form, follow these simple steps:

1. Locate the container you would like to add the button to and add a **Submit Button** widget from the palette.

2. Ensure that the **Submission Method** is selected as *Save Draft*

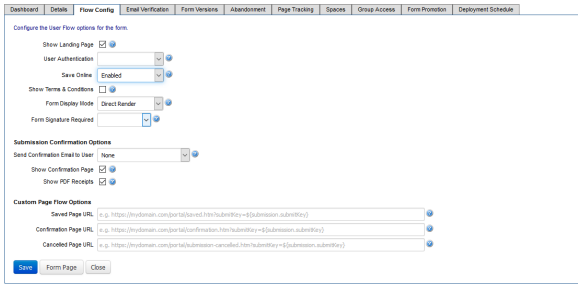


3. The save button is now available in your form (note: the save will not work when you are in Preview mode)



Save types in Transaction Manager

To allow the form filler to save a form before submission, the 'Save form' field must be configured in Transaction Manager. To configure this, edit the form in TM, and navigate to the 'Flow Config' tab.

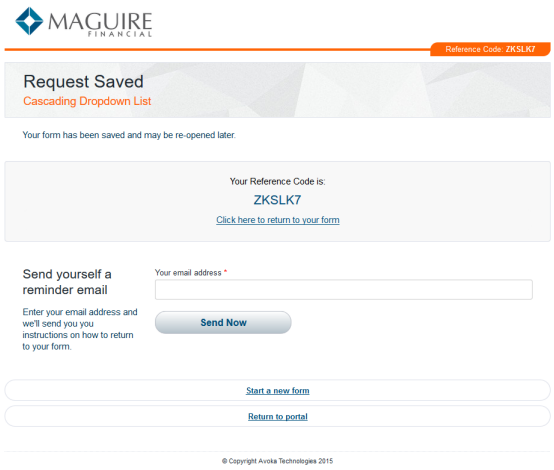


There are two types of online save; Authenticated Only, and Anonymous Only.

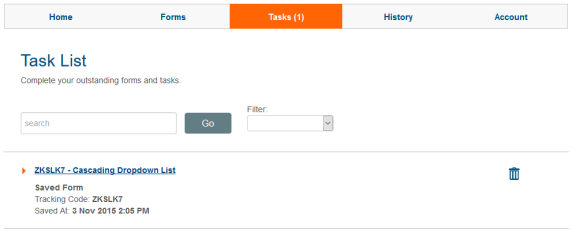
Authenticated Online Save

Authenticated online save requires the user to have a login account to the portal that the form is hosted on. On the Details tab (see screenshot above), ensure that you uncheck Test Mode field or the user will not be able to access the saved copy of the form via the portal.

When the form filler saves the form, they will be redirected to the page pictured below, with the option to either return to the form, or send themselves a link to the form to continue filling out at a later date, by providing an email address.



Authenticated users will also be able to access saved forms when they log into the portal. Any saved forms will appear in their task list, as pictured below:



Anonymous Online Save

When the user saves a form using anonymous save, they are given a URL to access the saved copy of the form. The sample form is configured to save anonymously: https://tm.demo.avoka.com/govassist/cookbook/save-html/

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Addresses
- Address Block Samples

How to configure background saving in Composer forms

Unknown macro: 'redirect'

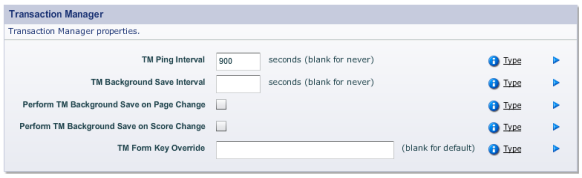
Compatibility

Since	4.0
Deprecated	

Since version 4, you are able to specify when data entered in a form should be saved, in the background. This makes it possible for the form filler to return to a form without having to reenter all the information they previously entered if his/her browser crashed, for example.

Composer form

To enable this feature, update the property 'Nuts & Bolts > Form Options > Generation Options > Properties > Transaction Manager.



There are 3 ways to configure background saving (and they are not mutually exclusive):

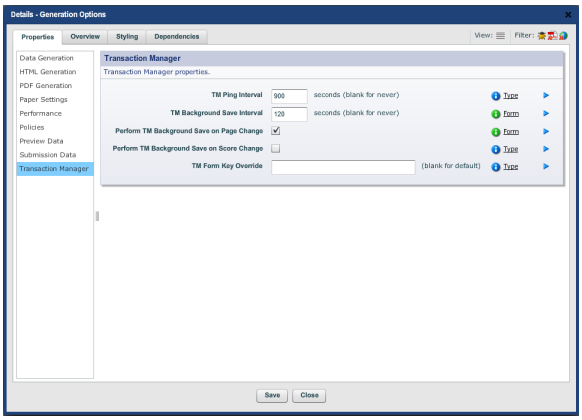
TM Background Save Interval: enter the number of seconds after which the form data will be saved;

Perform TM Background Save on Page Change: the form data will be saved when the form filler navigates to another page on the form;

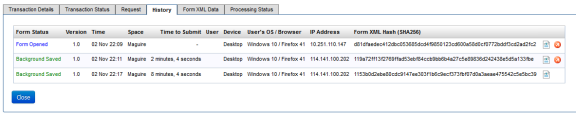
Perform TM Background Save on Score Change: the form data will be saved when the numerical score of the form changes (scores may be attributed to fields on a Composer form in order to determine where a form was abandoned). For this property to work, the feature must enabled in 'Nuts & Bolts > Transaction Manager Support > TransactionScore > Rules > Calculation > Script Editor (an example can be seen in the screenshot below):

Example

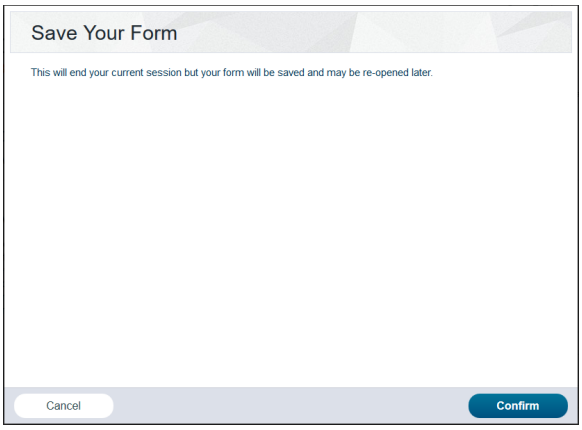
John Smith begins to fill in a form called 'Saving in Background' anonymously. This form has been configured in Composer with the following 'Background Save' options.



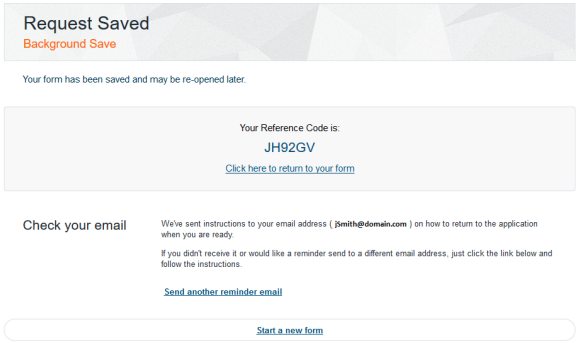
In Transaction Manager, if you select Operations > Saved Transactions, select the form's 'Tracking Code' (found in the top right hand corner of the form) and select the History tab, you will see that the form has been saved at various intervals. The  button allows you to see what has been completed in the form thus far.



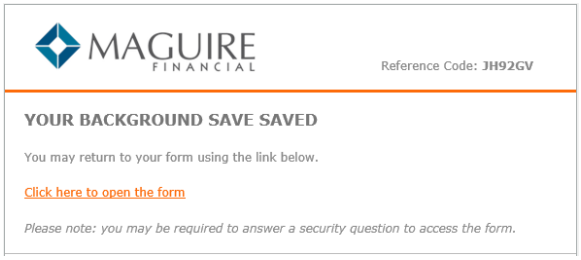
John then selects the 'Save and Close' button on the form and the following page is returned.



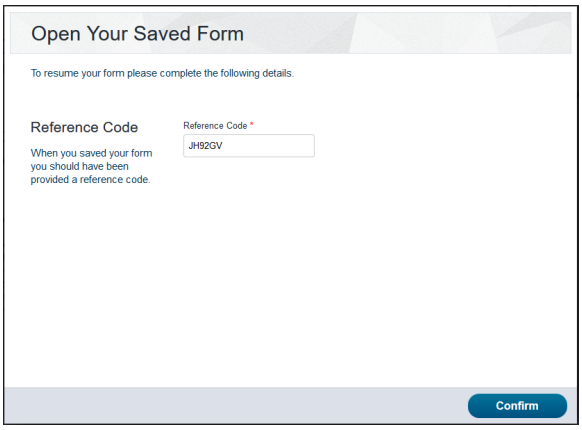
If he sends himself an email, he will get the following confirmation page.



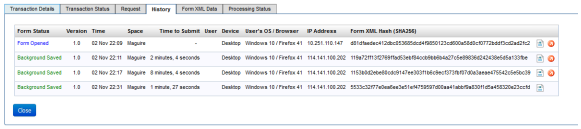
When he opens the email, he will see a link back to the form.



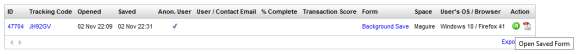
Selecting the form link, will take him to the following 'Security Check' page. This may be customized to include an additional challenge question if appropriate.



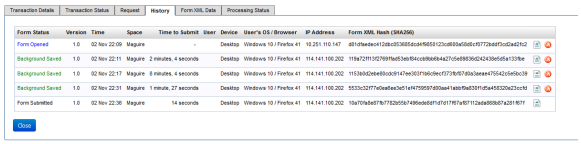
If he continues to fill in the form and saves it again, Transaction Manager will continue to save the form at regular intervals or on page change.



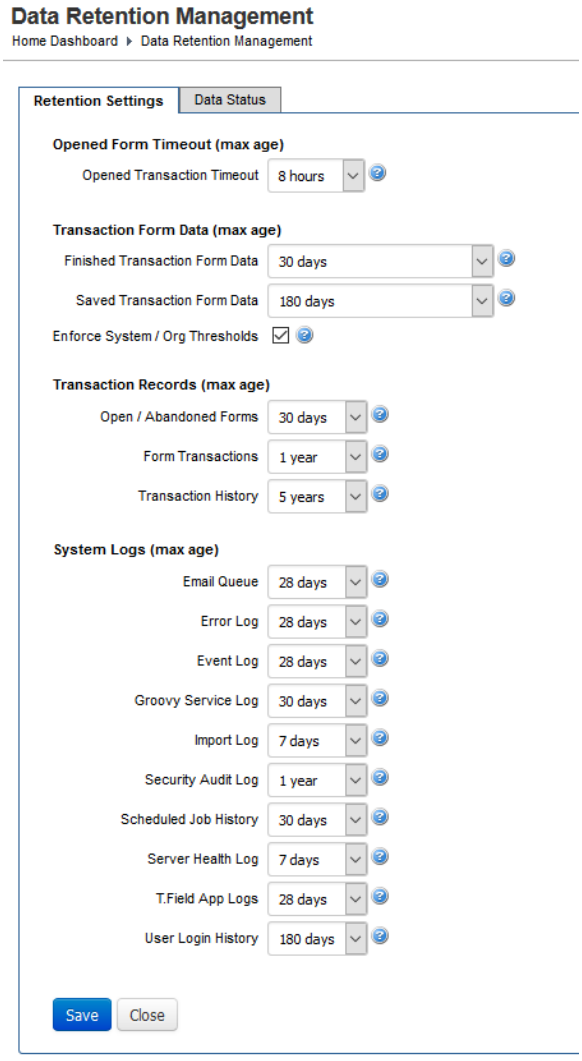
If John accidentally closes the browser window while completing the form and is unable to access the form, he may call up your customer service staff to retrieve his partially completed form. Staff with access to the Operations menu in Transaction Manager are able to access the form's URL and send it to him after verifying his identity.



When John finally submits the form, the transaction will no longer be visible in the 'Saved Transactions' Page but will now be found on the Operations > Form Transactions screen.



Note: The data from the partially completed form is in stored in Transaction Manager for the period specified in System > Data Retention Management.



This may be overwritten at Organization level if appropriate.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- How to launch one form from another, with prepop
- Collaborative form completion with anonymous users (form sharing)
- Addresses

How do I set sequential or strict navigation in my form Wizard?

Unknown macro: 'redirect'

There are a number of policies you can set in your Form Wizard to allow users to continue or navigate forward or backwards within the form. There are three options you can make for your Wizard

1. **Unconstrained page navigation, validate on submit (default)**

This allows users to navigate through any section on the form, going forwards, backwards or jumping to a different section by clicking the section in the Wizard and fields are only validated on clicking the Submit button.

2. **Sequential page navigation, navigate on page change**

This is what is known as a strict wizard where the user cannot continue to the next section until they have completed all mandatory fields within the current section.

3. **Sequential page navigation, navigate on submit**

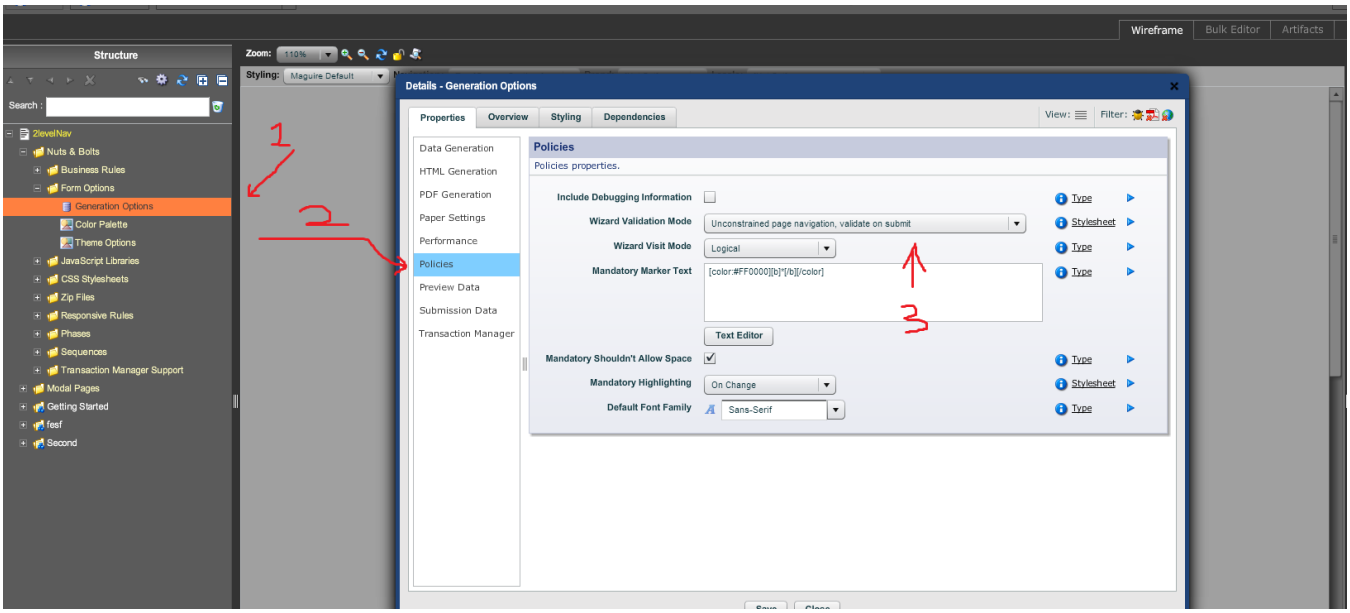
This allows users to navigate from Section 1, 2, 3, 4 and so on and will only validate fields on click of the Submit button.

Steps to change

1. Open your form in Composer and go to Nuts and Bolts > Form Options

2. Open the Generation Options window and go to the Policies section

3. Change the Wizard validation mode



Related articles

- How to launch one form from another, with prepop
- Collaborative form completion with anonymous users (form sharing)
- Setting up 'Leave Page Confirmation' in Composer using JavaScript
- Understanding transaction management in Maguire forms
- Building a Top navigation menu and 2-level navigation menus

How to setup mandatory rules in Composer

Unknown macro: 'redirect'

Mandatory fields

Mandatory fields are used to ensure the end user understands the fields that are required to be filled, and drives validation rules within the form that will prevent form submission until all mandatory fields are filled in.
Fields may be optional if they are not relevant in all end user scenarios, or if it's further information a user can provide. Although forms should not capture useless information, it is important to keep the form simple and avoid unnecessary fields to improve completion and accuracy.

Approaches within a form

All fields mandatory by default

Pros

- This approach can avoid needing to mark every required field individually as mandatory - which makes the form appear simpler to the user. (It can also be argued that all mandatory fields should be marked as mandatory anyway)
- Ensures all fields are filled

Cons

- Making all fields mandatory may force users to enter fields that aren't always necessary
- If the user requires information that they may not have on hand (e.g. an account number) this may impede form completion (Impacts completion)
- Users may enter garbage to get around mandatory validation (Impacts accuracy)

All fields optional by default

Pros

- Increased rate of form submission since users won’t be prevented from submitting a form

Cons

- Users may miss fields that are required

Mix of mandatory and non-mandatory

One of the above approaches can be extended upon by marking

- Only Optional Fields - Optional Fields can be clearly marked to the user by adding (Optional) to the label
- Only Required Fields - Mandatory Fields (aka Required Fields) are by default marked with a red asterisk (*) appended to the end of each field's caption

Pros

- Can ensure all critical information is gathered
- Allow users to not enter non-critical fields (Although consideration should be given; is the field truly necessary in the first place?) to improve submission rate

Best practice is to use a single, consistent approach within a form to avoid confusing users.

Note: Users with accessibility needs may be using screen readers. Need to ensure they can differentiate between fields which are optional and mandatory.

Mandatory Types

Simple

A field can be mandatory/not mandatory under all circumstances.

Conditional

A field can be mandatory under particular conditions defined via a script or rule/s.

Pros

- This is useful for fields that can be filled but aren't always necessary in all scenarios, unless certain conditions are met

Cons

- In a large form, lots of business rules can slow down the form

Conditional visibility

Mandatory fields may only be visible in certain conditions. When the field is not visible, under the hood it is not mandatory.

Pros

- Keeps the form simple and only collect information when it is actually needed

Cons

- A user may enter data in a visible field, break the visibility condition and then be confused why the data that was captured is no longer visible

Application to Fields and Forms

Sample form: https://tm.demo.avoka.com/finance/servlet/SmartForm.html?formCode=mandatory-fields3

Checkboxes

At least one checkbox is mandatory (e.g. you must select at least one product category, or agreeing to Terms and Conditions)

Cons

- If a checkbox on its own is mandatory (e.g. Terms and Conditions), doesn't mean the user will necessarily have read and understood why they're checking the box to avoid an error message.

Radio Groups

There are two approaches to ensuring a user provides a value in a radio group

Mandatory Rule

A mandatory rule may be applied to the ‘radio button group’ widget to ensure a value is selected. *Note*: You will need to manually apply a mandatory marker to the radio button text (see ‘Help Text’ below).

Pros

- User must consciously select a value

Preselected Value

A value can be preselected in which case the user can only change the selection and never deselect.

Pros

- In this case, making the radio button mandatory isn't necessary because the criteria for being mandatory has been fulfilled i.e. a value is already selected
- If a particular value is commonly selected, preselecting this may help the user fill out the form quicker

Cons

- User may accept the preselected value without properly considering it

Dropdown Lists

A dropdown list can be made mandatory to ensure the user selects an option from the list.

Note: In PDF, Dropdowns can have an initial value of null or un-selected.

Mandatory Blocks

Fields can be contained within a Mandatory Block. This is useful when at least one field is mandatory (e.g. you must enter either your home phone or your mobile number)

Mandatory marker customisation

The mandatory marker visible on each field can be customised via Nuts & Bolts > Form Options > Generation Options > Policies > Mandatory Marker Text.

A mandatory marker can also be omitted from a mandatory field by the 'Don't Show Mandatory Marker' option in the field's advanced properties.

Mandatory Highlighting

When the form filler does not complete a mandatory field, the field is highlighted. This colour can be customised using Nuts & Bolts > Form Options > Theme Options > Field Errors > Field Error Background Color [HTML].

You can also determine when highlighting occurs by updating Nuts & Bolts > Form Options > Generation Options > Policies > Mandatory Highlighting. The 3 options are On Visit, On Change and On submit/navigate.

Help Text

The styled mandatory marker can be included in labels/rich text fields using the tag

```
${options:policy.mandatory.marker}
```

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Hidden fields
- Addresses

How to setup visibility rules in Composer

Unknown macro: 'redirect'

Block visibility rules

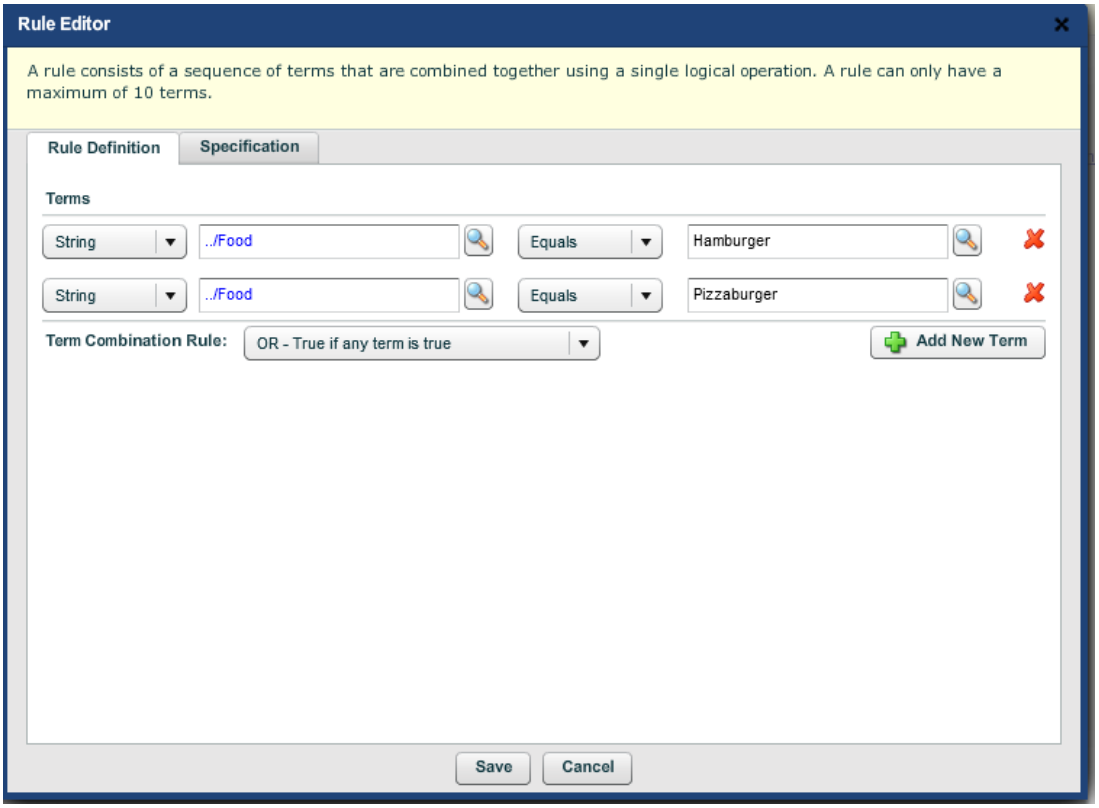
The most common use of visibility rules is to change the information presented in a form as the user enters information.

Sample form: https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=hidden-fields3

The sample contains a dropdown list called ‘Food’ with the options ‘Pizza’, ‘Hamburger’, and ‘Pizzaburger’. When a selection is made the form shows additional questions based on the value selected in the dropdown.

This is achieved using visibility rules on the secondary questions. Visibility rules are configured through the ‘Rules’ menu, under ‘Visibility Rule’:

The ‘Burger Ingredients’ question has the following visibility rule configured:



This will show the block if either ‘Hamburger’ or ‘Pizzaburger’ is selected.

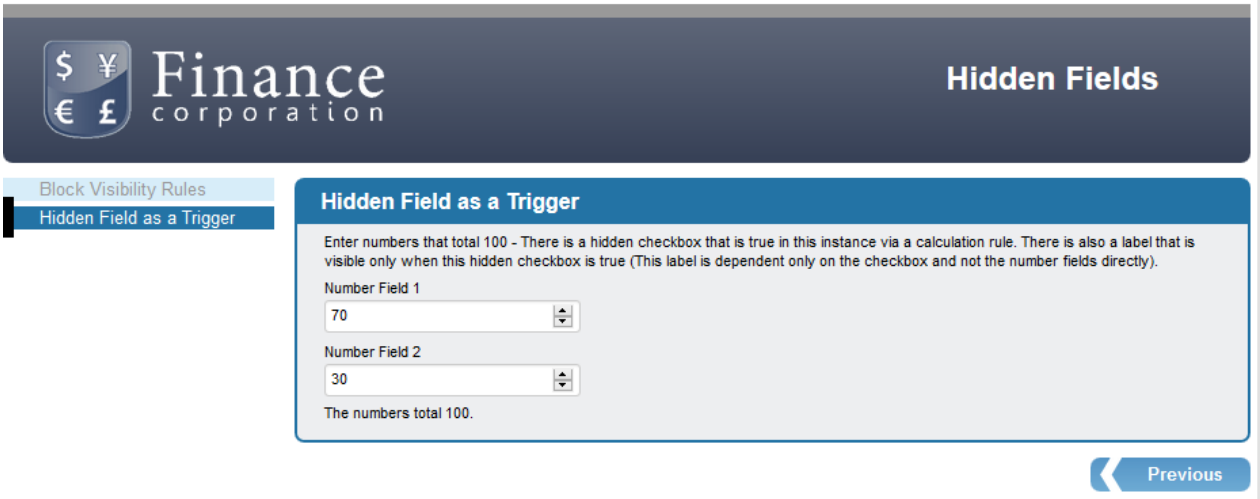
This could also be achieved using code, using the following:

> *return ({Food} == “Hamburger” || {Food} == “Pizzaburger”);*

Hidden field as a Trigger

Sample form: https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=hidden-fields3

Visibility rules can also be used in a variety of other ways. The sample form contains an example of a hidden field used as a trigger to display other objects in the form. It has a checkbox, with the visibility rule set as ‘Never visible’. The checkbox has a calculate script, which sets the value to ‘true’ if two number fields contain values that add up to exactly 100. When the checkbox is ‘true’ additional text appears on the form informing the user that the values entered add up to 100.



Using ‘hidden data fields’ in this way can make visibility, or other data driven rules in the form far less complicated and easy to maintain.

Note: When an object has its visibility rule set to ‘Never visible’ it will not show in the wireframe. The object will only be accessible through the ‘Structure’ section to the left of the wireframe.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Tabular list from dynamic data
- Static Dropdown list

Building a Top navigation menu and 2-level navigation menus

Unknown macro: 'redirect'

Introduction

This article describes the use of the Top level navigation menu as well as 2-level menus in the Maguire theme.

When to use a Top level navigation?

Top menus are more modern and make better use of screen real estate than left menus, and have been made the default option in the Maguire theme.

There are, however, several things to keep in mind when using a top menu:

- A form with more than 5 or 6 pages will not suit a top menu due to horizontal space considerations.
- If the form is large, consider consolidating multiple related pages into a single page to limit the number of pages to 5 or 6 max.
- From a user perspective, seeing a long list of sections can be overwhelming and discourage users from completing a form. Version 4.0 has 2 level menus where a group of level 1 sections can be grouped.
- A form with only 2 sections does not typically warrant a menu and consideration should be given to consolidating the form into a single page or breaking is up into 3 or more pages.

Building a Top navigation menu

1. Create a new form.
2. Select the Maguire Form Template
3. Under Navigation, **if you wish to use 2-level menus, you must select Top [Groups+Sections]**, otherwise use Top [Groups]

When creating Section Level 1's, they will be added to the Top menu by default.

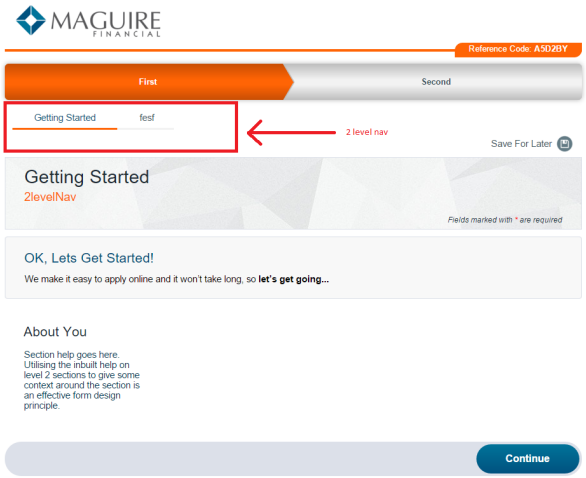
To remove them, open the Section Level 1 property editor > Properties > Wizard,> untick the 'Create Navigation Menu Entry' property. *****

*****There is an issue in Composer 4.01 and prior where if the Section Level 1 is hidden and has 'Create Navigation Menu Entry' unchecked, mandatory fields will not be skipped during validation, rendering users unable to continue with the form. If you are experiencing this issue, upgrade your version of Composer or keep this option checked.

Building a 2-level navigation menu

To group several Section Level 1's under one menu item, open the Section Level 1's property editor > Properties > Wizard and **provide the same Section Group Name to group various section level 1's together**.

To change the name of the item created in the second level of the menu, change the property 'Wizard Menu Text' to suit.



Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- How to launch one form from another, with prepop
- Collaborative form completion with anonymous users (form sharing)
- Addresses

How to build a repeating section in a Composer form

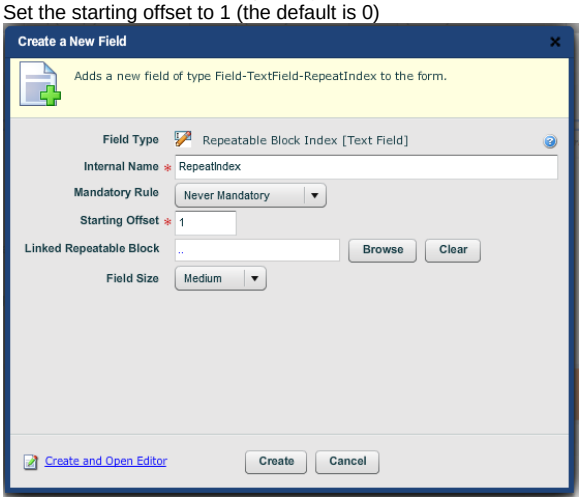
Unknown macro: 'redirect'

Compatibility

Since	4.0
Deprecated	

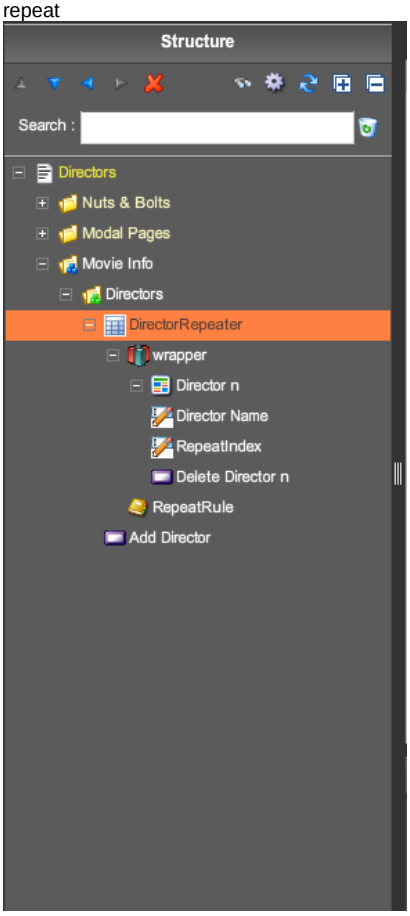
For the purposes of this example, I'll add a list of "Directors", but names and labels are unimportant. Important – Avoid the use of Tables in the Maguire template or similar responsive designs. Design for mobile FIRST.

Create the necessary widgets

1. Drop in a repeating block
2. Drop in a simple block. (This will be just used to tweak look and feel and spacing.)
3. Drop a rich text object into the block, label = "Director n". This will be the heading for the block. We'll fix the "n" later.
4. Drop a Text Field into the block, named Director Name.
5. Add a Repeatable Block Index into the block, named RepeatIndex.
    - Set the linked field to the Repeating block
    - Set the starting offset to 1 (the default is 0)
    - 

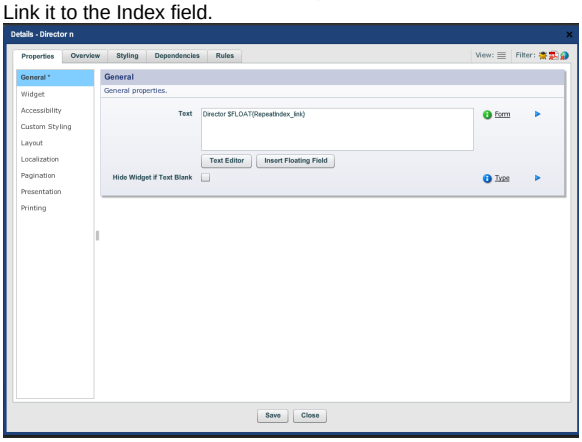
6. Add a Delete Row button into the block, name it "Delete Director n", and link it to the repeat (We'll fix the "n" later).
7. Add a Business Rule – General Purpose to the repeating block.

8. Add an "Add Row" button outside the Repeating block, name it "Add Director", and link it to the repeat



Customize the Heading

1. Double click on it, and add a floating field to the end of the text.
2. Link it to the Index field.



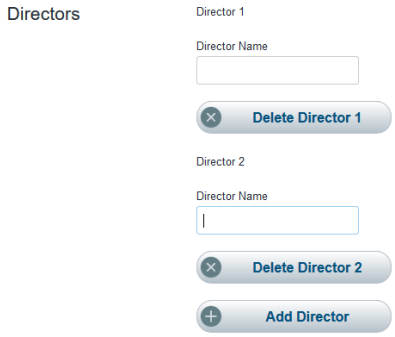
3. You may want to style the heading. (Or you can try using a Section level 2 or 3).

Customize the Delete button

On a small screen, it can become a little bit confusing about which Director is going to be deleted, so let's modify the caption on the button so it's more specific.

1. Double click on the business rule.
2. Change the rule to Script Based, and click the Edit button.
3. Double click the Delete Director button (because we need it's node id to update its caption) and the Repeat Index field (because we need its value to know what the current index is).
4. Use the variables that get inserted for you to write the following script:

sfc.setCaptionText({DeleteDirector}, "Delete Director " + {RepeatIndex});

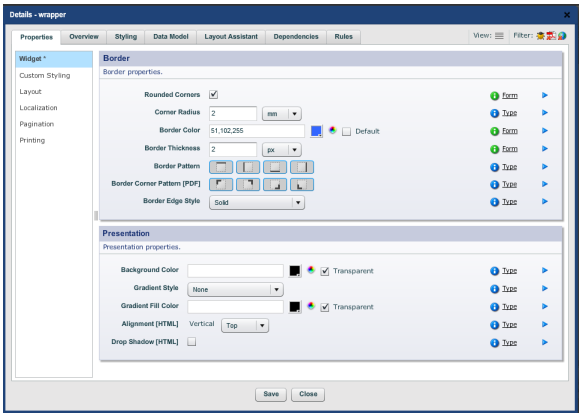


Alternately, you could get a little more fancy, and get the button caption to be either a number or a name (if one has been entered), with the following script:
if ({DirectorName}+ "" == "") { sfc.setCaptionText({DeleteDirectorn}, "Delete Director " + {RepeatIndex});
} else { sfc.setCaptionText({DeleteDirectorn}, "Delete Director - " + {DirectorName}); }

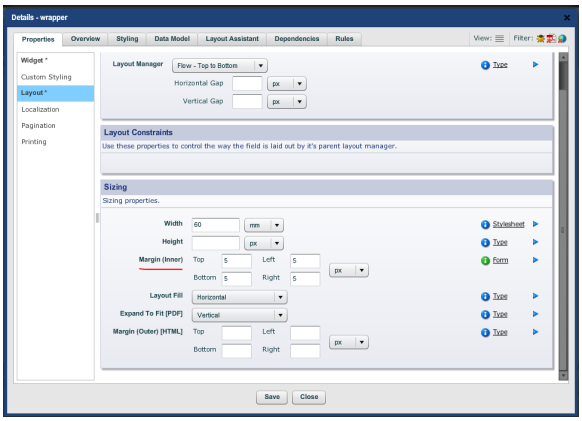
Styling

You may want to have a pre-defined style block to "gather" the items in the repeat together visually. There are many ways to do this, but a simple start is:

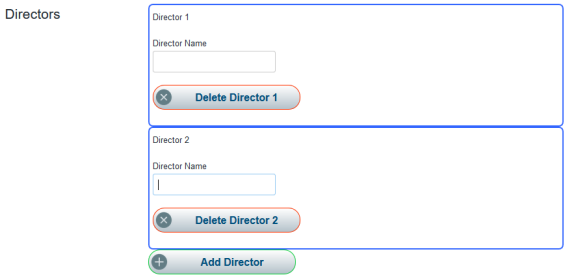
- A coloured border



- Some padding and margins



- You may also want to style the Add/Delete buttons.



Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Tabular list from dynamic data
- Static Dropdown list

Configuring confirmation emails upon Save and Submit

Unknown macro: 'redirect'

Compatibility

Since	4.1
Deprecated	

Forms in Transaction Manager can automatically send confirmation emails upon successful submission. Confirmation emails are configurable and customisable, with the ability to specify the recipient, subject, content, and HTML template.

The example scenario will be an **Event Registration** form. Upon submission respondents will receive a personalised confirmation email with the following content:

Hi Jane,

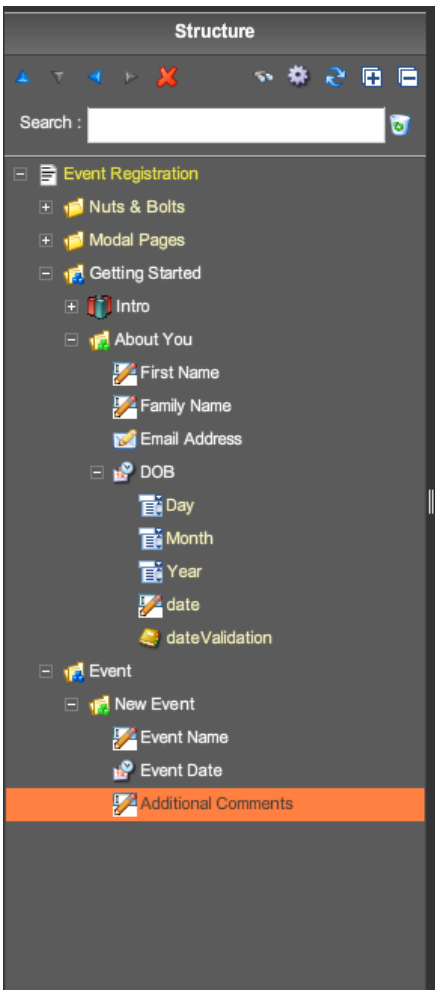
Thanks for registering to attend our event 'Rapid form design, in Composer 4.1', we look forward to seeing you on the 23rd of September, 2015.

Regards,
Avoka Events Team

Part 1 – Form Setup

Create a form matching the example below, the steps required have been omitted for brevity. As a minimum the form should contains mandatory fields for first name, email, event name and event date.

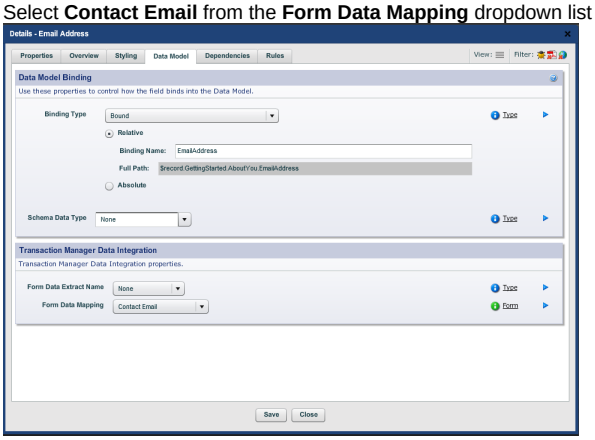
Example:



Once created, configure the form using the steps provided to enable submission data access from the email template. This is done using **Form Data Extracts** and **Form Data Mapping**, both of which are found on the **Data Model** tab when editing the properties of a field in Composer.

To get started, Transaction Manager needs to know which field contains the destination email address for confirmation messages. Set this by doing the following:

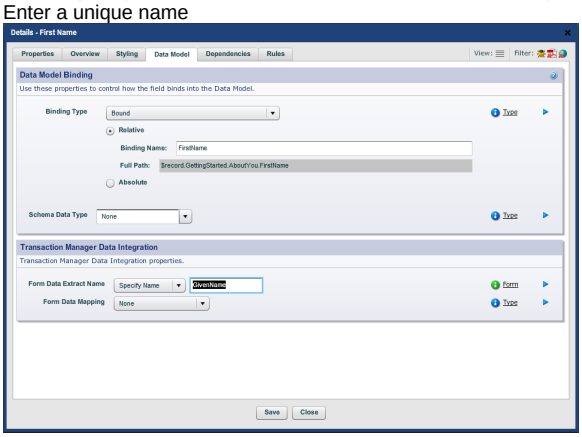
1. **Edit** the properties of the email field
2. Navigate to **Data Model** > **Transaction Manager Integration**
3. Select **Contact Email** from the **Form Data Mapping** dropdown list



4. **Save** the properties

Next, setup **Form Data Extract Names** for each of the fields required in the email message (name, event name, event date). Repeat the following for each field:

1. **Edit** the properties of the field
2. Navigate to **Data Model** > **Transaction Manager Integration**
3. Select **Specify Name** from the **Form Data Extract Name** dropdown list
4. Enter a unique name



5. **Save** the properties

Part 2 – Configuring Confirmation Emails

Once the submission data has been configured (Part 1), there are two ways you can configure automatic email confirmations. The options are:

1. Form configuration
2. Transaction Manager configuration

Both of these options have their own strengths and weaknesses. While configuring the settings in the form can reduce the amount of work required on TM, at the time of writing it doesn't offer all of the options available in TM.

Option 1: Form Configuration

1. Edit the properties of the form element in the **Structure** pane (the highest level element).
2. Navigate to **Properties** > **Form** > **User Experience**
3. Tick the **Auto-Send Confirmation Emails** checkbox
4. Save the properties

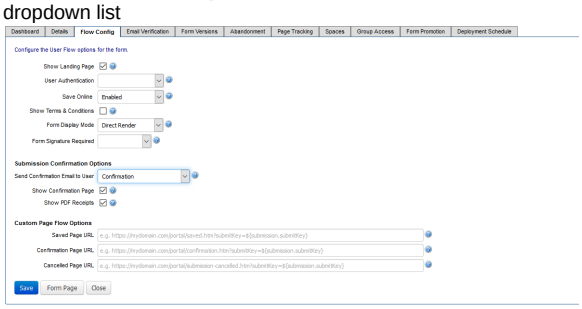
5. Publish the form

Option 2: Transaction Manager Configuration

The form should be ready for publishing and configuration. Publish the form to **Transaction Manager**.

Configure the form on TM to send confirmation emails:

1. Navigate to your newly published form's **Dashboard**
2. From the **Dashboard** click the **Flow Config** tab
3. Under **Confirmation Options** select **Confirmation** from the **Send Confirmation Email to User** dropdown list



4. Click **Save**

Your form is now sending confirmation emails containing the default content upon submission. The default content looks something like the image below, although it has likely been customised for your organisation's **Portal**:

Test this functionality by submitting your published form with your own email address in the email field.

Part 3 – Customising Email Messages

The content of confirmation emails, by default, is maintained at the **Portal** level. All forms using a specific portal will typically have the same confirmation email. Using **Properties** this can be overwritten at either the **Organisation** level or the **Form** level.

The properties used to configure confirmation emails are:

Property Name	Description
Email Form Receipt Message	The body of the email. This content is generated using the Velocity Templating Engine and allows for the injection of form values.
Email Form Receipt Subject	The subject of the email. This content is generated using the Velocity Templating Engine and allows for the injection of form values.
Email Template HTML	The HTML template for ***All*** emails for this Portal/Organisation/Form

Properties can be found in the following locations, depending on the level at which they are being configured.

Portal

System > **Portals** > *Your Portal Name* > **Properties** tab

Organisation

Forms > **Organisations** > *Your Organisation Name* > **Properties** tab

Note: *Organisations do not contain all properties by default, you must specifically add the properties you wish to overwrite by clicking* ***New*** *and selecting the property from the* ***Property Type*** *dropdown list.*

Form

Forms > **Forms** > *Your Form Name* > **Form Versions** > **Edit** *(current version)* > **Properties** tab

Note: *Forms do not contain all properties by default, you must specifically add the properties you wish to overwrite by clicking* ***New*** *and selecting the property from the* ***Property Type*** *dropdown list.*

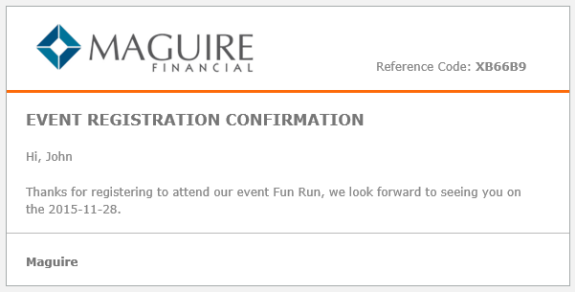
To create the email message for the **Event Registration** form, specific form data is embedded in the email message. The data being embedded is specific to this form, meaning the properties should be overwritten at the **Form** level.The following explain how to change the required property:

1. Navigate to **Forms** > **Forms** > *Your Form Name* > **Form Versions** > **Edit** *(current version)* > **Properties** tab
2. Click **New**
3. Select **Email Form Receipt Message** from the **Property Type** dropdown list
4. A **Value** textbox will appear, containing default content. Delete this content.

   Note: *The content of the email message allows for standard HTML formatting, hence the use of and other tags.*
5. Enter the new content. For example:

Code
Hi, ${formDataMap.GivenName}   Thanks for registering to attend our event ${formDataMap.EventName}, we look forward to seeing you on the ${formDataMap.EventDate}.
6. **Save** the property

The steps described result in an email like the following:



Note: *to access form data in the email message, subject, or HTML template, use the notation* **${formDataMap.***NameOfDataExtract***}***. This notation is based on the Apache Velocity Engine, for more information see http://velocity.apache.org/.*

Related articles

- Task Assignment
- Collaboration Jobs - Separate out emails from Task Assigns using multiple actions
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer

How to include 'Please Select...' in a Mandatory Dropdown list

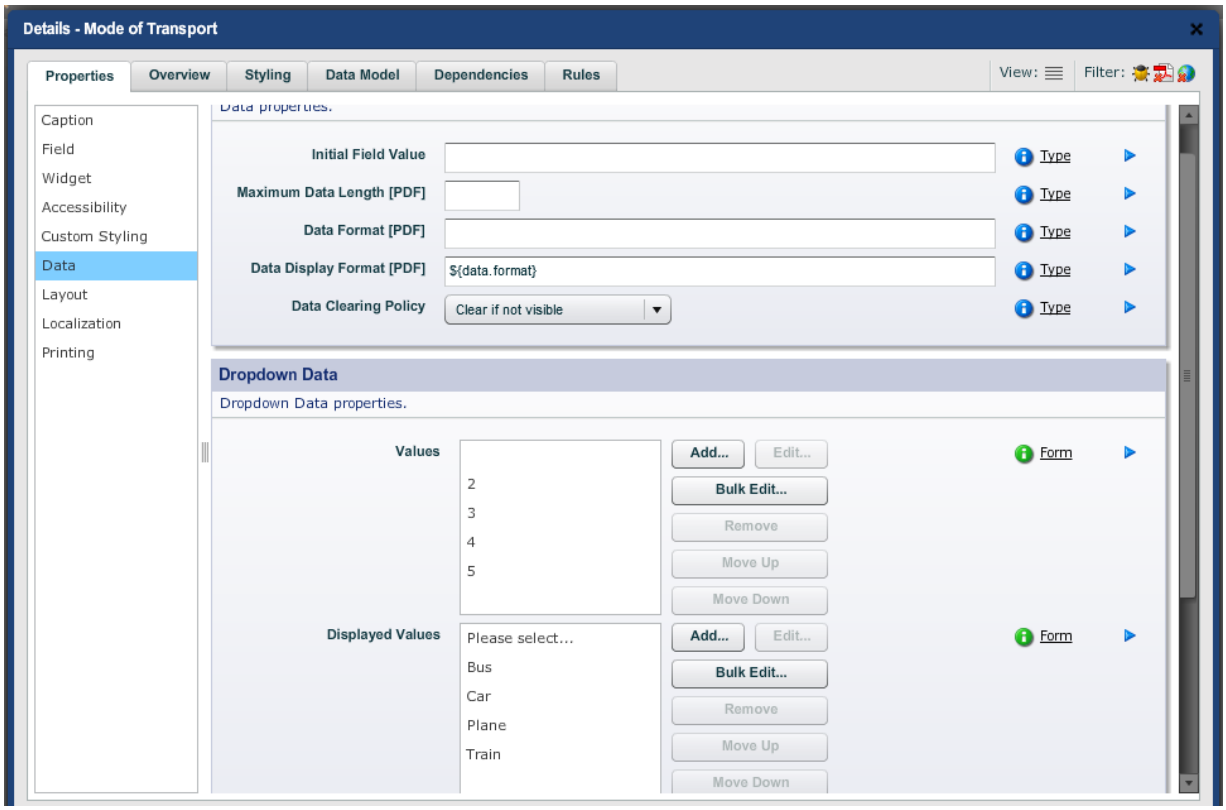
Unknown macro: 'redirect'

In Composer, the capability to add 'placeholder text' to a dropdown list is not a standard feature.

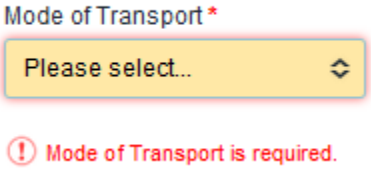
Therefore to add informational text which is not a valid option e.g. 'Please select...' to your mandatory dropdown list, proceed with the following steps:

- Make the field mandatory.
- Add the “Please Select...” display value with a corresponding 'blank' data value.

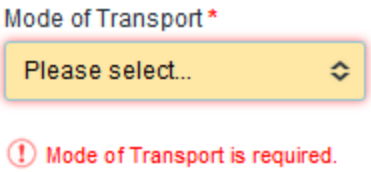
Resulting Dropdown Data:



- The following screen shot shows the resulting dropdown list in the form:



- Mandatory validation will ensure you to select some other value than “Please select...” as it relies on data value.



Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Tabular list from dynamic data

Using a Processing Spinner for lengthy operations in Composer forms

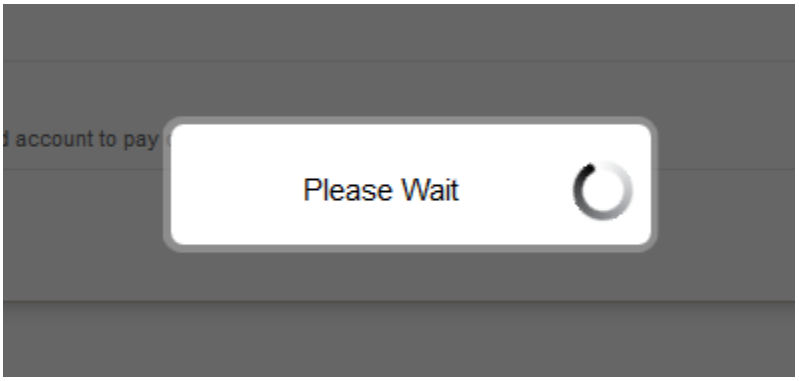
Unknown macro: 'redirect'

Introduction

This article describes how a form author could add a spinner/progress bar when the form is performing a lengthy operation such as a dynamic data lookup (e.g. identity validation).

Using the Composer Built-in Progress Spinner

If you are happy to use a generic progress spinner you can use the Composer built-in feature which looks like this:



To activate and deactivate the spinner you can use the following JavaScript to call the internal Composer function.

Showing and hiding the built-in progress spinner

```
// Show the progress spinner
  sfcInternal.av_html_showProgressWindow();

// do stuff

// Hide the progress spinner
  sfcInternal.av_html_hideProgressWindow();
```

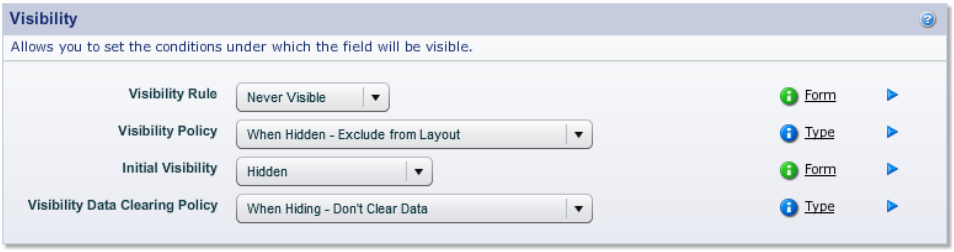
Ensure that your script calls the hide function on all scenarios otherwise you may be left with a progress spinner that won't go away.

Implementation

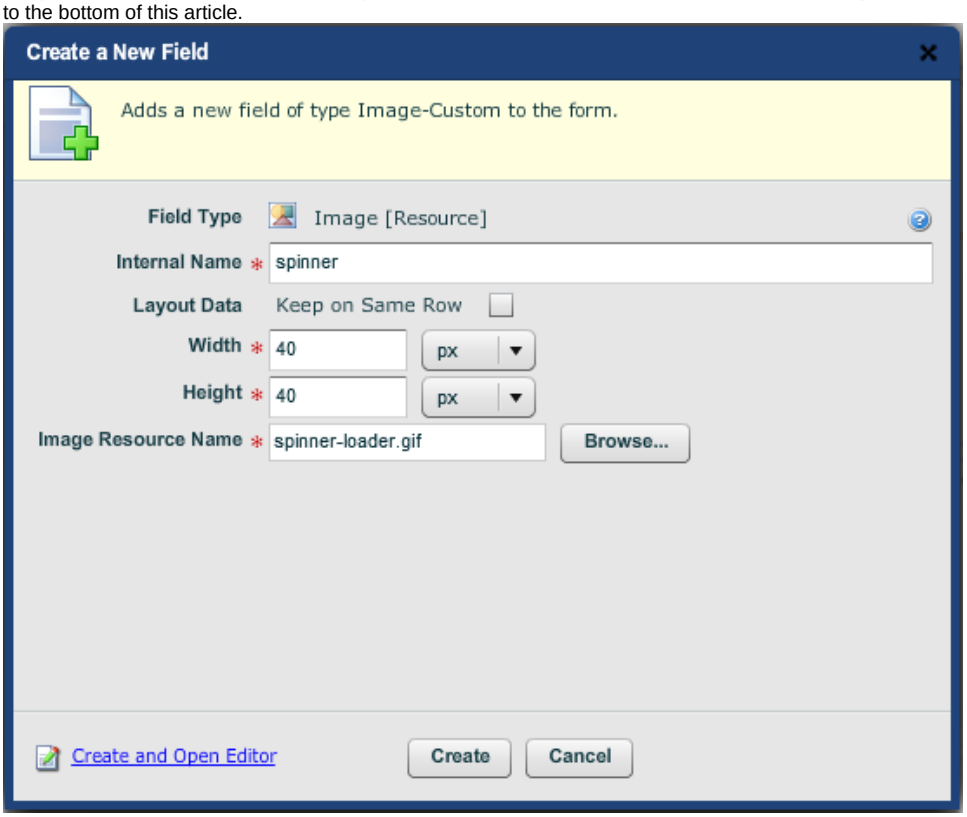
To create your own nicely styled "spinner" or "Please Wait" progress notification upon form submit, there are 2 steps as follows:

Step 1: Set up the form structure

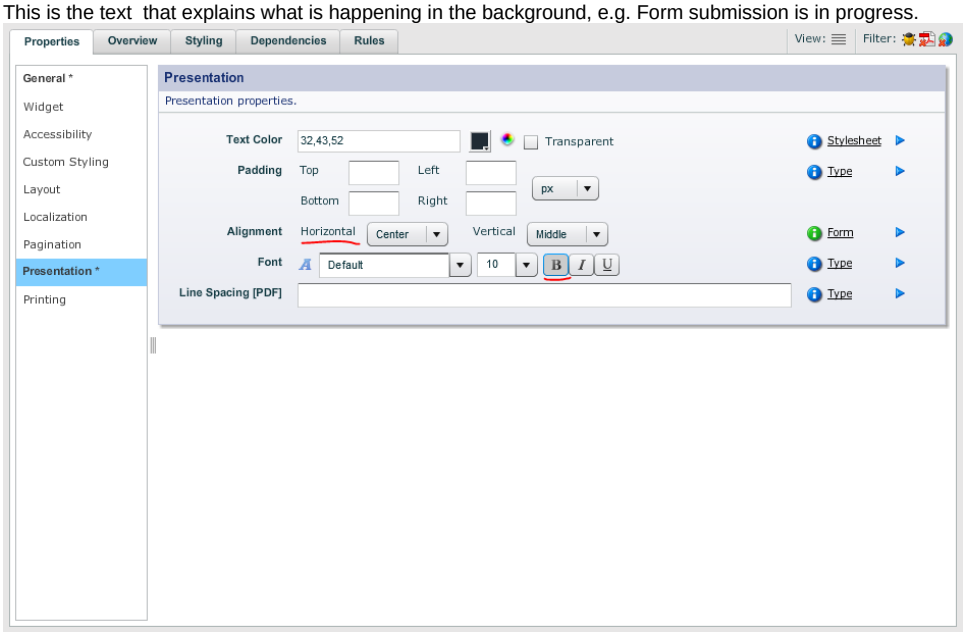
1. Insert 2 Empty Blocks into the form i.e. 1 block for the gray-out background (SpinnerOverlay) and another one for the spinner and rich text field (SpinnerBlk). Change the properties as follows: Set the Visibility Rule of both blocks to 'Never Visible' and the Initial Visibility to 'Hidden'.



2. Insert an Image Resource object and a Rich-Text field into the SpinnerBlk and apply the following property changes:
   - Image Resource - set the Width and Height as follows and set the Layout Fill to Horizontal. The spinner-loader.gif image file is attached to the bottom of this article.



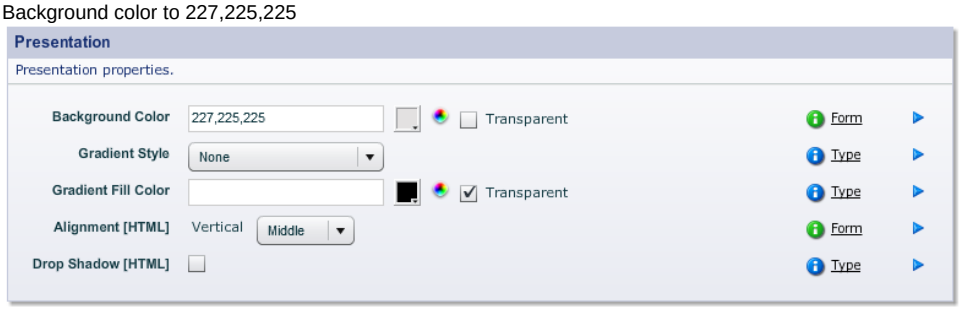
   - Rich-Text field - Set the Horizontal Alignment of this field to Center and Font Bold.
     This is the text that explains what is happening in the background, e.g. Form submission is in progress.



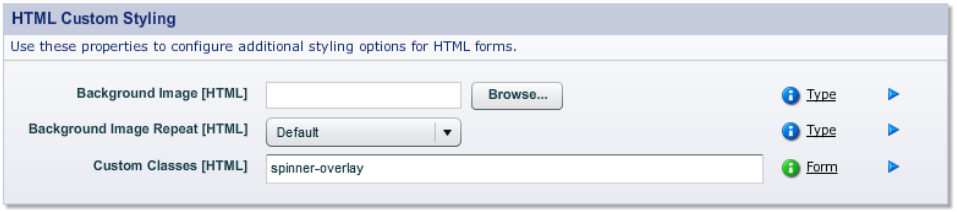
3. Add a 'CSS snippet' widget to Nuts & Bolts > CSS Stylesheets with the following style:
   - .spinner-overlay {
     position: fixed;

```
  width: 100%;
  height: 100%;
  left: 0;
  top: 0;
  z-index: 100;
  opacity: 0.8;
}

.spinner-blk {
  position: fixed;
  top: 50%;
  left: 0;
  margin-top: -9.5px;
  /* half height of the spinner gif */
  text-align: center;
  z-index: 111;
  overflow: auto;
}
```

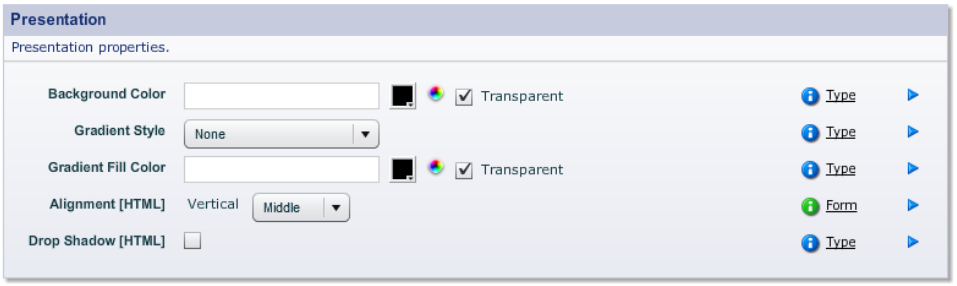
4. Apply the following property changes to block SpinnerOverlay:
   - Alignment [HTML] to Middle
   - Background color to 227,225,225



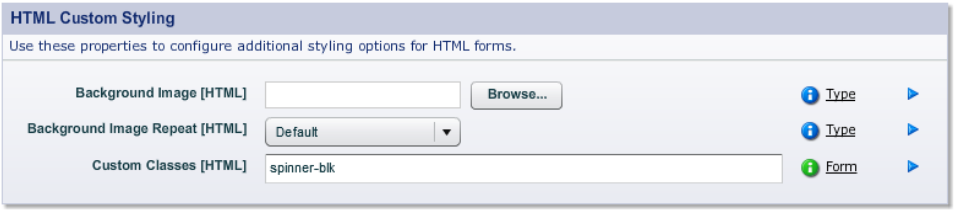
   - Custom Classes [HTML] to spinner-overlay



5. Apply the following property changes to block SpinnerBlk:
   - Alignment [HTML] to Middle



   - Custom Classes [HTML] to spinner-blk



Step 2: Showing the spinner

On Form Submission

In the Structure pane, select 'Show Internal Fields'

Open the property editor of the Submit button > Rules > Click > Click action, and add the following code to display the progress bar:

```
sfc.setCaptionText({txtCaption}, "Please wait, form submission is in progress."); // set the caption text
sfc.updateVisibility({SpinnerOverlay}, true, 'exclude'); // show the gray-out background
sfc.updateVisibility({SpinnerBlk}, true, 'exclude'); // show the caption text and progress bar
```

Using a Dynamic Data Lookup

Add the following line of code on Dynamic Data Lookup button:

1. On the Preinvoke rule, add the following lines of code to display the progress bar:

   ```
   sfc.setCaptionText({txtCaption}, "Please wait, data verification is in progress."); // set the caption text
   sfc.updateVisibility({SpinnerOverlay}, true, 'exclude'); // show the gray-out background
   sfc.updateVisibility({SpinnerBlk}, true, 'exclude'); // show the caption text and progress bar
   return true;
   ```

2. On Success/Failure Notification rule, add the following lines of code to hide the progress bar:

   ```
   sfc.updateVisibility({SpinnerOverlay}, false, 'exclude'); // hide the gray-out background
   sfc.updateVisibility({SpinnerBlk}, false, 'exclude'); // hide the caption text and progress bar
   return true;
   ```

- Spinner Gif :

Related Articles

- Managing multiple service endpoints and credentials for external service calls
- Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Including Web Fonts in Composer forms

Creating and updating a custom block in Composer

Unknown macro: 'redirect'

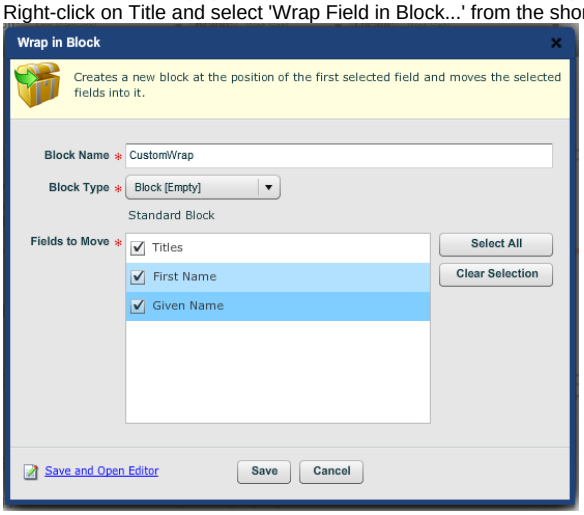
Compatibility

Since	4.0
Deprecated	

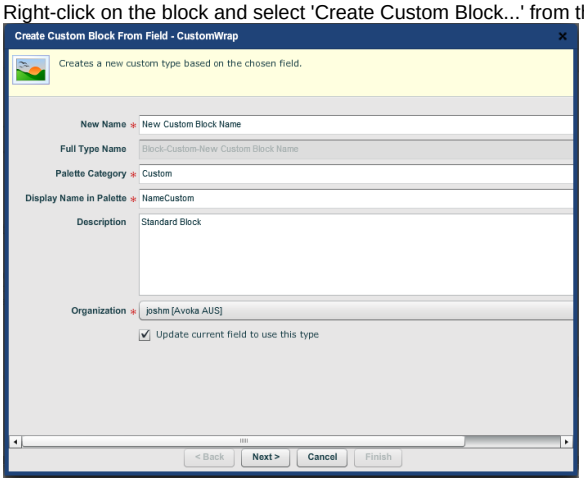
To illustrate the concept, I am going to create a 'Personal Details' custom block:

Example:

- Create a new form using the Maguire template.
- Add the 'Dropdown List [Titles]' to the 'Getting Started > About You' section.
- Add 2 Text fields called 'First Name' and 'Family Name' on the same row as Title and spreading out to fill all remaining space on the line (hint: use Field Size: Fill)
- Right-click on Title and select 'Wrap Field in Block...' from the shortcut menu



- Right-click on the block and select 'Create Custom Block...' from the shortcut menu



Palette Category refers to where the custom block will be located in widget palette once created.

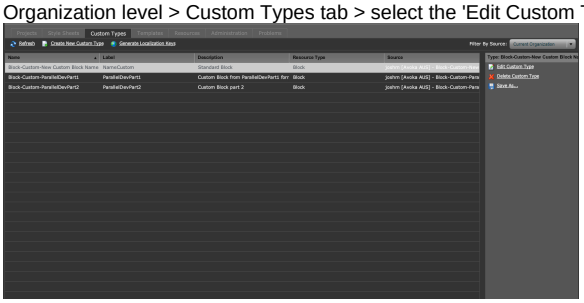
Note: Because the check box 'Update Form to use this Type' is selected, you will be unable to use this form to update your custom block. If you did, you may get an error when you save the custom block:

The custom block is stored at the Organization level > Custom Types tab so it may be reused by multiple forms across multiple projects within the organization if required.

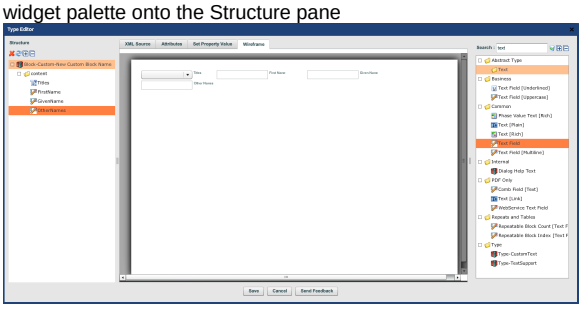
- Test your new custom block.

Updating a Custom Block

- Organization level > Custom Types tab > select the 'Edit Custom Type' button



- You are able to update the Custom block using XML or you may drag and drop widgets from the widget palette onto the Structure pane



- Once you have saved your changes in the Custom Type editor, test your changes.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Tabular list from dynamic data
- Static Dropdown list

Creating and updating a custom widget in Composer

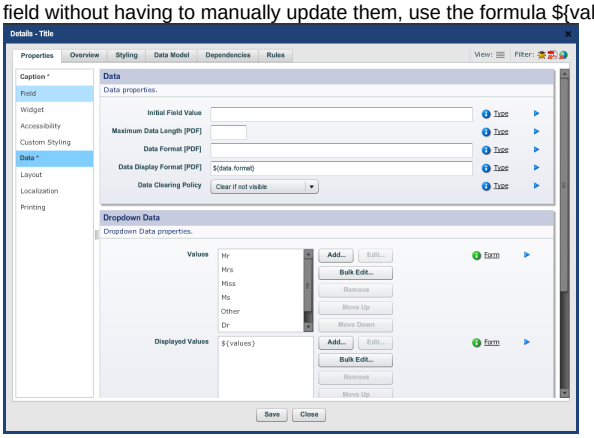
Unknown macro: 'redirect'

Compatibility

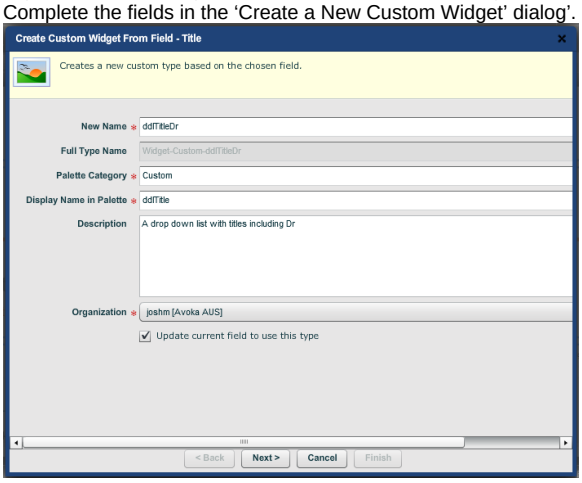
Since	4.0
Deprecated	

To illustrate the concept, I am going to update the standard 'Dropdown List [Titles]' widget to add the value ‘Dr’.

- Create a form.
- Include the ‘Dropdown List [Titles] widget’ in your form with a caption of 'Title'.
- Update the property editor of Title > Properties > Data > Dropdown Data > Values to include Dr.
  Note: this widget has the same values in the ‘Values’ property and the advanced property ‘Displayed Values’. The ‘Values’ field must contain the same number of values as the 'Displayed Values'. So if you want the form filler to see the same values as are in the ‘Values’ field without having to manually update them, use the formula ${values}.

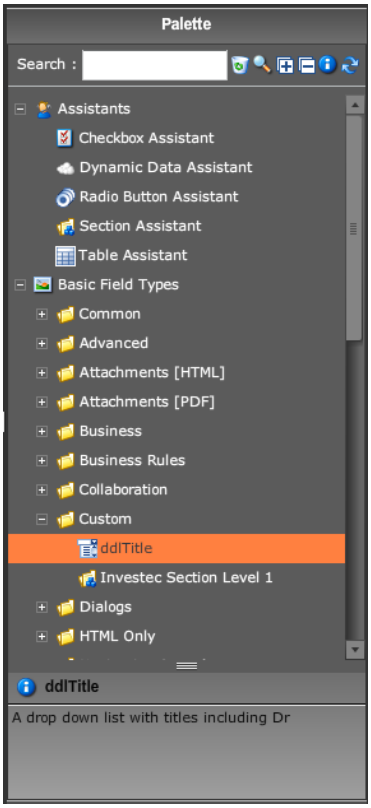


- Save the form.
- Right click on the widget and select ‘Create custom widget’.
- Complete the fields in the ‘Create a New Custom Widget’ dialog’.



Note: Because the check box 'Update Form to use this Type' is selected, you will be unable to use this form to update your custom block. If you did, you may get an error when you save the custom block:

- The custom widget will be visible in the 'Custom' category in the widget pallette.



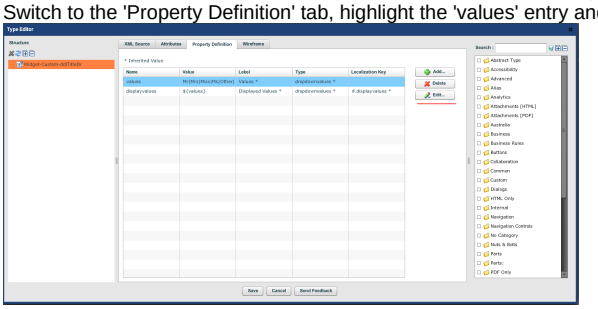
Note: all custom types are stored at organization level so that they maybe reused.

- Save and close the form.
- Test the custom widget by building a form with it included.

Updating a custom widget

I am going to include an additional value 'Prof'.

- At organization level, select the 'Custom Types' tab.
- Select the 'Edit Custom Type' button (found in the 'Actions' column).
- Switch to the 'Property Definition' tab, highlight the 'values' entry and select the 'Edit' button



- Select the 'values' property in the 'Name' field and update the 'Value' field to include the title 'Prof'.
- Rerun your form to test for the updated 'Title' values.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Tabular list from dynamic data
- Static Dropdown list

Loading a form without the wireframe in Composer

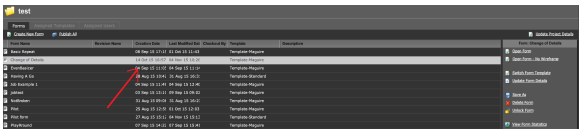
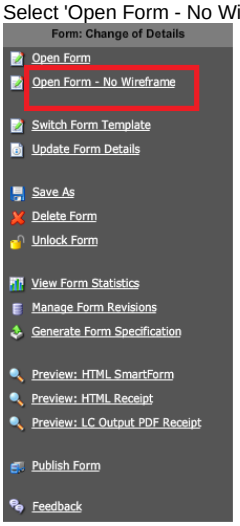
Unknown macro: 'redirect'

From Composer version 4.0 it is possible to open a form without the wireframe being presented. This is particularly useful if you are making minor changes quickly, or not completing visual changes and would like to avoid waiting for the processing time of the wireframe loading.

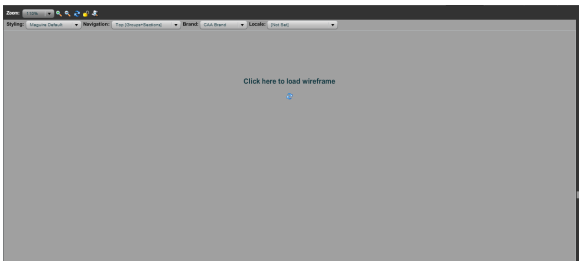
Compatibility

Since	4.0
Deprecated	

Method

- Select your project in the left hand menu.
- Click once to select your form in the right hand menu - the Action palette is now visible.
- 
- Select 'Open Form - No Wireframe.
- 

Result



Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Addresses
- Address Block Samples

Creating custom Composer templates

Unknown macro: 'redirect'

Templates are used as the starting point for a form. Templates are used to control the following:

- List of available technologies (skins that can be used to preview the form)
- List of available variants (i.e., HTML and PDF)
- List of available style sheets
- TPS file (which in turn contains master pages)
- Starter Content
- Properties that are applied at the form level (and may be requested in the form creation wizard)
- Standard libraries
- Paper size handling
- Locale support (later)

Creating a new template

The easiest way to create a new template is to copy an existing one as a starting point, and modifying it to suit your needs. To do this, you will need to export the latest library:

- Log into Composer as an administrator
- Go to ‘Libraries’, and right click the latest release, and select ‘Export…’
- Save the resulting zip file, extract it, and navigate to the ‘Templates’ directory
- Copy the ‘Template-Standard.xml’ file, and rename it to something meaningful

Editing a template

Within the template xml file, in the <formtemplatename element, change the following properties:

name: If two templates in the same organisation have the same name, only one will be visible.

label: Change the label to something meaningful especially when duplicating an existing template as they will both show in the template selection list with the same name

NB: the <property> items are questions that are asked when creating a new form.

Scroll down and delete all stylesheets expect one that is closest to the style you would like for the overall look and feel

<starterContent> - This is optional in the template and when a user creates a form. This content the user can delete or modify regardless. Objects in the <content> section cannot be deleted but the user can set attributes on the objects it contains.

The best way to add content into the <starterContent> section is to create the content in a new form in Composer, then switch to the ‘XML Source’ tab at the top right, and copy the desired elements from the xml into the template.

Re-Importing a template

To import a new template, you must export the organisation that you wish to import the template into, save the template into it, then re-import it:

- Log into Composer as an administrator (you will need to speak to your Avoka Representative)
- Navigate to ‘Projects’, right click the organisation, and select ‘Export…’
- Save the resulting zip file, and extract it.
- Delete everything except the ‘organisation.xml’ file
- Create a new folder called ‘templates’, and copy the template xml file into it.
- Zip the file. Make sure to zip the direct parent of the ‘organisation.xml’ file. If you selected ‘extract to /(organisation name)’ you will need to navigate into the extracted folder before zipping.
- Go back to Composer, right click the organisation, and select ‘Import…’, and import the new organisation.zip file.

The new template will now be available within the organisation.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)

- Addresses

Parallel development in Composer (multi-part forms)

Unknown macro: 'redirect'

Overview - what's the problem?

When building larger forms, it sometimes becomes desirable to split the form into multiple parts, so that several developers can work on it simultaneously.

One simple way to achieve this is to build different parts of the form using blocks. However, blocks are ideal for smaller groups of fields such as an address block, but are not ideal for larger parts such as an entire section. In addition, the block editor can be quite difficult to use for this purpose.

(Note that you cannot save an entire section as a block.)

Another problem is that when working on large forms, the Composer editor can become slower to use.

The approach outlined in this article solves both of these problems quite elegantly.

Parallel Development

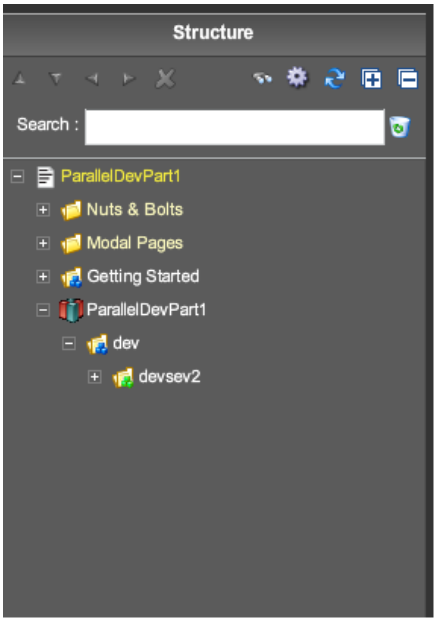
The basic principle of this technique is that you can wrap an entire section (or multiple sections) into a block, save that block as a custom block, and then re-use that block in other forms. Let's go through this step by step.

A sample project is currently located at: https://composer.prod.avoka.com/composer/, Organization: Cookbook, Project: Parallel Development

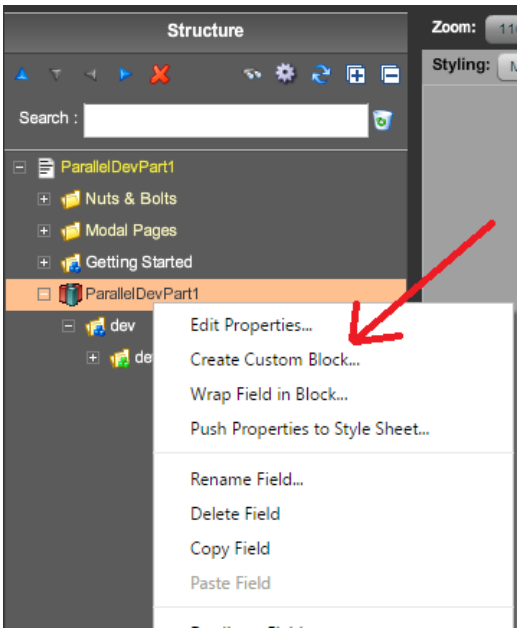
In these steps, we will use specific names - you would substitute these with names that make sense to you.

Part 1

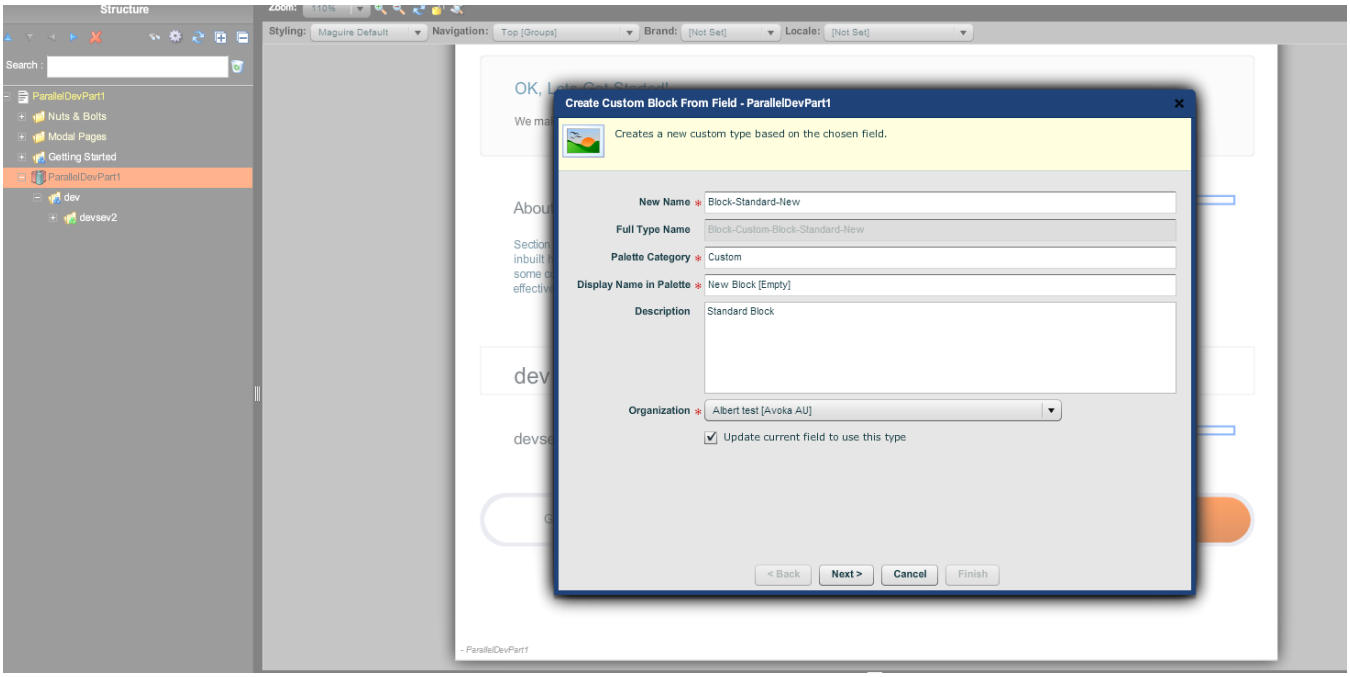
- Create a form called ParallelDevPart1.
- In this form, create one or more sections. In our case, we created Section 1 and Section 2. You can put any normal content into these sections, including business rules.
- Wrap these sections in a block named ParallelDevPart1. (Our convention is to name the block the same as the form, but this is up to you.)



- Then right click, and "Save as Custom Block"



- Name the block appropriately. You must remember this name, as you will need to re-type it next time. Our convention is to name the block the same as the form and the block name, so that all names are consistent.
- Remember to UNCHECK the "Update Form to use this Type" checkbox.



- We put all our parts into a Category called Parts, but this is just a convention that allows us to find our parts conveniently.

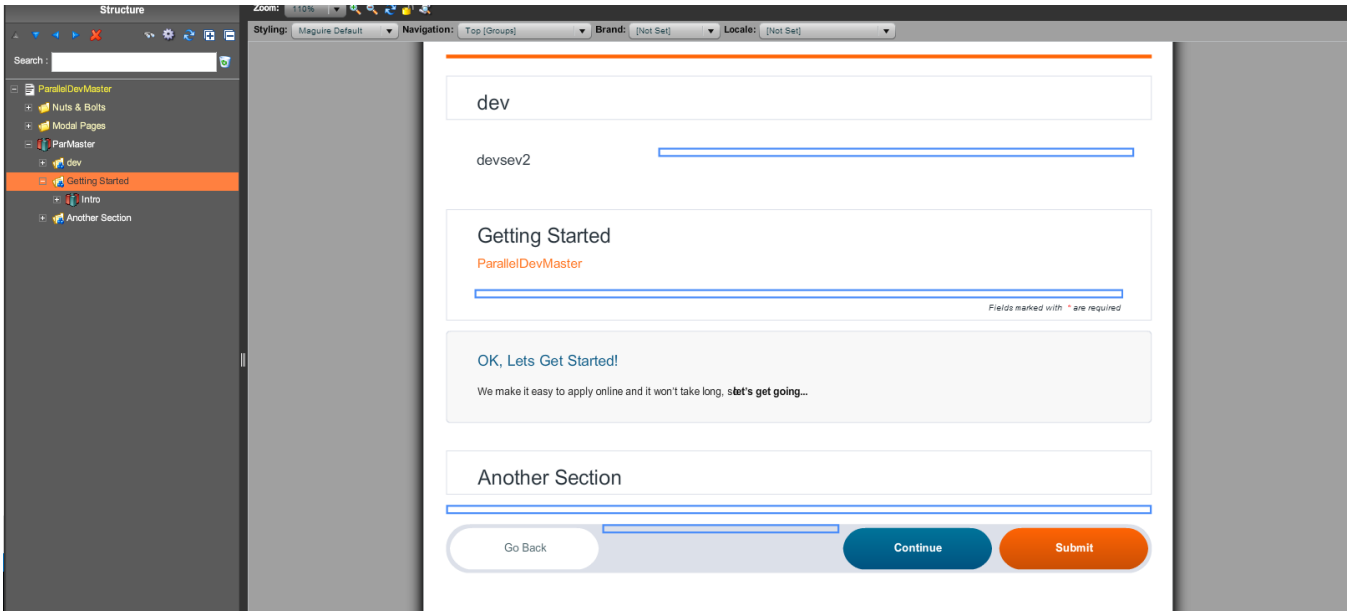
Part 2

Part 2 is structured exactly the same as Part 1. In our example, Part 2 contains a block named ParallelDevPart2, which in turn contains Section 3. We published the block to ParallelDevPart2.

Master Form

- Create a master form, which we called "ParallelDevMaster".
- Into this form, drag the blocks defined earlier into the top level of the form.
- You can add additional content to this form if you want. In this case, we have added another section called "Another Section".

- Even though the Sections are enclosed in blocks, they will still function correctly in the overall form, as you can see below.



What does this achieve?

- Multiple developers can easily work on different parts of the same form at the same time.
- You can still use the usual smaller blocks (such as an address block) within your parts.
- Each part is smaller than the overall form, which makes it simpler to understand and navigate, and also makes the Composer user interface run faster.
- You can use the full Composer editor and wireframe to build, preview, and unit-test your parts.
- You can preview and test your parts using the normal preview capability.
- You can use form versioning to effectively version your parts. (Blocks are not versioned.)
- You can encapsulate most business rules within the parts, but you can also implement cross-part business rules in the usual way (by implementing them in the Master).
- Using this technique, you can also build multiple different forms using the same basic set of parts, assembled in different ways.

Some Tips

- Use a naming convention to help you navigate the different parts. We've proposed a simple naming convention above.
- You must ensure that all your section names are unique. Composer will not enforce this, and your form will break if you have naming conflicts.
- You can build business logic (such as hide/show rules or calculations) into the parts in the usual way.
- You can build any cross-part business logic into the master form. For example, a visibility rule that affects part2, but is based on information captured in part 1, can be built into the Master Form.
- You can also override functionality in any of the parts within the Master form. When you do this, consider whether you should be implementing this functionality in the part itself, or within the form.
- For XML binding, a simple way to organize this is to ensure that each section has a unique binding name (this is the default). This will mean that each section has it's own top-level element in the overall structure.
- Alternately, you may define a consolidated XML seed file that can be used to explicitly bind each field to. Data bindings performed within the parts will be honoured by the master form.

Some warnings

- Don't forget to "Save as a custom block" every time you edit one of your parts.
- If you build multiple forms by assembling parts together, don't forget that updating any single part will cause all the forms that import that part to be updated next time they are previewed or published. You can use the impact analysis tool to work out where your parts are used.
- Don't forget to NOT "update form to use block" each time you "Save as a custom block"
- Use the same block name every time you "Save as a custom block".
- Never edit your blocks using the block editor. The form that contains the block is the "source of truth". Always edit the blocks in their "part form", and then save the block from there.
- Remember that blocks are global to an organization. If you are using this technique across multiple forms, you may want to set up your naming convention so that your parts can be easily identified.

Related articles

- How to disable the standard Submit/Attachment button in TransactField

- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Addresses
- Address Block Samples

Testing forms - simple tips for ensuring quality products

Unknown macro: 'redirect'

Testing is one of the most important activities you will perform during your forms project. To that end, a common sense approach is needed as to what can be achieved within the time allocated for testing. For instance, if the form is a little more complex then it will be necessary to spend more time ensuring the form works as expected. It is imperative that you discuss the allocated time with the Project Manager and highlighting as early as possible if the time is not adequate. This can help ensure then product, when delivered, is of high quality.

The following list provides some guidance as to the common things to look for when testing forms. Specific requirements also need to be considered and it is important to ask the forms developer of any areas that will need special attention (i.e. complex scenarios).

- First and foremost is to check the responsiveness of the form > this is to be done in either Chrome or Firefox
    - Has a specific browser (and version) been requested to develop/test the form on? (If so, then the form will have to be checked there as well.)
- Has a url been supplied that will take you to the form?
- Does the form open without error(s)?
- Look and feel of the form – does it flow and look clean?
    - Does it flow from top to bottom and left to right?
    - Are there any obvious anomalies?
- Logo
    - Is it a sharp and clear example (not pixelated)?
    - Is the form branded to the 'Organization's' usual style?
- Spelling and/or Grammar errors
    - Note: as a matter of convention colons ':' are no longer added at the end of field labels
- Date field(s)
    - US date format or Aust / English format
- Tabbing order should be logical
- Hide/show functionality works as specified
- Validation
    - Errors in form works
    - Mandatory fields working
    - Error messages are as per specifications
- Tooltips
    - Clear and to the point. Avoid using “Please…”. Ensure accessibility compliance.
- Attachments
- Submission
    - Does the submitted XML look correct?
- Receipt
    - Ensure all the data entered has appeared on any receipt files generated post-submission
- Payment

While testing can consume a lot of time, using the above list, combined with a comprehensive testing plan which includes your form(s) specifics, will help ensure quality forms time after time.

Related articles

- Setting up 'Leave Page Confirmation' in Composer using JavaScript
- Understanding transaction management in Maguire forms
- Adding a security question in Maguire forms
- Blocking users with unsupported browsers
- Tuning the dependencies on the 'Triggering Conditions' tab in the Script Editor

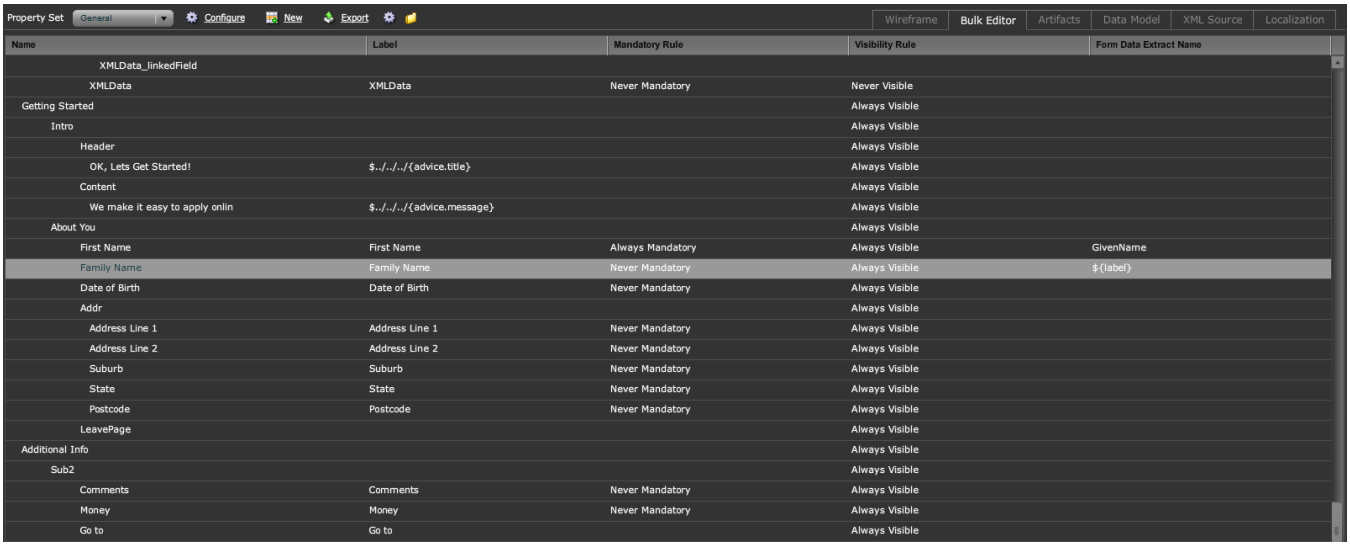
Using the Bulk Editor in Composer

Unknown macro: 'redirect'

The Bulk Editor is used to edit the same property across multiple fields without having to open each field's property editor. This is done by displaying properties in columnar format. Properties are defined using property sets which are stored at organizational level.

How to use the Bulk Editor

1. In the 'Bulk Editor' tab, select the required property set from the drop down list.
2. Each row indicates a widget in the form, with editable properties separated into columns. Click in the appropriate column, in line with the field and update the property e.g. rename label 'Surname' to 'Family Name'.



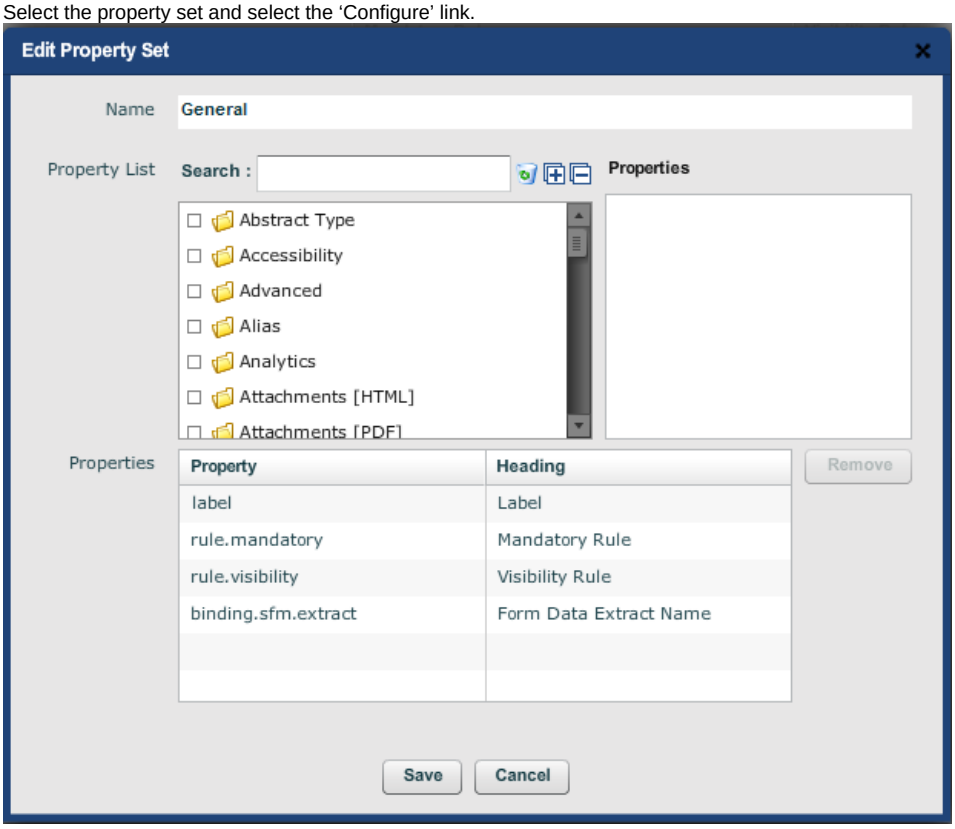
Name	Label	Mandatory Rule	Visibility Rule	Form Data Extract Name
XMLData_linkedField				
XMLData	XMLData	Never Mandatory	Never Visible	
Getting Started			Always Visible	
Intro			Always Visible	
Header			Always Visible	
OK, Lets Get Started!	$../../../{advice.title}		Always Visible	
Content			Always Visible	
We make it easy to apply onlin	$../../../{advice.message}		Always Visible	
About You			Always Visible	
First Name	First Name	Always Mandatory	Always Visible	GivenName
Family Name	Family Name	Never Mandatory	Always Visible	${label}
Date of Birth	Date of Birth	Never Mandatory	Always Visible	
Addr			Always Visible	
Address Line 1	Address Line 1	Never Mandatory	Always Visible	
Address Line 2	Address Line 2	Never Mandatory	Always Visible	
Suburb	Suburb	Never Mandatory	Always Visible	
State	State	Never Mandatory	Always Visible	
Postcode	Postcode	Never Mandatory	Always Visible	
LeavePage			Always Visible	
Additional Info			Always Visible	
Sub2			Always Visible	
Comments	Comments	Never Mandatory	Always Visible	
Money	Money	Never Mandatory	Always Visible	
Go to	Go to		Always Visible	

How to update a property set

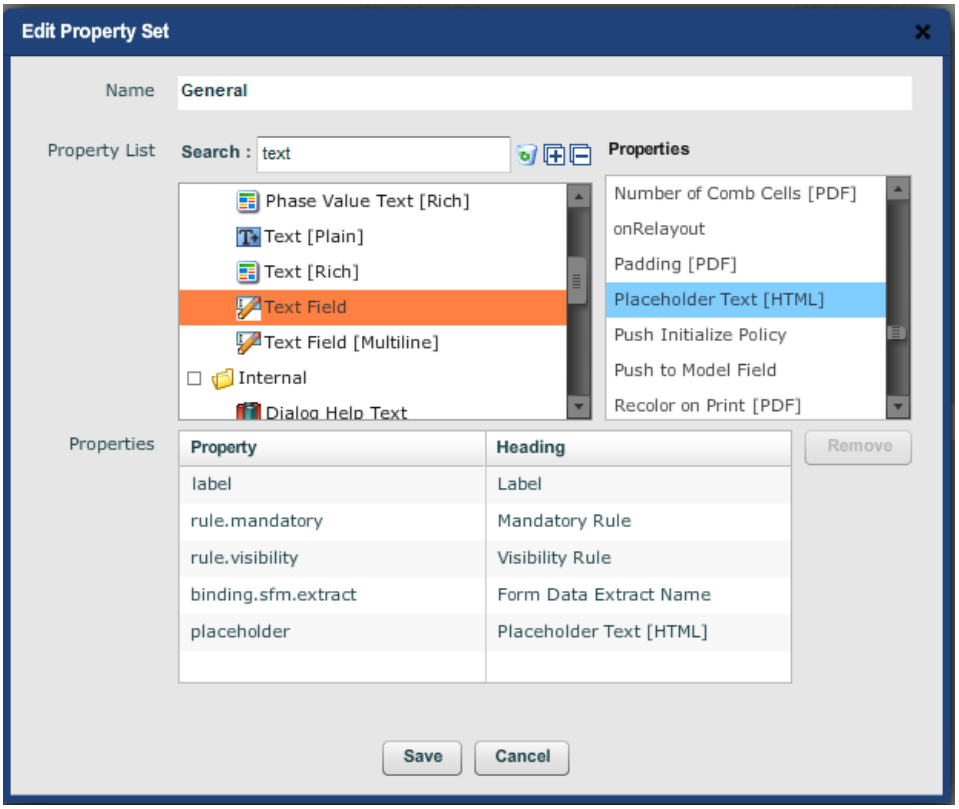
Property sets determine which properties appear in the bulk editor.

1. Using the property editor, verify the name of the property you want to update.

2. Select the property set and select the ‘Configure’ link.

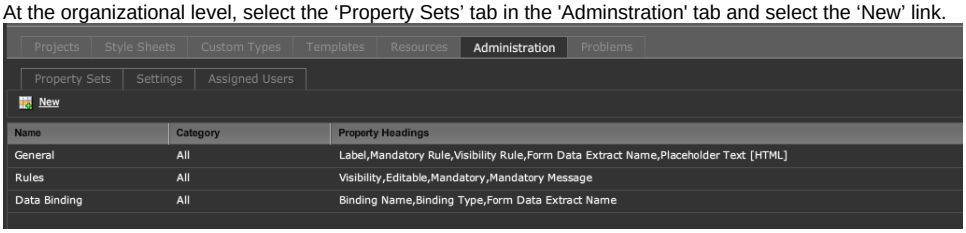


3. Enter the name of the widget the property applies to e.g. text field. A list of related properties are displayed. Double-click the required property e. g. ‘Placeholder [HTML]’ to add it to the existing list of properties being displayed.

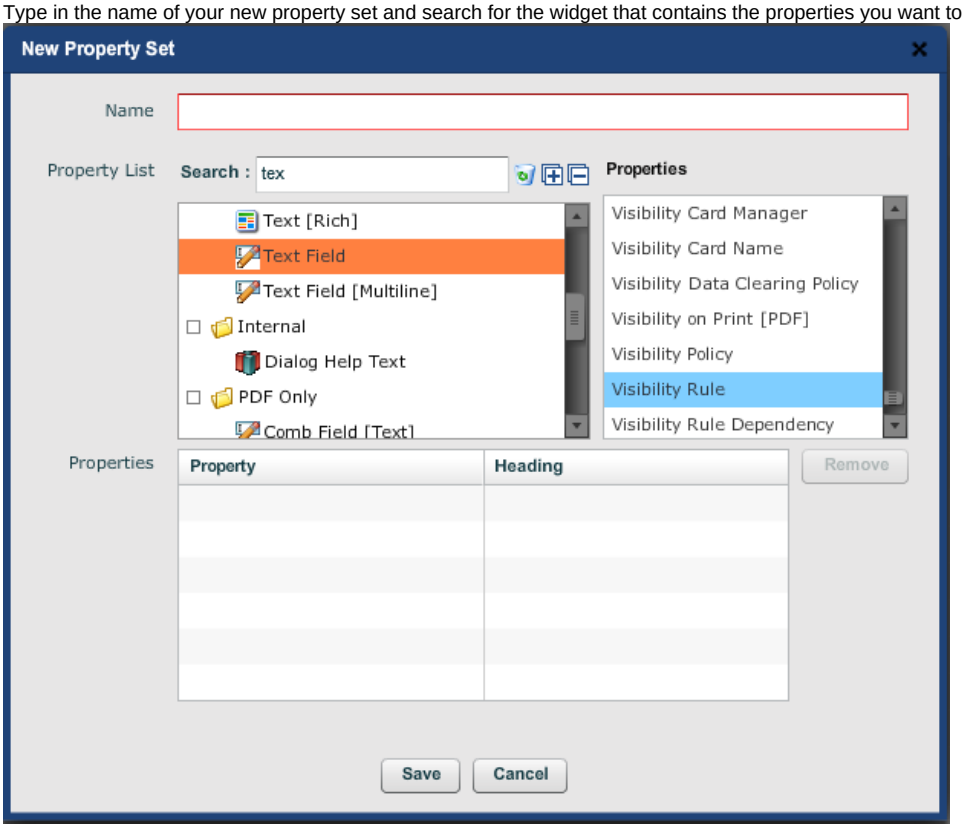


How to create a new property set

1. At the organizational level, select the ‘Property Sets’ tab in the 'Adminstration' tab and select the ‘New’ link.



2. Type in the name of your new property set and search for the widget that contains the properties you want to include.



3. From the properties list, double click to add the relevant properties to the list.
4. When you open the Bulk Editor tab in your form, the new property set will appear in the property set dropdown list.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Addresses
- Address Block Samples

XML Binding in Composer Forms

Unknown macro: 'redirect'

Binding types

There are three types of binding available in Composer forms; Bound, Not Bound, and Skipped.

Bound

Bound objects will appear in the submitted XML. Bound fields will have the value entered into the field stored in the XML, while bound blocks and sections will appear as elements in the XML, with their children containing values. Objects can be bound using absolute or relative binding.

Absolute binding allows you to control exactly where the object will be bound to in the submitted XML.

Relative binding will place the element inside the element that the parent object is bound to.

For example, a block with absolute binding of “$record.FormData.Name”, which contains a field using relative binding to “FirstName” will appear in the submitted XML as:

```
<FormData>
   <Name>
      <FirstName>Value entered by user</FirstName>
   </Name>
</FormData>
```

The value “$record” in absolute binding references indicates the root element of the schema.

Not Bound

Objects that are not bound will not appear in the submitted XML, and neither will any of their children. This means that any values entered into these objects will not be saved or submitted. This setting is generally used for objects that are not related to data capture, such as menu buttons, or calculated fields used to control form layout.

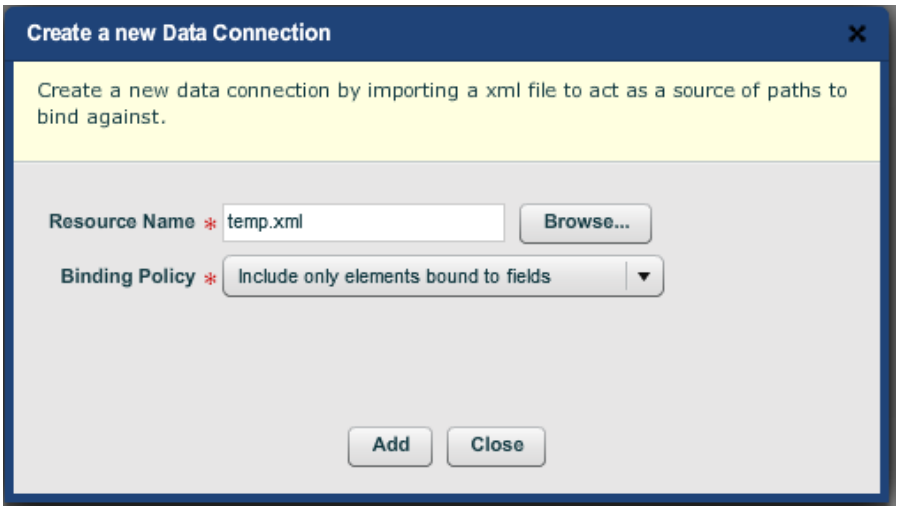
Skipped – Process Children

Objects with the ‘Skipped’ binding type will not appear in the submitted XML, but any of their children that are bound will be. This setting is generally used for block or section widgets.

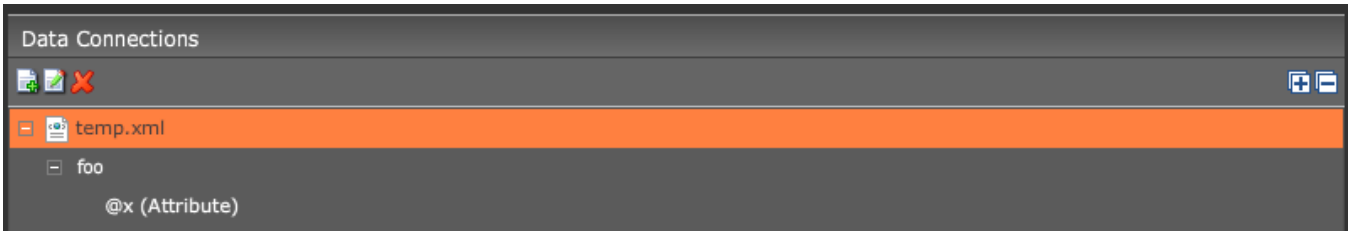
Binding to an Imported Schema

Composer forms generate their own schema using the values entered into the binding reference properties. The default schema can be viewed in the Data Connections window on the Data Model tab.

It is also possible to import an existing schema, and bind the form objects to it. To do this, go to the ‘Data Model’ tab, and click the ‘Add Data Connection’ button  .



Upload an XML or XSD file as a resource, select it, and click the ‘Add’ button. The imported schema will appear in the ‘Data Connections’ window. To bind objects to the new schema, drag them from the ‘Bindable Fields’ window, and drop them onto the appropriate element in the imported schema.



The same method can be used to bind form values to attributes.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Managing multiple service endpoints and credentials for external service calls
- Groovy Services
- Collaborative form completion with anonymous users (form sharing)

Accessibility - are Transact forms accessible?

Unknown macro: 'redirect'

Avoka Transact allows WCAG 2.0 AA compliant HTML forms to be built easily and quickly. Form Designers using Transact must apply best practice to ensure this level of accessibility is achieved. But, ultimately, the *only* way you can be 100% sure that the forms you build are accessible is to get an accessibility expert to test them.

Avoka aims to achieve 100% WCAG v2 AA compliance in Total Validator, and 100% compliance in SiteMorse.

While we do pass automated tests, we also understand that there is a difference between passing tests and being truly accessible to users with accessibility conserns. Any additional/specific Accessibility requirements are not automatically in scope, instead being dealt with on a case by case basis.

Q: Are we confident to state that forms built using Transact Composer will be WCAG v2 AA compliant?

A: No. We will never be able to certify that. But we can make it as easy as possible to achieve AA compliance.

Long answer:

1. Passing automated tests doesn't guarantee compliance (although it's a good start). Automated tests tell you when you've done something wrong, but they don't tell you when you've failed to do something right. We are in the process of doing further human expert tests on sample forms to improve our compliance.
2. Ask two different "experts" to certify a form, you can very often get two different answers – there is a level of subjectivity in interpreting WCAG. It also depends on the OS, browser, and assistive technologies (such as Screen Readers) being used to access and render the form.
3. Composer is just a tool. It helps you to build forms that work well, but it also allows you to do what you want. You could easily do things in your form, either intentionally or unintentionally, because of the way you build it, or the JavaScript you hand-craft, that breaks accessibility. We can NEVER certify that your forms will be accessible just because our tool starts from a good place. For example, you may put light grey text on a dark grey background and fail contrast requirements. Or name an image inappropriately. It's a bit like asking Microsoft to certify that all documents you produce will be spelled correctly just because Word has a spell checker.

Reference

https://www.totalvalidator.com/

Related articles

- Accessibility - are Transact forms accessible?

JavaScript in Composer

Unknown macro: 'redirect'

- Composer Scripting Quick Reference Guide
- Debugging Script in Composer Forms
- Using JavaScript to access fields in repeat
- Importing a JavaScript library into a Composer form
- JavaScript: Doing number arithmetic correctly
- Tuning the dependencies on the 'Triggering Conditions' tab in the Script Editor
- Blocking users with unsupported browsers
- Setting up 'Leave Page Confirmation' in Composer using JavaScript

Composer Scripting Quick Reference Guide

Unknown macro: 'redirect'

Referencing fields in a script

To reference a field in a script, use the hierarchy panel in the bottom left of the script editor window. The hierarchy matches the ‘Structure’ panel to the left of the wireframe on the main form development page.

To insert a reference into the code, find the field in the structure, and double click it. The field will appear in the list of dependencies as a parameter, and a reference will be inserted into the code. Once a field is in the dependencies list, it can be reused by double clicking.

Parameter types

Parameters in the dependencies list can be configured different data types, depending on the purpose of the script. For example, a common scripting requirement is to show a field or section if a radio button is set to ‘Yes’. The best way to achieve this would be to reference the ‘Yes’ radio button in the list of dependencies, using the data type ‘Is Selected’. This will return a Boolean value to the code, set to either true or false, depending on the status of the radio button.

The full list of parameter types and their purposes follows:

String

This returns the value of a text field, i.e., if there is a field called ‘Name’ and the user enters the value ‘Bob’, the variable will contain the value ‘Bob’.

Boolean

A boolean value is either ‘true’ or ‘false’. This parameter type is commonly used to return the value of a checkbox. The advantage of using a Boolean data type over a string is that it can be used in ‘if statements’, i.e.,

```
if({fieldReference}) {
    //do something here
}
```

Number

Much the same as a String, in that it returns the value entered into a field. The advantage of using a Number is that it can be used in calculations, i.e.,

```
var total = {fieldReference1} + {fieldReference2};
```

Is Selected

Only used with radio buttons and check boxes. It returns a Boolean value indicating if the referenced object is selected.

Node

Referencing a parameter type of node returns the entire field object as a variable. This allows the use of library functions, such as checking if the field is mandatory, or setting the value of the field using the script.

```
if(sfc.isMandatory({fieldReference})) {
    //do something here
}

sfc.setRawValue({fieldReference}, “Change field value to this”);
```

For more information about library functions, see the ‘Using Library Functions’ section at the end of this document.

Raw Value

This returns the value entered into a field, much the same as String. The difference is that it ignores any styles or patterns set on the field. For example, a currency field set to display the value “$1,000.00” will return a raw value of ‘1000’.

UID

This returns the ID of the field. This can be used in library functions that require the UID to be passed as a parameter, i.e.,

```
sfc.convertToNodeArray({uidParameter});
```

Arrays

When referencing objects that are configured to repeat, parameters can be configured as Array (node) and Array (value) types. Arrays contain a list of each instance of the repeating object. Value arrays contain a list of strings containing the value entered into each repeating field, while node arrays contain a list of the objects themselves. Individual elements of the array can be accessed using the index in square brackets after the parameter reference, i.e.,

```
var value1 = {arrayParameter}[0];
var value2 = {arrayParameter}[1];
//(Array indexes start at 0)
```

Script types

Different script types need to return different parameters to the form. Failing to return the correct parameter, in the correct format will cause errors in the form. For example, a calculate script needs to return the new value that is calculated for the field. This is done by adding a return statement to the end of the script, i.e.,

```
var total = 0;
total = {field1} + {field2};
return total;
```

The full list of script types and their requirements follows:

Visibility

Returns either true or false. When set to true, the object that the script is set on will show on the form, when false it will be hidden.

Editability

Returns either true or false. When true, the field will be editable by the user, when false it will appear greyed out, and won't be editable.

Calculation

Returns the new value of the field. The data type of the value should apply to the field that the script is configured on - for example, returning a string in a calculation rule set on a checkbox will cause an error.

Validation

Returns either true or false. True indicates that the value entered into the field is valid, false indicates that it is not, and an error will appear on the form. The user will not be able to be submit the form until the value is corrected.

Mandatory

Returns either true or false. True indicates that the field must be completed before the form can be submitted, false indicates that the field is optional, and can be completed or left empty.

Click action

Used on buttons, and is fired when the user of the form clicks the button. These scripts do not require a return statement.

Business Rule

Error and warning business rules return either true or false. When true, the form will display an error message or warning. When false nothing will be displayed. General purpose business rules do not require a return statement, and are just used to execute scripts.

TM Attachment Rule

Returns either true or false. When true, the attachment will be displayed in the attachment table if one is configured in the form and the attachment page will appear after the form filler submits the form. When false, the attachment will not appear in the table.

Using library functions

Composer forms have a built in JavaScript library, which allows developers to access some advanced functions. For example, it is possible to check if the form is displaying in receipt mode, using the code:

```
sfc.isReceipt();
```

or to set the value of a field from a script using:

```
sfc.setRawValue({nodeReference}, “Change the value to this”);
```

To access the full list of library functions, click the question mark at the top right of the script window. The functions listed in the ‘sfc’ class can be used in scripts within the form.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services

Debugging Script in Composer Forms

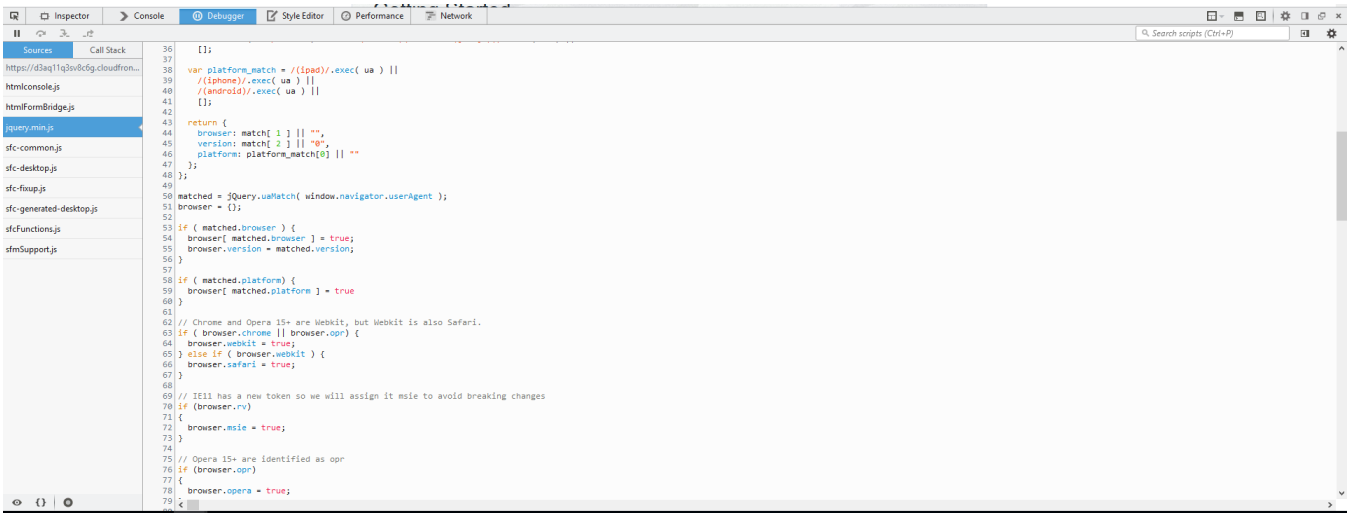
Unknown macro: 'redirect'

There are 2 types of errors that you may need to debug within a script i.e. syntax errors and data errors.

Debugging Logic errors

If an error message does not provide enough information to debug the form, it is possible to get more information by using the development tools built into most web browsers. To access these, preview the form in HTML Desktop mode (even if the form is being developed as a PDF form), then bring up the console (press F12 in IE or Chrome, in Firefox go to Tools -> Web Developer -> Web Console). JavaScript errors will be reported on the console, as well the JavaScript library, and the line number that caused the error.

Clicking on the reference to the script library will open the file, and highlight the line that caused the error.



While this will not do much to help identify what is wrong with the script, it will at least give clues to which piece of code is causing the error.

It is important to note that the error will not always be on the line that it is reported. In the example above, there is obviously nothing wrong with the highlighted line, but the line above is missing a close bracket, which is causing the error to trip on the next line.

Common errors to look for

Common errors to look for when debugging a form, apart from basic syntax errors are:

Ensure that script objects are returning the required parameters

Calculate scripts need to have the calculated value in a return statement at the end of the script, visibility and mandatory scripts need to return a Boolean value. If there is no return statement in the script, Composer will insert it around the last line of code, which can cause syntax errors.

Ensure that the correct variable types are used

When declaring dependencies in the script, check that the correct variable type is selected for the purpose. For example, setting up a dependency of type ‘String’ can cause errors if you try to perform mathematical operations on it. Likewise, calling sfc.setRawValue() on a String, rather than a Node will cause an error.

Development techniques to aid script debugging

There are a few things to keep in mind when developing a form that can lead to a less frustrating experience when debugging script errors:

- Put comments in scripts to help identify which field is causing an error when looking at the code.
- Develop and test scripts one by one – it will be easy to determine which script is broken if you have only modified one script since the form was last working.
- Break complex scripts into smaller segments using hidden data fields.

Debugging Data errors

The easiest way to debug scripts that aren't throwing an error, but aren't running as expected, is to log information to the console. This can be done using the sfc.debug() function. For example:

```
var x = "123";
sfc.debug("The value of x is: " + x);
```

When this script is triggered, the following will appear in the console:

To display the console in Chrome or IE, press F12. To display it in Firefox, go to Tools -> Web Developer -> Web Console.

Related articles

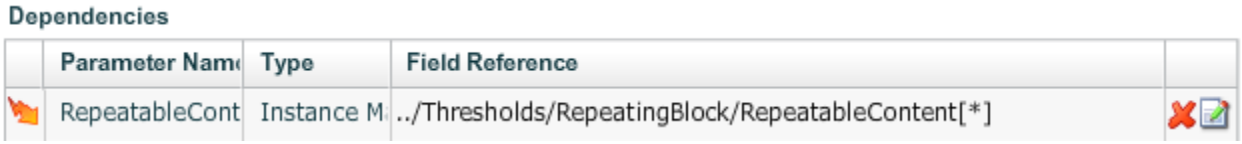
- How to disable the standard Submit/Attachment button in TransactField
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services

Using JavaScript to access fields in repeat

Unknown macro: 'redirect'

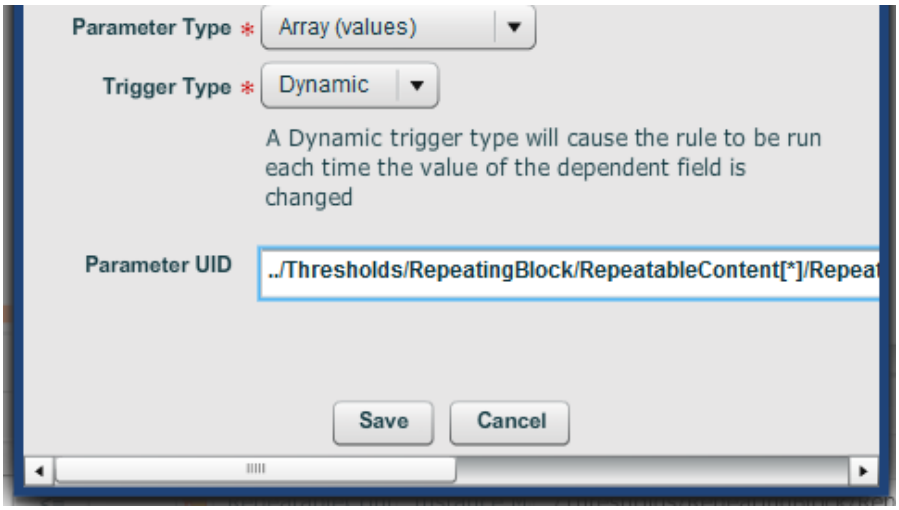
Accessing field nodes in individual instances

To access fields within a repeatable block, one must reference the instance manager for the repeat, in this example, this will be {RepeatableContent}. To reference the correct object, one may have to show the hidden internal fields with the gear icon while in the script editor.



Once you are sure you have the correct reference, you can access specific instances of the repeat using the getInstance() function or iterate through all the instances as shown below.

To access the fields within the repeat, you can convert it into a node via the sfc.normalizeUid function and the path of the field. The hidden fields are also required as part of the path, so if you're not sure, drag in the desired field to use as reference and check it's Parameter UID.



Once you have the node object, you may manipulate as you wish.

```
var targetIM = {RepeatableContent};


for (var i = 1; i < targetIM.getInstanceCount(); i++) {

   var targetNode = sfc.convertToNode(sfc.normalizeUid(targetIM.getInstance(i), "./RepeatInner/Item/_outerArea/_contentArea/RepeatingFields/Name"));


   sfc.setRawValue(targetNode,"Name " + i);

}
```

For example this will set the name field in the repeat to "Name 1" for the first instance, "Name 2" for the second and so on.

Related articles

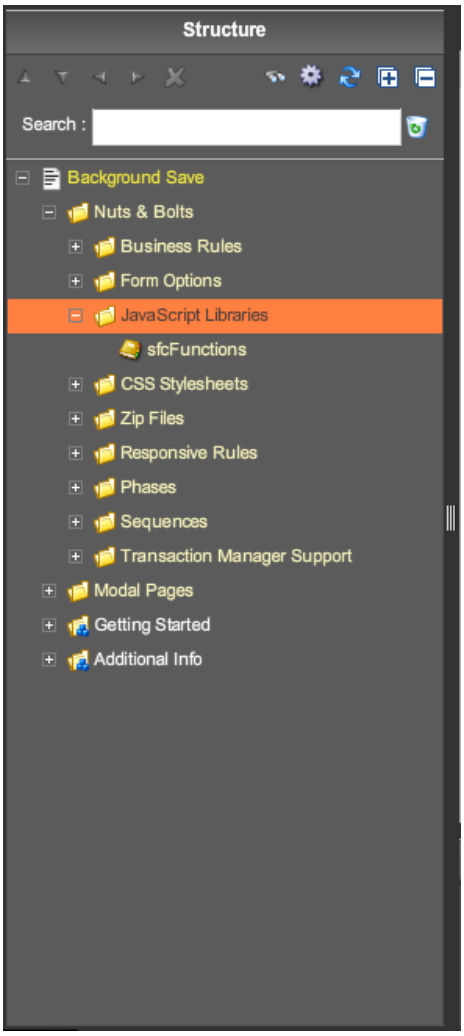
- How to disable the standard Submit/Attachment button in TransactField

- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services

Importing a JavaScript library into a Composer form

Unknown macro: 'redirect'

Javascript libraries can be imported into composer forms by dragging a ‘Javascript Library’ widget into the ‘JavaScript Libraries’ folder, inside the ‘Nuts & Bolts’ folder in the form.



Select the Javascript resource in the widget configuration window. The source file must be loaded into the organization as a resource, however it is possible to do browse the file system and upload a new resource from within the Javascript Library widget’s configuration window.

Sample form: https://composer.prod.avoka.com/composer/secure/composer.htm > **Composer Scripting > JavaScript Import Demo** imports a javascript library that contains the following code:

```
this.ExampleUtils = (function(exports) {
  exports.logEvent = function(name) {
    console.log("hello " + name);
  };
  return exports;
})(this.ExampleUtils || {});
```

This makes a class called ‘ExampleUtils’ available to the form, which contains the function ‘logEvent’, which can be used to print a name (supplied as a parameter) to the console.

The example calls this function using a button, with the code:

```
ExampleUtils.logEvent({Name});
```

where {name} points to the value of a text field in the form.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services

JavaScript: Doing number arithmetic correctly

Unknown macro: 'redirect'

Javascript is a bit relaxed about types.

For example, most field values are text by default, and "1" + "2" treats the values as strings and concatenate rather than add them, and will return "12", which might not be what you're expecting.

To be sure you're dealing with numeric values, you can use the built-in function parseInt()

http://www.w3schools.com/jsref/jsref_parseint.asp

However, be careful - if the string has a leading zero, then parseInt will assume a hexadecimal value, which again may not be what you're expecting.

To ensure you get what you want, use one of the following:

- parseInt (stringvalue, 10); // The '10' tells parseInt to interpret the value as a base-10 number
- sfc.convertToInteger(stringvalue); // This is a built-in Composer function which will do the right thing

The way way to get the value of the current field is as follows:

- sfc.convertToInteger(sfc.getRawValue(me));

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services

Tuning the dependencies on the 'Triggering Conditions' tab in the Script Editor

Unknown macro: 'redirect'

Compatibility

Since	4.0
Deprecated	

There are a number of possible strategies for tuning complex forms to improve performance:

- Tuning the dependencies
- Tuning the triggers that make scripts fire
- Tuning the events that make scripts fire
- Tuning the observers watching form objects and events
- Use of the Business Rules widgets on the form

The example below illustrates the 'Tuning the dependencies' option.

The 'Triggering Conditions' tab is found in the Script Editor which maybe accessed from the property editor -> 'Rules' tab. It was introduced in version 4.

Dynamic vs Static Dependencies

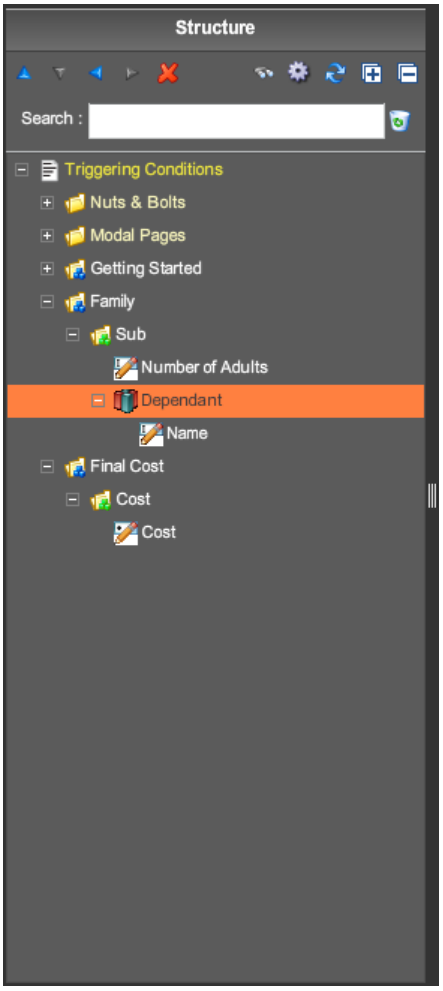
When you write a script in the script editor of the Edit Properties dialog, you reference a form object using a 'Parameter'. This creates a dependency, which appears in the 'My Rule Dependencies' panel of the 'Dependencies' tab.

The default setting for these is to have a 'Dynamic' trigger type, as indicated by the presence of the lightning arrow in the left-hand column of the Dependencies table of the Script Editor:

Changing the trigger type to Static means that the widget will no longer be observed. So, if we change a dependencies to a 'Static' trigger type, that rule will not fire. That we do want the rule to fire sometimes goes without saying, because we want the total to be the correct value. Which brings us to the 'Trigger Conditions' tab in the Script Editor.

Here, the widgets are only read when particular form events occur. We can specify additional events in the Triggering Conditions tab of the Script Editor:

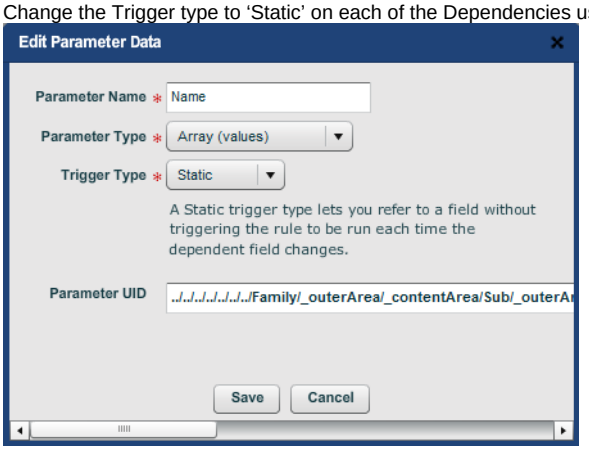
Example 1: Create a simple form containing repeatable section and a final cost calculation i.e. (Number of adults * 2000) + Cost of Dependants.



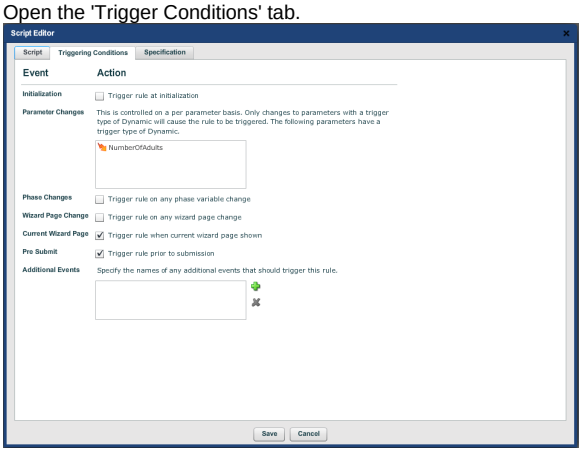
- This calculation is triggered 5 times by default (once on form initialization even though no values exist in the form; once when data is entered into ‘Number of Adults’ field even though the 'Dependants section' has not been completed and whenever the ‘Cost of Dependants’ field is updated e.g. 3 times if there are 3 dependants;

Solution

- Change the Trigger type to ‘Static’ on each of the Dependencies using the  button



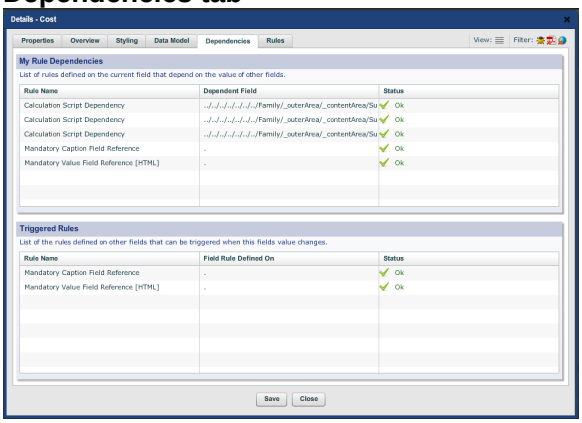
- Open the 'Trigger Conditions' tab.



- Remove the 'Initialization' tick ('Final Cost' will not be calculated because no values exist in the form when it is rendered).
- Select the 'Current Wizard page' checkbox (this field will be recalculated when the wizard page containing 'Final Cost' is accessed).
- Select the 'Pre Submit' checkbox (when the form filler selects the 'Submit' button, this field will be recalculated).
- **Result**: The 'Final Cost' calculation is triggered twice versus 5 times originally.

Select the Dependencies tab in the property editor to view the dependencies of the script.

Dependencies tab



My Rule Dependencies
Lists the Rules and associated form elements which, when changed will fire this element's Rules or Scripts depend.

Triggered Rules
Lists the rules which will fire upon this element's change.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services

Blocking users with unsupported browsers

Unknown macro: 'redirect'

Due to the interactive nature of SmartForms, Composer forms include features that are not supported by some browsers, such JavaScript libraries, and use of responsive design. Older versions of Internet Explorer are particularly limited - see Composer supported browsers. If a user does try to use a Composer form with an unsupported browser, it can be advantageous to display a message informing them of the fact, rather than simply letting them try to navigate a form that is not functioning correctly. This article demonstrates how to build features into a form that will redirect users with unsupported browsers to a page informing them which browsers are supported.

Demo form illustrating this functionality - this form is configured to block users on IE6 and IE7.

Detecting a user's browser:

It is possible to detect a user's browser and version by importing a JavaScript library into the form. The demo form uses the following code to detect the browser and version:

```
JSClientDetection.getClientInfo = function() {

    var nVer = navigator.appVersion;
    var nAgt = navigator.userAgent;
    var browser = navigator.appName;
    var version = '' + parseFloat(navigator.appVersion);
    var majorVersion = parseInt(navigator.appVersion, 10);
    var nameOffset, verOffset, ix;


    // MSIE
    if ((verOffset = nAgt.indexOf('MSIE')) != -1) {
        browser = 'Microsoft Internet Explorer';
        version = nAgt.substring(verOffset + 5);
    }

    var jscd = {
        browser: browser,
        browserVersion: version,
    };

    return jscd;

}
```

The browser and version can then be called in a calculate script:

```
var browser = JSClientDetection.getClientInfo().browser;
var version = JSClientDetection.getClientInfo().browserVersion;
```

Validating the user's browser

The demo form uses a form prefill service to import a list of unsupported browsers into the form when it is opened. While it is possible to hard-code the list into the form, it creates extra maintenance overhead. As the web environment changes, and different browsers and versions are released, the list will need to be updated. Hard coding the list into each form means that as a new unsupported browser comes into use, every form will have to be updated and re-released. By incorporating this list into a prefill service, the list will only have to be maintained in one place, and all forms will stay up to date.

For information on how to prefill data into a form, see Input XML Prefill Mapping.

The demo form loads the following XML containing the unsupported browsers:

```
<prefillData>
    <unsupportedList>
        <unsupportedItem>
            <browser>Microsoft Internet Explorer</browser>
            <version>6</version>
        </unsupportedItem>
        <unsupportedItem>
            <browser>Microsoft Internet Explorer</browser>
```

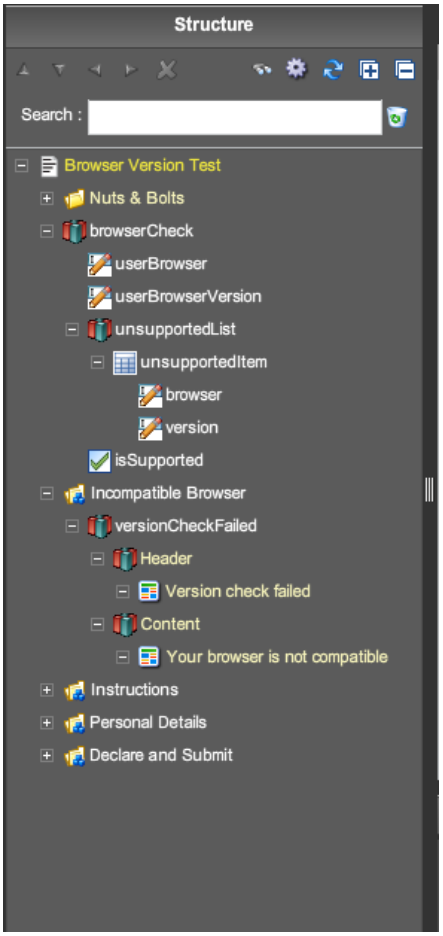
```
            <version>7</version>
        </unsupportedItem>
    </unsupportedList>
</prefillData>
```

Once the browser list is imported into the form, use a calculated data field to determine if the user's browser is on the list. The demo form contains a hidden checkbox with the following calculate script:

```
var returnVal = true;
var browserList = {browser};
var versionList = {version};

for (var i = 0; i < browserList.length; i ++) {
    if ({userBrowser} == browserList[i] && {userBrowserVersion}.split(".")[0] == versionList[i]) {
        returnVal = false;
    }
}

return returnVal;
```

Displaying content based on the browser version

By creating a hidden checkbox with a calculate event determining if the user's browser is supported, it becomes simple to show specific content using the checkbox in the visibility script. The demo form contains the following Section level 1:



The rest of the level 1 sections of the form have the opposite visibility rule applied. Therefore if a user with an unsupported browser attempts to view this form, all they will see is this message, with no option to submit the form. However if a user with a supported browser views the form, they will just see the form pages as normal.

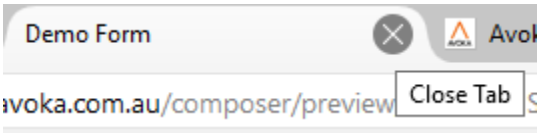
Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services

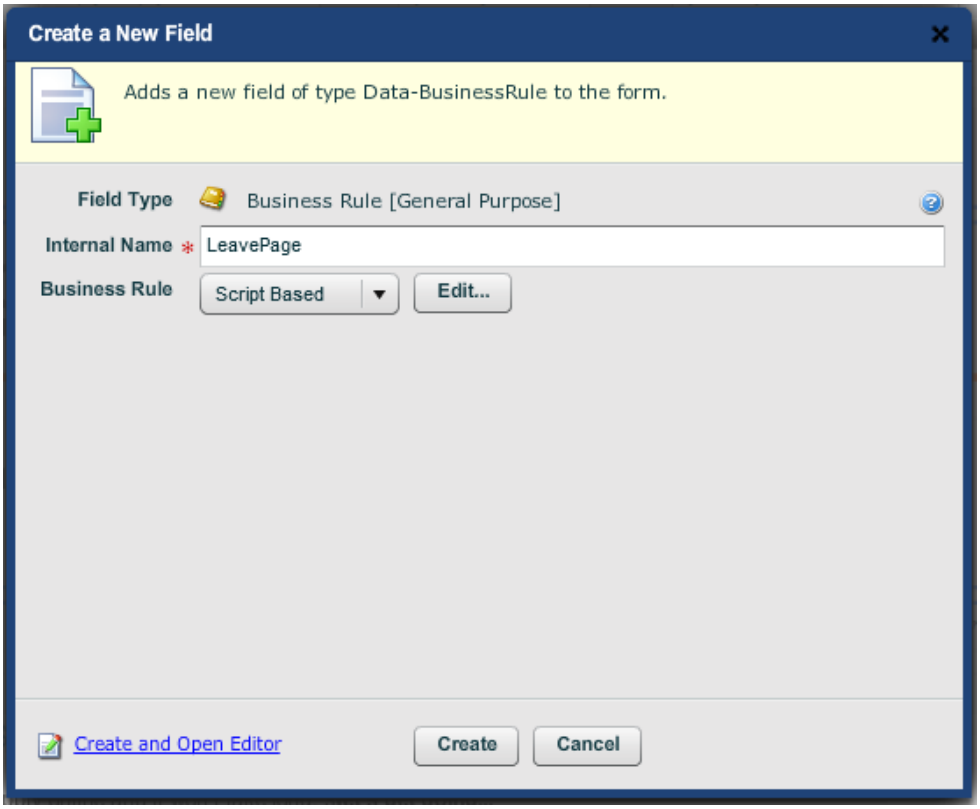
Setting up 'Leave Page Confirmation' in Composer using JavaScript

Unknown macro: 'redirect'

In some forms, it maybe necessary to confirm if the form filler wants to exit an HTML form when they select to close the browser.



To prompt the form filler, add a 'Business rule - General Purpose' widget anywhere in the form and open the property editor.



Include the following JavaScript code in the 'Business Rule' property:

```
window.onbeforeunload = function() {
    return "(confirmation message)";
}
```

Note: Different browsers will handle the confirmation message differently:

- Chrome will display the message, along with its own message saying ‘Are you sure you want to leave this page?’
- Firefox will ignore the message entirely, and simply display its own message saying ‘This page is asking you to confirm that you want to leave - data you have entered may not be saved.’
- Internet Explorer will simply display the message, with the options ‘Leave this page’ and ‘Stay on this page’.

Sample form: Composer production > Composer Scripting > Leave Page Confirmation

https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=leave-page-confirmat

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services

The Maguire Template

Unknown macro: 'redirect'

- Understanding transaction management in Maguire forms
- Configuring form level help in Maguire forms
- Adding a security question in Maguire forms

Understanding transaction management in Maguire forms

Unknown macro: 'redirect'

Compatibility

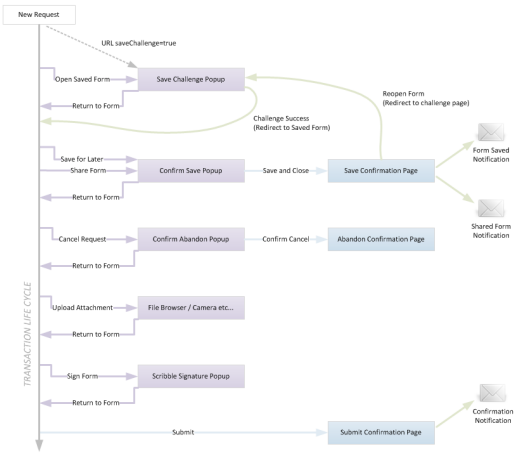
Since	4.1
Deprecated	

While a data capture form is commonly the primary element of an online business transaction, there are often a range of additional functions and capabilities required including signatures, attachment handling, payment processing, secure online save and resume, collaborations between multiple users, confirmation pages and notifications. All of these elements fall into the area that we refer to as '**Transaction Management**'.

Prior to Avoka Transact Version 4.1 all these additional features were managed outside of the form and facilitated by interactions with the 'Self Service Portal'. The Maguire 4.1 Composer pack utilizes new APIs in the Transact platform to assume responsibility for many of these functions, allowing form designers to configure entire transaction experiences from with the familiar Composer design environment, and removing the requirement to include the 'Self Service Portal' in these interactions. New capabilities for dialog boxes and modal pages within the form design make this possible.

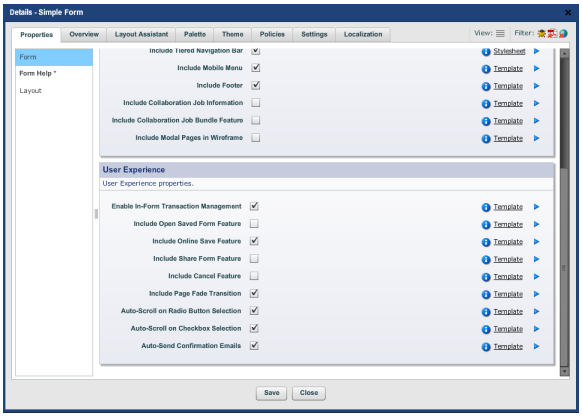
Maguire Supported Interactions

The following chart shows the interactions that are now supported in the Maguire 4.1 Composer pack:



Many of these interactions can be enabled or disabled in the form properties dialog in Composer (i.e. the root object of the Structure pane). Included in the configuration options are the following properties:

- **Enable In-Form Transaction Management**: If selected then the new in-form dialogs and confirmation pages will be utilized, otherwise the legacy portal flows will be used instead. If deselected, form designers can still make use of the in-form attachments and signatures but save and resume, abandonment, and submission confirmation will be facilitated by the portal.
- **Include Open Saved Form Feature**: If selected then a feature will be included on the first page of blank forms allowing users to access the Save Challenge popup dialog to open a previously saved form. If not selected, this feature will not be presented and any resume functionality will revert to the Self Service Portal.
- **Include Online Save Feature**: If selected, then a 'Save for Later' button will be presented in the form to facilitate saving of partially completed forms.
- **Include Share Form Feature**: If selected, then the 'Share Form' button will be presented in the form and the sharing section on the Save Confirmation Page will be exposed.
- **Include Cancel Feature**: If selected, then the 'Cancel / Exit' button will be presented in the form.
- **Auto-Send Confirmation Email**: If selected then both the Save and Submit confirmation emails will be automatically send when the user saves and submits respectively. Note: this requires that the Contact Email property is populated with the user's email address.

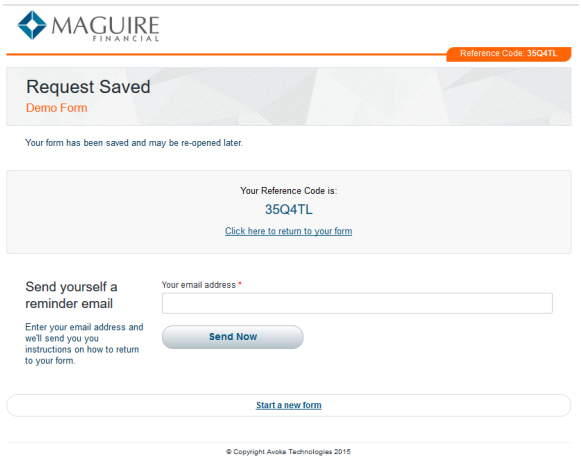


> ***NOTE**: With the exception of signature capture, none of the in-form transaction management interactions described in this article are supported in the TransactField mobile app. Support for in-form attachments will be introduced in a later release, however TransactField supports known authenticated users only and many of the in-form dialogs and confirmation pages described here are considered to be applicable for anonymous users only and are therefore not considered to be relevant in a field worker context.*

Save & Resume

The ability to save a form partially completed online and resume it at a later time is entirely facilitated within the form design.

When a user clicks the 'Save for Later' function button, the Confirm Save popup dialog is presented. If a security question is configured it will be shown in this dialog and must be answered prior to confirming the save action. When successfully saved, the Save Confirmation Page is presented highlighting the unique transaction reference code.



If automatic confirmation emails is enabled (Composer form properties), the user's Contact Email Address is known (linked to TM 'Contact Email' property) and no confirmation email has already been sent to the user (on a prior save action), then the user will automatically receive an email with instructions on how to return to the form. If the confirmation email is not automatically sent, the user will have the option to enter their email address and send one manually.

To return to the form the user can use the same URL that they originally used to access the form to open a blank form and select the 'Resume a saved form' link and manually enter their reference number and security answer or use the link they were sent in the confirmation email they were sent to automatically load the Save Challenge popup dialog with their reference number already populated. Upon successful completion of the challenge dialog, the users form will open on page one with all data previously entered.

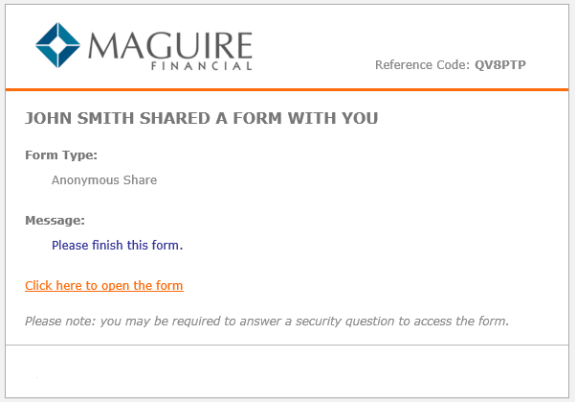
Users can save and resume their form as often as they wish. The automatic confirmation email will only be sent once, but can be triggered manually each time they save.

Form Sharing

Form sharing allows the user to send instructions on accessing the form to another individual so that they can open and assume control to assist in the completion of the form. The form sharing interaction piggy-backs on the Save & Resume flow because a form can only be shared once the form has been saved and the security question has been answered.

If form sharing is enabled (Composer form properties) the 'Share Form' function button will be presented in the form and available at any time. Clicking this button will open the Confirm Save popup dialog for the user to complete the security question and confirm. The Save Confirmation Page is then presented in 'Sharing' mode where the sharing fields are shown on the page. Note: if form sharing is enabled in the form, the Share Form section will always be presented on the Save Confirmation Page, but it will be presented in 'collapsed' mode unless the 'Share Form' button was selected.

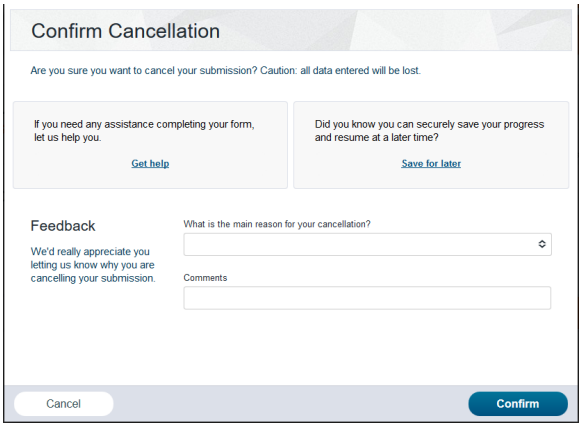
To share the form the user must enter the email address of the individual they wish to share with, their own full name and optionally a personal message to the recipient. The recipient will receive an email similar to the one below:



Form Abandonment

Where form abandonment is a key concern, the Maguire template supports an abandonment flow that can be enabled in Composer form properties and is exposed in the form as a 'Cancel / Exit' function button. By giving the user an explicit option to cancel their submission, this allows us to take control of the flow, offer some alternative options and prompt the user for more information to gain an understanding as to why people are abandoning.

When the 'Cancel / Exit' button is clicked, the Confirm Cancellation popup dialog is presented. This dialog offers the user options to get help on their submission or save it for later instead of abandoning. Underneath these options the user is prompted for a reason for their cancellation and any comments they have. These fields are optional but if completed the reason is surfaced in the Abandoned Transactions page within the Manager Console.

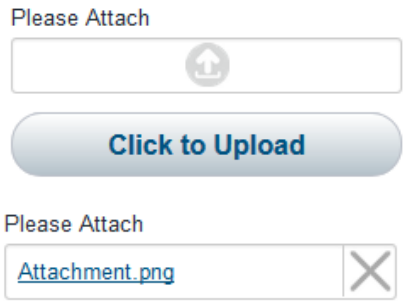


Upon confirmation, the user is presented with the confirmation page advising that their submission is now cancelled and will not be processed.

In-Form Attachments

To capture an attachment, add the Attachment Field to the form in the location that you wish to capture the attachments. Attachment fields support visibility, editability and mandatory rules like standard fields.

In its empty state, the attachment field shows the 'Click to Upload' button which launches the standard browser file dialog. When an attachment that complies is provided the upload button disappears and the file name is populated into the field as a clickable link that will trigger a download if clicked. This link will be active, even if the field itself is disabled. The form can be cleared again by clicking the delete ('X') button.

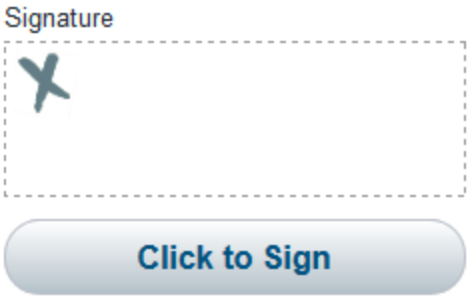


Form designers can configure the attachment field to support the manual delivery option where users indicate they will provide the attachment by conventional means.

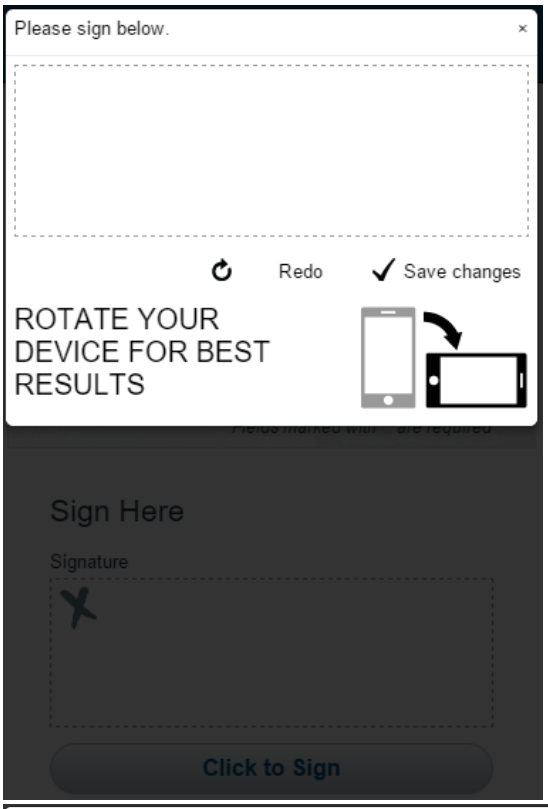
> ***NOTE**: As of version 4.1, the attachment field is not yet supported in TransactField although it is intended that this capability will be included in a later release.*

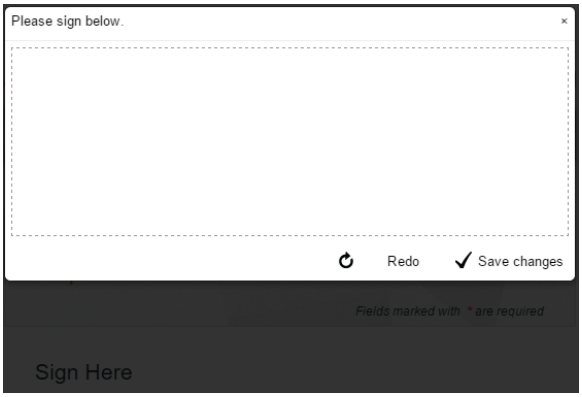
Sign on Glass

The Signature Field is not new to version 4.1, however it has had a significant upgrade. To capture a signature in the form add a Signature Field to the form in the location you wish to capture it. In its un-signed state an X mark indicates where the signature is required. The 'Click to Sign' button launches the Scribble Signature popup dialog (clicking on the field itself will also activate this dialog). When successfully signed, the X mark disappears and the captured signature is shown inside the field. The Scribble Signature popup dialog can be re-launched at any time to replace the signature if required.



The Scribble Signature popup supports all screen sizes with messaging to help users get the optimal result on mobiles and allows the user to clear and restart their signature if required. Users can use their finger or stylus on a touch device or their mouse on a laptop or desktop computer.



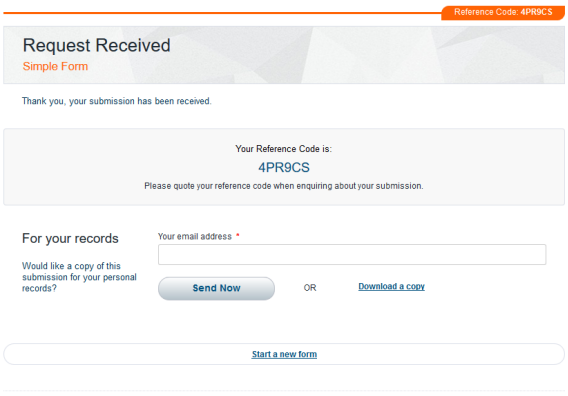


> ***NOTE**: the capture of a signature does not automatically set all fields to read-only so that no further changes can be made. If this is the desired behavior then it must be explicitly implemented separately.*

Submission Confirmation

When the form is valid and complete, the user may submit to the server. Upon successful submission the Submit Confirmation Page is presented highlighting their unique transaction reference code.

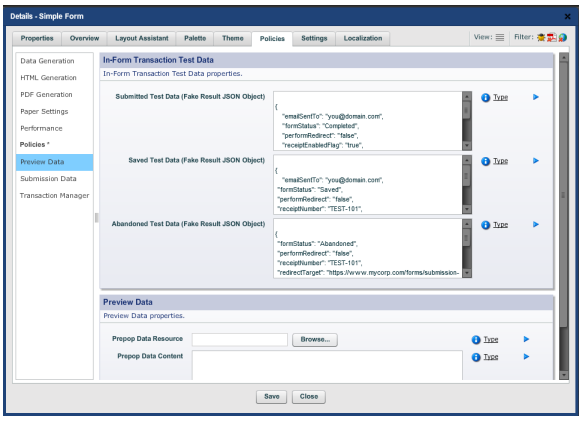
If automatic confirmation emails are enabled (Composer form properties), the user's Contact Email Address is known (linked to TM 'Contact Email' property) then the user will automatically receive an email confirming their submission and optionally providing a PDF receipt or document of record. If the confirmation email is not automatically sent, the user will have the option to enter their email address and send one manually or download the receipt directly from the confirmation page.



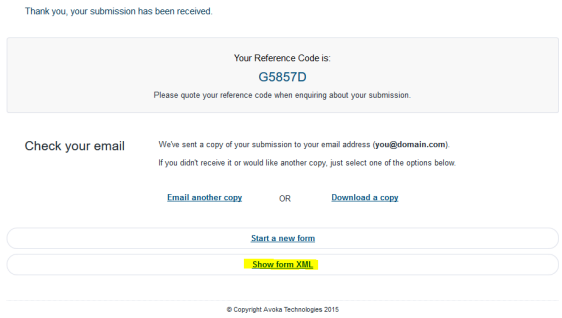
> *NOTE: the Submit Confirmation page also supports collaboration jobs with form bundles but this is beyond the scope of this article.*

Testing the Interactions in Preview Mode

In the Composer development environment these 'Save', 'Submit' and 'Abandon' interactions can be tested in Preview Mode. This is facilitated by sample response data that can be customized in the Preview Data section of the Form Policies tab. Default samples are provided in the Maguire package:



Additionally, each of the confirmation pages presents a 'Show Form XML' link at the bottom of the page while in Preview mode, that allows the form designer to verify the actual submitted XML data:



Remaining Portal Responsibilities

There are a couple of transaction management interactions that are not available in the form design and must still be handled by the Self Service Portal. These are:

- Wet Signature Flow: where an ink signature is required.
- Payment Processing

Of course the Self Service Portal also provides all the authenticated user functions such as task and history lists and profile pre-fi

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Interacting with TransactField using Custom URLs
- How to rescue attachments from your iOS device
- How to pre-fill static map images into TransactField Tasks
- Determining the IP address that a form is requested from

Configuring form level help in Maguire forms

Unknown macro: 'redirect'

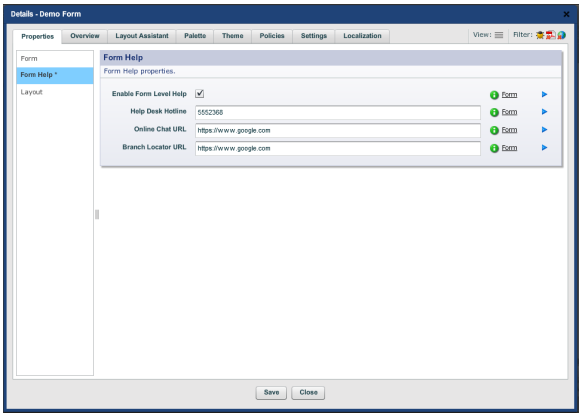
Compatibility

Since	4.1
Deprecated	

In many scenarios it makes sense to provide form users with one or more options to seek some type of human assistance to help them complete the form. The Maguire form template has in-built support for providing options of phone support, online chat, and access to branch locator.

Activating the help feature

To activate form level help, navigate to the Form Help sub menu of the form properties dialog and select the 'Enabled Form Level Help' checkbox.



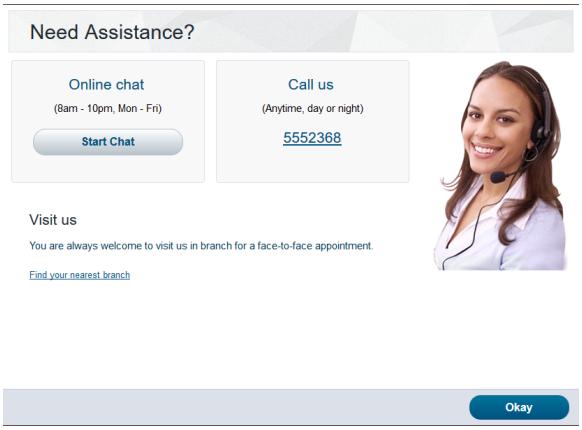
On this page you will also see three input fields used to capture the targets for the available help options. The visibility of the relevant sections in the Form Help dialog is driven by these fields such that if no value is entered then the corresponding section will not be shown in the dialog.

1. Held Desk Hotline: If you have a call center that is enabled to provide assistance to form users then you can enter the phone number in this field and the relevant section in the help dialog will be made visible. The phone number will be presented in the Form Help dialog as a clickable link which will send the user to the phone's dialer service if on a compatible device.
2. Online Chat URL: A web link to an online chat facility should be placed into this field to enable this option in the Form Help dialog.
3. Branch Locator URL: If a branch locator service is available then the web link to this service should be placed in this field.

Any URL will be loaded in a new browser tab/window so that the user's form session is not lost.

Testing the Help Options

To test the help options simply render the form directly from Composer and click the 'Need Help' button in the top right of the form header. The Form Help dialog should be presented with the help options that you have configured.



Customizing the Help Dialog Messaging

All the messaging in the Form Help dialog can be customized in Composer style sheets. An appropriate location for client specific customizations to be placed is in their Brand style sheet.

Some style sheet entries you may wish to consider customizations for are shown below:

```
    <style name="style.section.level1.form.help">
        <setproperty name="section.heading" value="Need Assistance?"/>
    </style>
    <style name="style.need.help.chat.advice.block">
        <setproperty name="advice.title" value="Online chat"/>
        <setproperty name="advice.message" value="(8am - 10pm, Mon - Fri)"/>
    </style>
    <style name="style.need.help.start.chat.button">
        <setproperty name="label" value="Start Chat"/>
    </style>
    <style name="style.need.phone.advice.block">
        <setproperty name="advice.title" value="Call us"/>
        <setproperty name="advice.message" value="(Anytime, day or night)"/>
    </style>
    <style name="style.need.help.level2.visitus">
        <setproperty name="section.heading" value="Visit us"/>
        <setproperty name="section.help.text" value="You are always welcome
to visit us in branch for a face-to-face appointment."/>
    </style>
    <style name="style.need.help.assistant.image">
        <setproperty name="width" value="201px"/>
        <setproperty name="height" value="300px"/>
        <setproperty name="image.path" value="Assistant.png"/>
    </style>
```

Please note:No license to use the 'Assistant' image is provided and an alternative image should be inserted if this facility is to be enabled. This image can be replaced by configuring the 'style.need.help.assistant.image' properties in the Composer style sheet as shown above. Effort should be taken to use an image of similar size to avoid layout issues with this dialog.

Replacing the Assistant Image

The default assistant image in the help dialog is not available for use in production scenarios without the client purchasing the image from the provider. To replace the image please follow these simple steps.

? Unknown Attachment

1. Show internal fields in the 'Structure pane' toolbar.
2. Expand _formHeader > NeedHelp > Instructions > Smile > _wrapper > Assistant
3. Replace the 'Image Resource Name' property by uploading another appropriate image and save.

? Unknown Attachment

Customizing the Help Dialog Content

If you wish to make content or structural changes to the Help Dialog then we recommend that you create an Alias block that either extends or replaces the built-in block 'Block-Dialog-FormHelp'.

This article does not cover the process of creating Aliases.

Migrating Form Level Help from 4.0 or Earlier

Prior to 4.0, form level help was provided as a hidden section in the form header that dropped down when the Form Help button was selected. In version 4.1 this feature has been extended, improved and moved into a pop-up dialog. Support for the in-form help content has been removed.

Any customizations to the Form Help content in a 4.0 or earlier form will need to be reapplied manually in the 4.1 version of the form. No migration capability is provided for this.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

Adding a security question in Maguire forms

Unknown macro: 'redirect'

Compatibility

Since	4.1
Deprecated	

When a public user is completing a form online and needs to save where they are up to so that they can return to their form later, it is good practice to inject a security question (in addition to the reference code) that will prevent unwanted access to the user's data and form in the interim. Security questions are typically personally relevant to the user and may be linked to a field within the form such as date of birth or phone number.

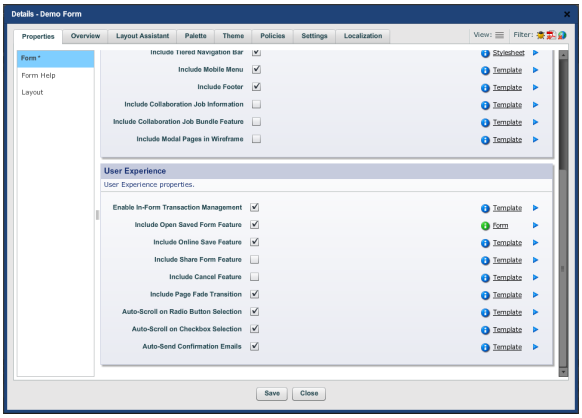
Prior to version 4.1, security questions could only be managed outside of the form as a function of the 'Self-Service Portal'. In 4.1 and later, security questions can be managed within the form, being implemented at design time instead of publish time. This development offers the following advantages:

- The process of asking the user to confirm the answer to their security question when they save, and challenging the user for the security question when they resume is completely integrated into the form filling experience.
- The UX and visual styling can be entirely consistent with the rest of the form filling experience without the need to duplicate styling effort into the Self-Service Portal.
- The entire palette of appropriate field types can be used to capture the security question answers where the portal only supports a text field. For example, a date field with pop-up calendar can be used to capture a date value.

Adding Your Security Question in Composer

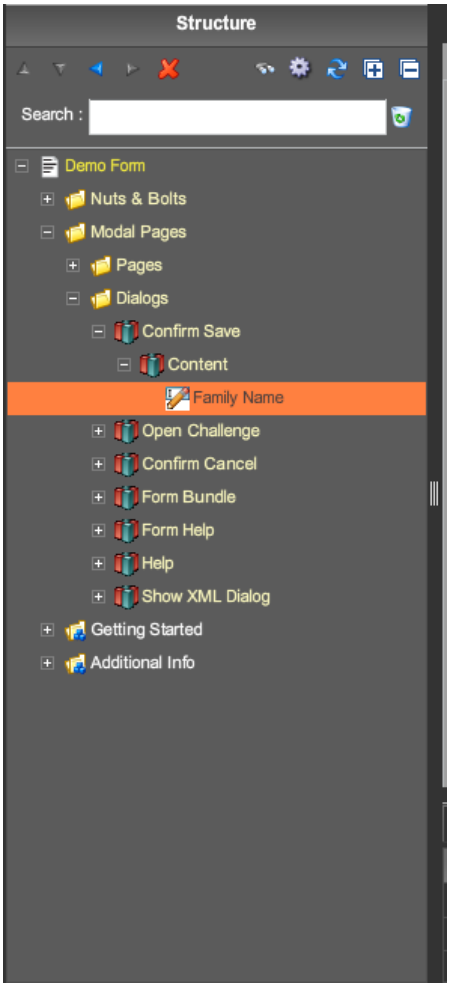
The process for adding a security question to a 4.1 form can be described as follows:

1. Ensure that your form is configured to support online save in both Manager and Composer. In Manager this can be found in the 'Flow' tab of the form configuration - the Online Save property should be set to 'Enabled' (default). In Composer, open your form properties and ensure the 'Enable In-Form Transaction Management', 'Include Online Save Feature' and 'Include Open Saved Form Feature' properties are selected.

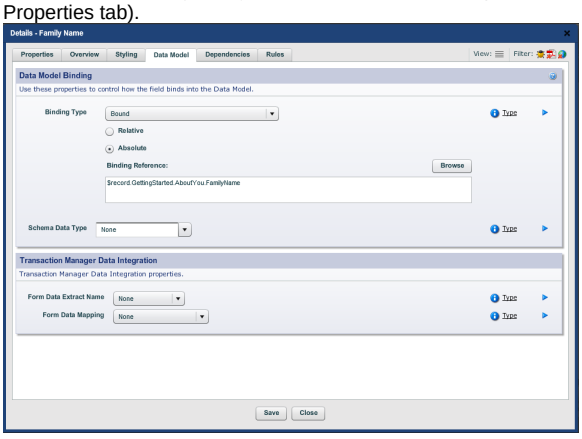


2. Determine the security question and type. For illustration purposes we will use the applicants 'Family name' text field. Find the field type you require in the palette and drop it into the hierarchy in Modal Pages > Dialogs > Confirm Save > Content. Give the field an appropriate

caption and ensure that it is configured to be ***mandatory***.



3. If you wish to link this field with an existing field on the form so that it can be pre-filled with an already provided answer when the user attempts to save the form, set the absolute data binding reference (under the Data Model tab) to point to the same location as the field in the form ***AND*** set the Data Clearing Policy to 'Preserve if not visible' (under the 'Data' sub menu of the Properties tab).

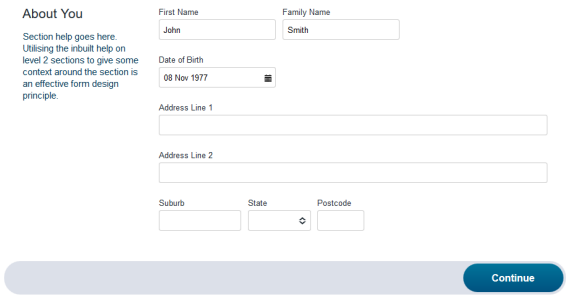


4. Copy and paste the field into the Open Challenge dialog Content block so that the same question is asked when the user attempts to open the saved form.

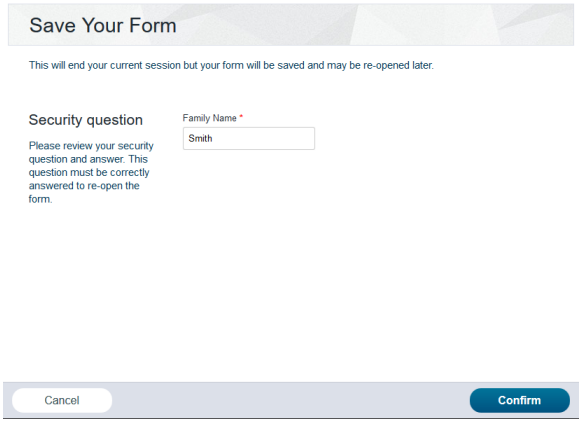
Testing the Security Question

To test this has been applied correctly, deploy your form to the Manager server and run the following verification steps:

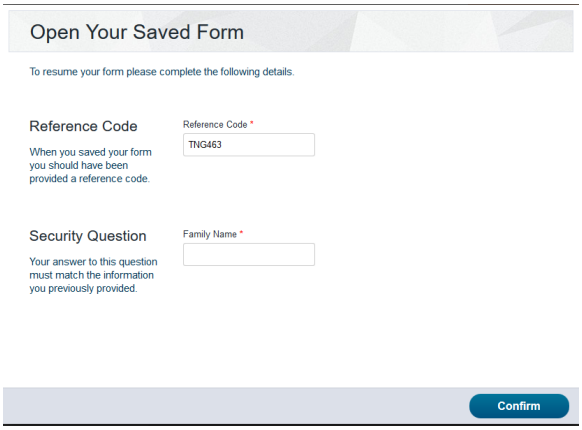
1. Open the form and complete some details. If you have linked your security question to a field in the form, ensure you complete that field in the form so that you can verify that the value is being pre-filled.



2. Click the 'Save For Later' button and complete the Security Question before clicking the 'Confirm' button. You should verify that the pre-fill is working as expected at this time.



3. Attempt to return to your form, enter your reference code (if not pre-filled for you) and the correct answer to your security question before clicking the 'Confirm' button. Ensure your form loads with the same data that you had entered previously.



Trouble Shooting

- The solution currently only supports a single security question. Ensure that you have only one question added to the Content areas of the dialog containers.
- If the form field linked to the security question is being cleared at submit time, ensure you have set the Data Clearing Policy of the security question in both dialog containers to 'Preserve if not visible' (under the 'Data' sub menu of the Properties tab).

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)

- Addresses

Datepicker Customisation in Composer

Unknown macro: 'redirect'

The JQuery Datepicker component included in Composer's Date field can be customised to exclude particular days of the week or specific dates if required. This how-to outlines how this can be applied to a Composer form.

Compatibility

Since	
Deprecated	

Step-by-step guide

Part 1: Excluding Particular Days

Here's the steps involved:

1. Determine the date types that need to be excluded, e.g. weekends, particular dates. You may need to give some though to how these exclusions will need to be maintained over time.
2. Add a Business Rule to override the JQuery Datepicker initalisation. This rule is used to encapsulate the exclusion logic to be applied. An example is shown below:

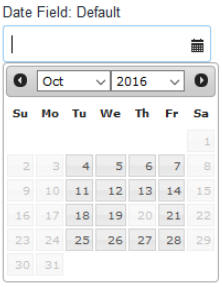
Datepicker Init Override

```
// a String array of invalid dates to block, e.g. public holidays or
the 20th of the month
// pro tips:
//  - use a format that can be easily parsed
//  - you could fetch this value from a data field or invoke another
business rule to generate it
var badDateList = ["2016-7-20", "2016-8-20", "2016-9-20", "2016-10-
20", "2016-11-20", "2016-12-20"];

// a function used by the datepicker to determine if a given day is
going to be a bad (invalid) day
// function is expected to return an array of true/falue values used
to indicate if the given date is selectable
// pro tip:
//  - you could invoke another business rule to encapsulate this
logic
var isBadHairDay = function(date) {
    // get the parameters of this date:
    // day of the week
    var day = date.getDay();
    // calendar parts
    var m = date.getMonth();
    var d = date.getDate();
    var y = date.getFullYear();
    // convert to our string format:
    var dateStr = y + '-' + (m + 1) + '-' + d;

    // now we apply our date exclusion rules in order of preferrence:

    // no weekends:
    var noWeekendList = sfc$.datepicker.noWeekends(date);
    if(noWeekendList[0] == false) {
        return noWeekendList;
    }

    // no Mondays
    if(day == 1) {
        return [false];
    }

    // not in our bad date list:
    for (var i = 0; i < badDateList.length; i++) {
        if ($.inArray(dateStr, badDateList) != -1 ) {
            return [false];
        }
    }
```

```
    // otherwise, all good
    return [true];
}

// initialise the Datapicker defaults globaly for the form and
configure it to call our exclusion function
sfc$.datepicker.setDefaults( {
    beforeShowDay: isBadHairDay
});
```

3. Configure the rule on run on form initalisation.

Here's an example of the above rule:



Pro Tips

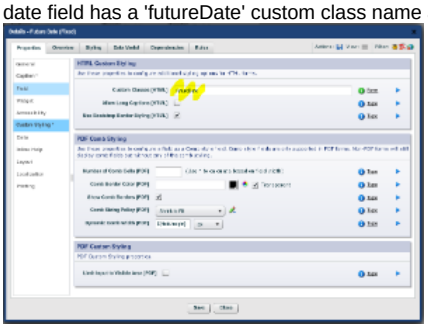
You might want to also consider:

- Using a data field or invoking another business rule to generate the list of exclusion dates.
- Using another business rule to encapsulate the date exclusion logic. That way it can also be available to other areas of your form in case you need to validate dates elsewhere.

Part 2: Showing Future Dates Only

Here's the steps involved:

1. Determine if you need to differentiate between 'normal' dates and 'future' dates in your form. If so, using a **Custom Styling Class** to 'tag' the future date field is useful. In this example, the date field has a 'futureDate' custom class name applied to it in Composer:



2. Add a Business Rule to override the JQuery Datepicker initalisation. A JQuery **find** operation can be used to target specific date fields by class name, an example is shown below:

Datepicker Init Override

```
// initialise datepickers on future date fields
sfc$(".futureDate").find("input").datepicker( {
        changeMonth: true,
        changeYear: true,
        showMonthAfterYear: true,
        yearRange: "-00:+50",
```

```
        minDate: 0,
        maxDate: "+50y"
});
```

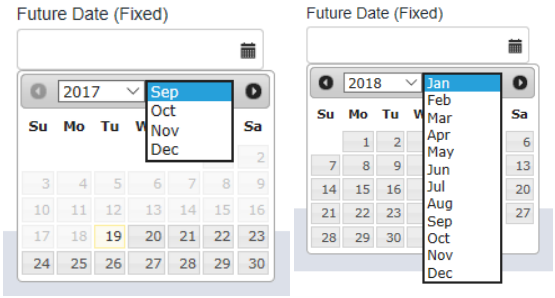
Other Tweaks

In this example:

- the month/year lists are swapped position (so that it's more naturally left-to-right)
- the date range is set from today to 50 years in the future.

3. Configure the rule on run on form initalisation.

Here's an example of the above rule:



Further reading

Some helpful resources are listed below:

- https://api.jqueryui.com/datepicker/
- http://www.spiceforms.com/blog/how-to-disable-dates-in-jquery-datepicker-a-short-guide/

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Addresses
- Address Block Samples

Datepicker Localisation in Composer

Unknown macro: 'redirect'

Here's a method for localising the JQuery Datepicker widget included in a Composer Date field.

Compatibility

Since	
Deprecated	

Step-by-step guide

Here's the steps involved:

1. Add the Datepicker localisation JavaScript as a JavaScript library resource to the form. These can be sourced from here: https://github.com/jquery/jquery-ui/tree/master/ui/i18n
2. Add a Business Rule to override the JQuery Datepicker initalisation, configure the rule on run on form initalisation:

   Datepicker Init Override

   ```
   sfc$.datepicker.setDefaults( sfc$.datepicker.regional[ "de" ] );
   ```

   where "de" refers to the German Datepicker JavaScript resource.

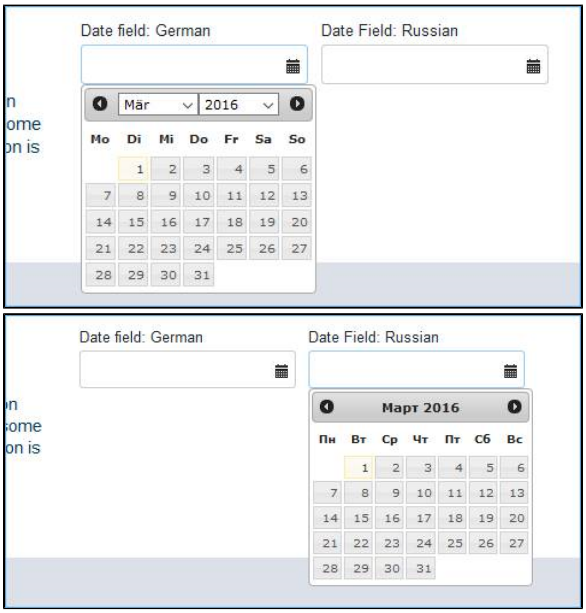
To target a specific a specific date field, you can add a custom CSS class to it and reference it by name in the Datapicker initalisation override Business Rule, for example:

Datepicker Init Override

```
sfc$.datepicker.setDefaults( sfc$.datepicker.regional[ "de" ] );
sfc$(".ruLocale").find("input").datepicker( sfc$.datepicker.regional[ "ru" ] );
```

Remember to add all the appropriate Datepicker JavaScript resources to the form!

Here's an example of the above rule:



Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)

- Addresses
- Address Block Samples

Data Validation (Composer)

Unknown macro: 'redirect'

Contents of this section

Client-side and Server-side Validations

Unknown macro: 'redirect'

Standard validation in Composer forms is to use client-side validation. This type of validation is performed by JavaScript in the browser or PDF.

A question is often asked: Is it possible to maliciously falsify a form in such a way that the form would contain invalid data, but the submission would be accepted by the server?

The short answer is yes, it would be possible to bypass the client-side validation.

There are reasons for this situation:

- Validations in Composer forms, while apparently simple, are actually very sophisticated, and have a high dependence on the visual state of the form, and are not just purely about data. In particular, validation is performed on fields, not on data values. For example, if a mandatory field is hidden (due to some section of the form not being relevant) it is marked as being non-mandatory, even though the form designer marked it as mandatory. This is obviously necessary, because the end user cannot complete a field that is invisible. However, when the form is submitted to the server, only the XML is submitted, and so there is no simple way for the server to determine whether the particular field is visible or not, and therefore not possible to determine whether it's mandatory or not.
- A second reason is that validation rules are implemented in JavaScript, which is a browser-based client-side scripting language. It is difficult to run JavaScript validation rules on the server in a simple and effective way.

For these reasons, it is usually better to re-implement the core set of data-oriented business rules on the server. There are a wide variety of supported techniques for doing this, including:

- Groovy scripts
- LiveCycle Orchestrations
- Validating against XML Schemas

Avoka Transact has a "Forms Submission Preprocessor" service which is called immediately after submission. GroovyScript (or LiveCycle orchestrations) can be coded to perform schema validation or XPath value checks before the receipt is rendered, and before the Confirmation Page is displayed to the applicant. This requires additional effort above the form building process and is not integrated into Composer, however this is the most secure and automated way of rejecting form submissions, whilst providing immediate feedback to the applicant.

Avoka Transact also supports approvals. A form can be routed to an employee to re-validate in a trusted browser environment. This is the simplest way to overcome the issue. Forms awaiting approval apear in an employee group queue as a 'task' and is accompanied by an email. An approved form is handled as a transaction within the management console.

It should be noted that client side validation is still superior to the level of validation provided by paper-based processes. A risk analysis should be performed on a case-by-case basis - sometimes reporting and reconciliation procedures provide enough protection against fraud. Where an identity check is required against third-party services, these are typically called after the submission has taken place (usually as a part of the delivery process), so that fraudulent users can't 'go fishing' for a successful outcome.

Related articles

- How to stop the mandatory marker from wrapping
- Data Validation (Composer)
- Mandatory Marker
- Maguire Error handling on different platforms V4
- Field Validation in Composer

Error Blocks in Composer

Unknown macro: 'redirect'

There are a number of methods for adding an error block to a form. The easiest method is to use the standard error block. This block is included at the end of the form on a standard form, and at the end of each page on a wizard form.

Alternatively you can add any of the three error widgets ‘Error List Block’, ‘Error Section’, ‘Error and Warning Block’. Error widgets can be placed anywhere in the form structure allowing for greater flexibility when presenting errors, the different widgets only really differ in their appearance.

Error blocks are intended to present a collated list of the current form errors to the form respondent. They also provide the ability to jump to a selected error, making it easy to find issues that need correcting. This is in contrast to techniques like in-line validation, where messages are displayed in close proximity to the field they relate to.

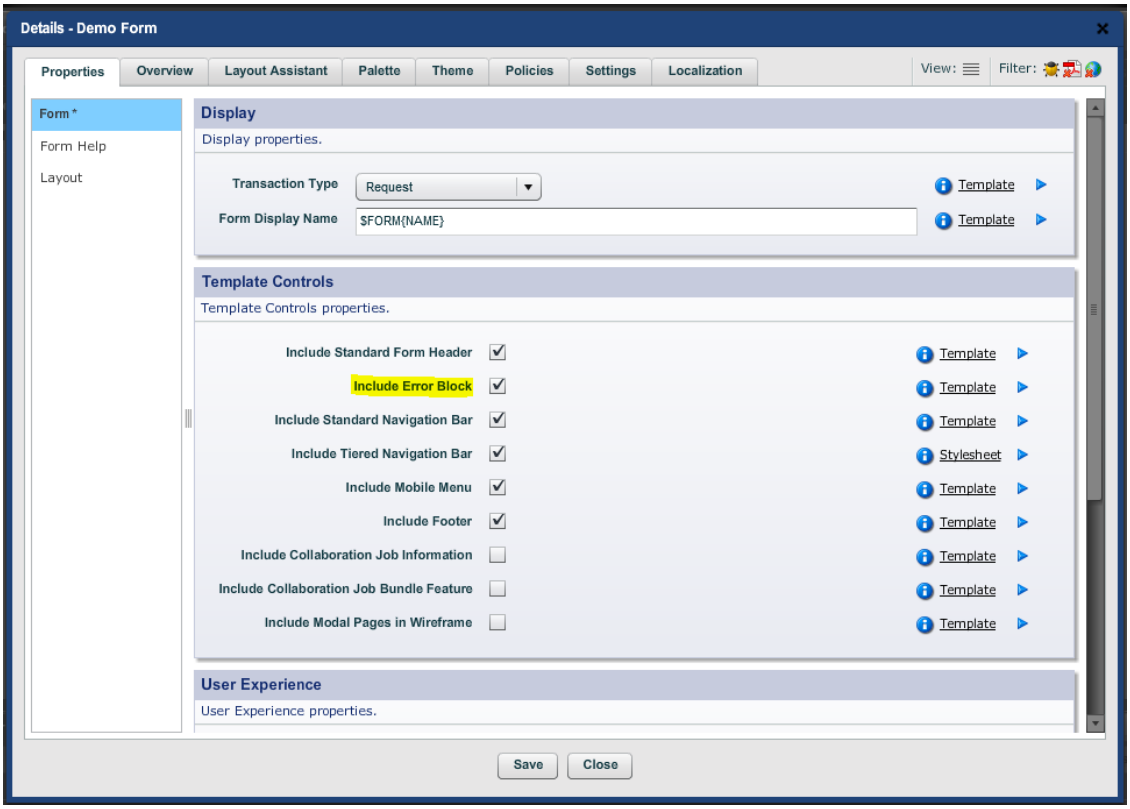
Some of the factors to consider are:

- Do you need an error block, or are you using some other method of displaying errors, such as in-line validation messages?
- Does the error block need to be placed in a specific location, such as the top of each page?

Standard Error Block

The ‘Standard Error Block’ is essentially an ‘Error List Block’ widget which is automatically placed in the form, ensuring it will appear at the bottom of the current form page when errors are present. Read below about the ‘Error List Block’ widget for more information.

This error block is quite easily added or removed from a form. To set whether it should be present, edit the properties of the form (very first item in the structure panel) and check or uncheck the box that says ‘Include Standard Error Block’.



Pros

- Easily added to the form.
- No need to adjust its positioning within the form structure.

Cons

- Positioning cannot be modified. Always placed at the bottom of the current form page.
- Errors are grouped at the end of the form, instead of being collocated with the fields.
- Does not display ‘warnings’, just ‘errors’

Error List Block

An ‘Error List Block’ can be dragged from the Pallet to the Structure in the same fashion as any other widget and can also be repositioned within the Structure similarly to other widgets. It should be noted that the Error List Block doesn’t display ‘Warnings’, just ‘Errors’.

TIP: If you would like an ‘Error List Block’ to be placed at the top of each page in a wizard form, simply include it in the structure before the ‘Section Level 1’ elements.

Pros

- Easily added to the form.
- Can be repositioned as need.
- Possible WCAG 2.0 AA compliance when placed at the top of a form.

Cons

- More effort than the Standard Error Block to implement
- Does not display ‘warnings’, just ‘errors’

Error and Warning Block

For the most part the ‘Error and Warning Block’ is the same as an ‘Error List Block’, except the ‘Error and Warning Block’ as its name suggests also displayed ‘Warning’ messages. Additionally, it has a different visual appearance to the ‘Error List Block’ as seen below.

Pros

- Easily added to the form.
- Can be repositioned as need.
- Displays ‘warning’ messages as well as ‘error’ messages.

Cons

- More effort than the Standard Error Block to implement

Error Section

The ‘Error Section’ widget is essential a ‘Section Level 1’ widget that contains an ‘Error and Warning Block’ widget. The entire section is hidden until validation is run and errors or warnings are present.

Pros

- Easily added to the form.
- Can be repositioned as need.
- Displays ‘warning’ messages as well as ‘error’ messages.
- Consistent in appearance to other form sections

Cons

- More effort than the Standard Error Block to implement
- Should not be placed inside other ‘Section Level 1’ widgets
- Less flexible in terms of visual appearance.

Related articles

- How to stop the mandatory marker from wrapping
- Data Validation (Composer)
- Mandatory Marker
- Maguire Error handling on different platforms V4
- Field Validation in Composer

Field Validation in Composer

Unknown macro: 'redirect'

Applying a validation rule to a field is a way of verifying the data entered into a form. The simplest and most common type of validations are single field validations, which validate the data in a single field. More complex validations use the values of several different fields.

Individual field validation

Fields can be validated using Regular Expression Patterns, Fixed Lengths, or Validation Scripts. When the data entered into a field fails the validation test, an error will be displayed in the form, and the user will not be allowed to submit the form until the data entered has been changed to conform to the validation rule. If a field is not mandatory, and a validation rule is set, a blank value in the field will never cause the validation rule to display an error.

Regular Expression Patterns

Regular expression patterns test whether the data entered into the field conforms to the specified regular expression.

Fixed Length

Fixed length rules ensure that the data entered into the field has a specific number of characters. Any characters greater or less than the specified number will cause the data to fail the validation rule.

Validation Scripts

Validation scripts must return either ‘true’ or ‘false’. When ‘true’, the data has passed the validation rule, when ‘false’, the data has failed, and an error will be displayed.

Sample form: composer production > Navigation and Validation > Validation - Traditional

The sample form contains a field, in which the data entered must be a number, and must be greater than or equal to 10. The script used for this is:

```
var thisVal = Number({Mustbeanumber10});
if (thisVal == NaN) {
   return false;
} else {
   return thisVal >= 10;
}
```

Validating multiple fields

Validation of multiple fields can be done using ‘Business Rule – Error’ widgets.

Sample form: Composer production > Navigation and Validation > Validation - Traditional

The sample form contains an example in which there are two phone numbers, and a validation rule specifying that at least one of the numbers must be entered. The business rule achieves this by using the following Business Error Rule script:

```
return (!sfc.isBlank({Phonenumber}) || !sfc.isBlank({Mobilenumber}));
```

The business rule must also have a ‘focus target’ set. The focus target dictates the behaviour of the error text in the form. When the form is configured to use ‘inline errors’, the error text will appear underneath the focus target. When the form is configured to use an error block at the bottom of the page, the focus target dictates where the user will be directed to when the error message is clicked.

In the sample form, the focus target is set to the parent block containing both of the phone numbers.

Validating one field multiple times (with multiple messages)

A common pattern in building forms is the need to apply multiple validation rules to a field, in sequence.

Related to this is the need to tailor the validation message depending on which validation rule has failed.

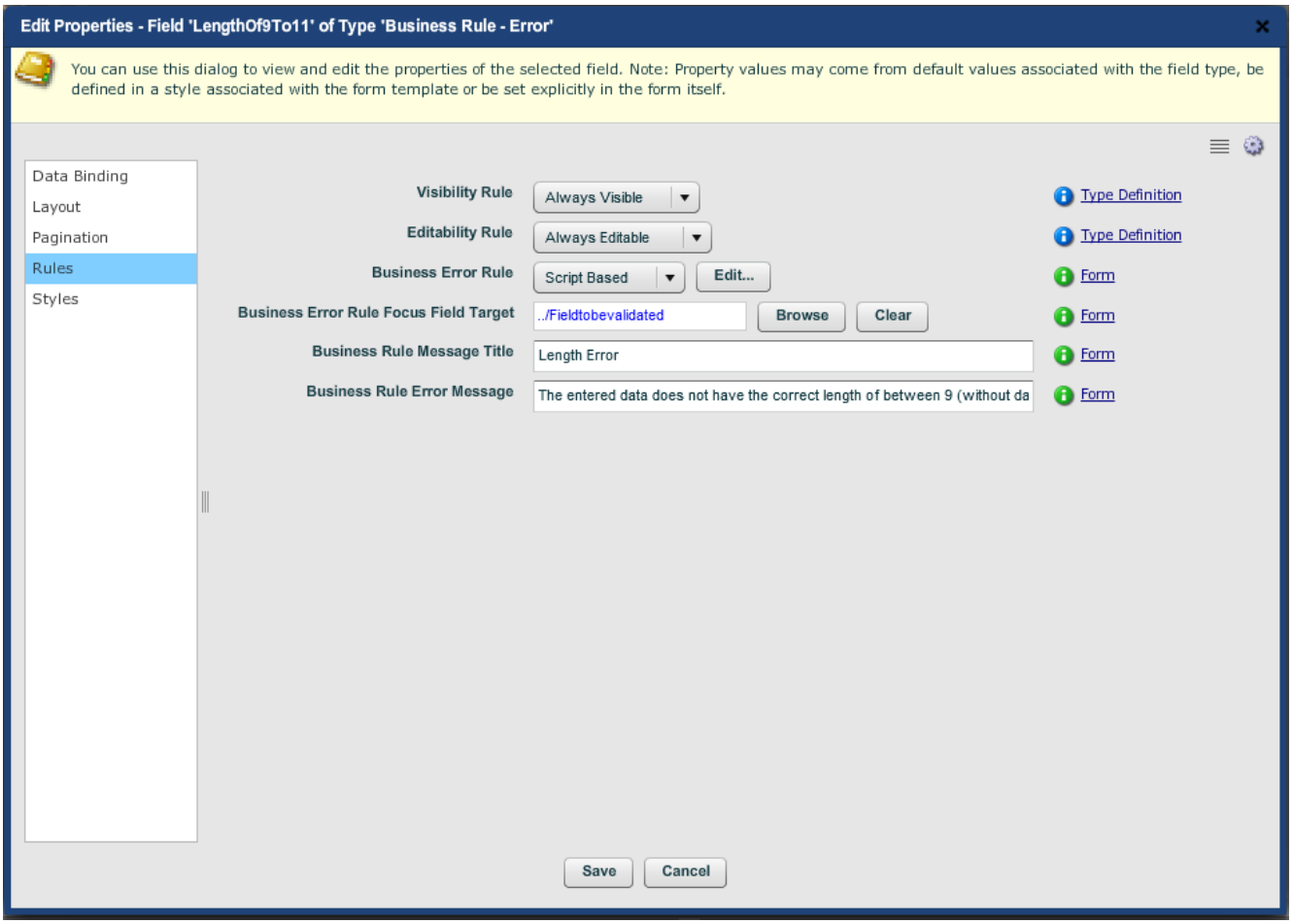
Sample form: composer production > Navigation and Validation > Multiple Validation Rules and Messages

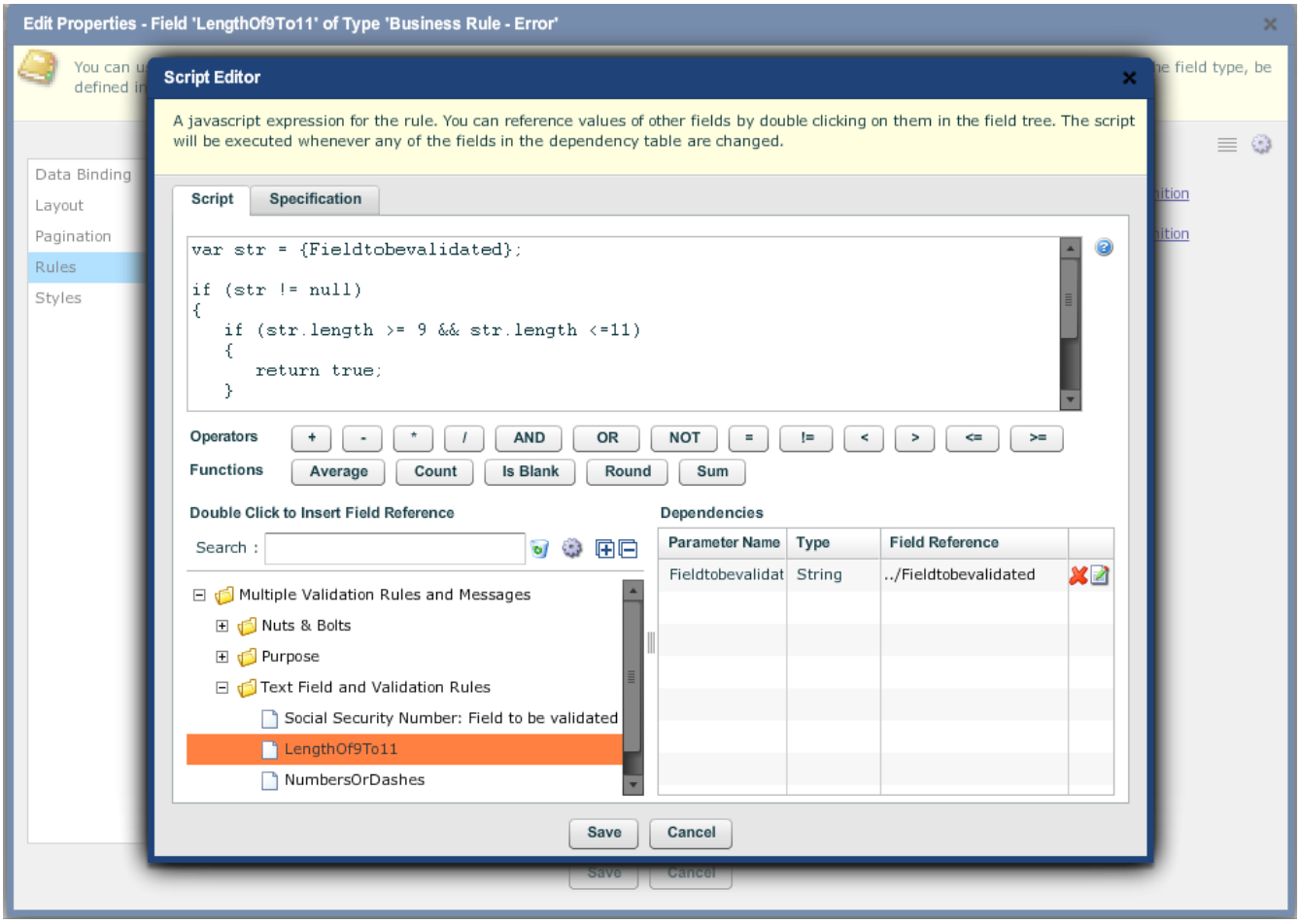
This can be done using multiple 'Business Rule - Error' widgets.

Each 'Business Rule - Error' widget can check particular conditions for a field and can also display a specific validation message if the conditions are not met.

Don't forget to set the target for the widget!

Please note: the target must be a field and NOT a block or section.

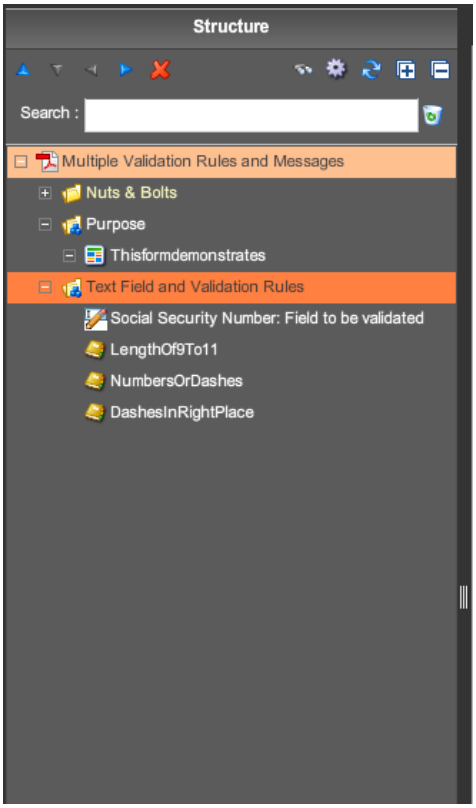




Order of Execution

It's crucial to understand that Business Rules are executed *in the order in which they appear in the hierarchy* moving from top to bottom.

In the Sample Form, there are 3 Business Rules and if you test the form you will notice that the top-most rule is run first, followed by the second, then the third.



Warnings

Fields can also be configured to display a warning when they fail validation, by using 'Business Rule – Warning' widgets. When a field fails the validation on these widgets, a warning will be displayed on the form, and the field will be highlighted, but the form will still be submittable.

Sample form: Composer production > Navigation and Validation > Validation - Traditional

The sample form contains an example, where a field is set to expect a number >= 10, but if anything else is entered, it will display a warning, rather than an error. The Business Warning Rule used to achieve this is the same as the example above, for a field that must be a number >= 10:

```
var testNum = Number({Expactsanumber10});
if (testNum == NaN) {
    return false;
} else {
    return (testNum >= 10);
}
```

Since it is configured using a business rule, the focus target needs to point to the number field.

Related articles

- How to stop the mandatory marker from wrapping
- Data Validation (Composer)
- Mandatory Marker
- Maguire Error handling on different platforms V4
- Field Validation in Composer

Maguire Error handling on different platforms V4

Unknown macro: 'redirect'

Introduction

This article describes how errors are handled in Avoka Transact, and which options are available to beginners and intermediate users.

Avoka Transact handles errors in a framework. Errors are generated by rules which form authors define, and are displayed to the applicant (or form-filler) in a way which is defined by form-level options.

Rules that generate errors can be defined as one or more of the following:

- Mandatory rule (Always, never, rule-based, script-based) on a field
- Validation rule (Regular Expression, Fixed length, script-based) on a field
- Business rule [Error]

The way that errors behave for applicants is dependent on a number of factors including:

- The template used (this article assumes Maguire on Composer 4.0)
- The device used to fill the form (Transact uses the screen width, not the specific device, however there are differences between device vendor browsers)
- The configuration of the form (or style/brand/customization of the template)

In all cases, forms may not be submitted until all errors have been resolved by the applicant.

Concepts

Avoka Transact uses the following features:

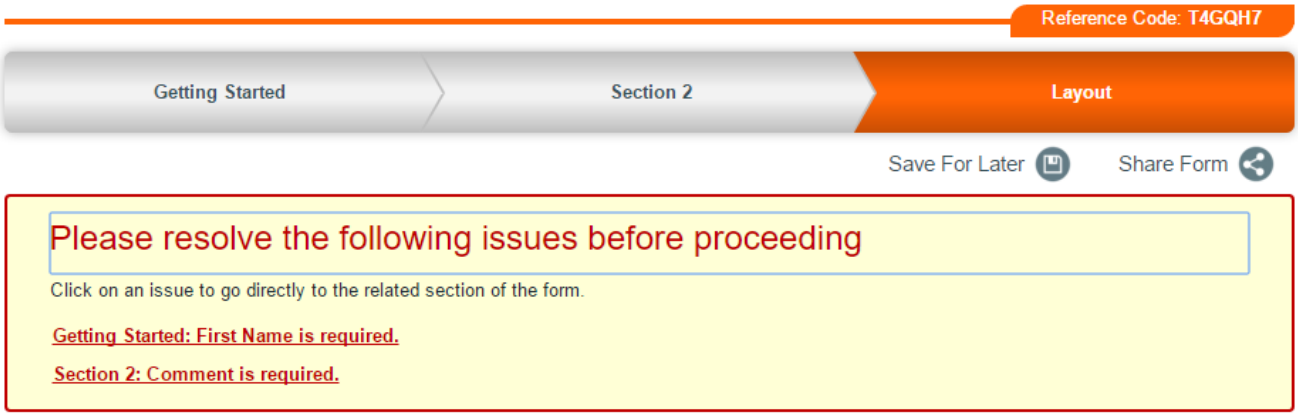
- The Error Block
- Inline Validation
- Error Dialogue

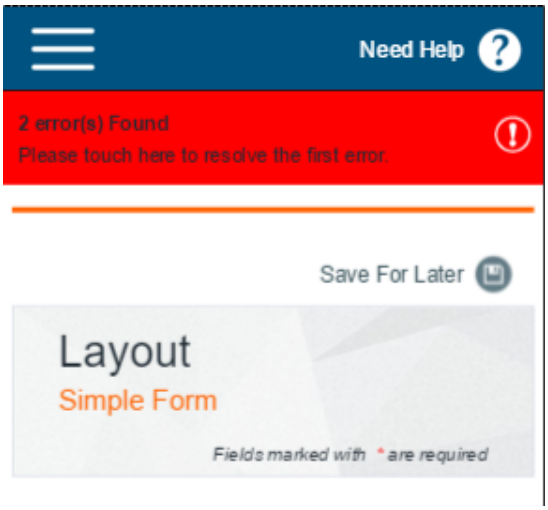
The Error Block

The error block is a special, built-in part of the form template which displays any error messages at the top of the form. Focus will move to the error block upon submission (or page navigation if defined) so that the applicant can see a summary of issues in the form. The error block displays differently, depending on device (screen width):

- On a desktop computer or tablet, it appears as a list of issues, each issue has a description (defined by the field rule or business rule), and behaves as a link to the relevant field. If the user chooses one of the links, the cursor will move to the field, and the screen will re-focus.
- On a mobile device, the error block refactors to a pull-down display. It is optimized for smaller screens, and only displays a "Count" of errors. In this case, when you select it, the focus changes to the first error in the list.

The error block can be switched off in the form object. It can be styled in the Theme Editor and saved into a Stylesheet or Brand.





Inline Validation

Inline validation messages are displayed after the field, after a row of fields, or after a block (depending on configuration). This type of validation typically appears after focus moves out of the field (IE: After an incorrect value is entered and the next field is selected, or a button or some type of control).

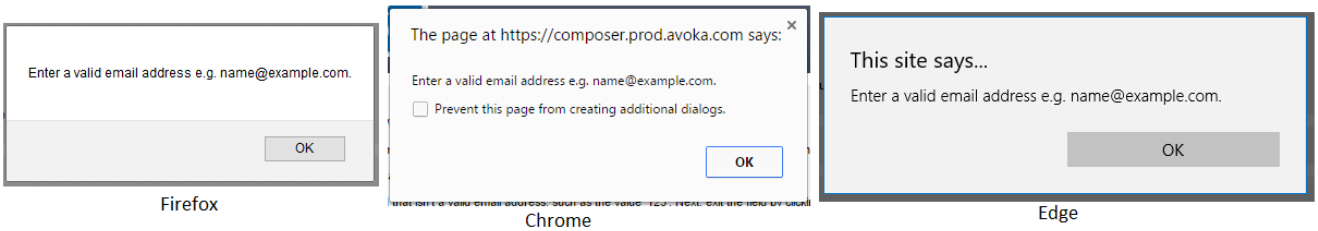
These types of validation messages are intended to be contextual, so can easily be lost if the applicant changes pages, or scrolls away from the field. It is for this reason that inline validation and the error block work hand-in hand.

Error Dialogue

The error dialogue is a pop-up window.

Avoka recommends that for a better user experience that these are always switched off, except in the most extreme of circumstances - for example, the template uses this style of error message to block the user from hitting the browser "Back button" or "close tab" control, so that form data is never lost.

This type of error appears differently in all browser types and versions, and cannot be styled:

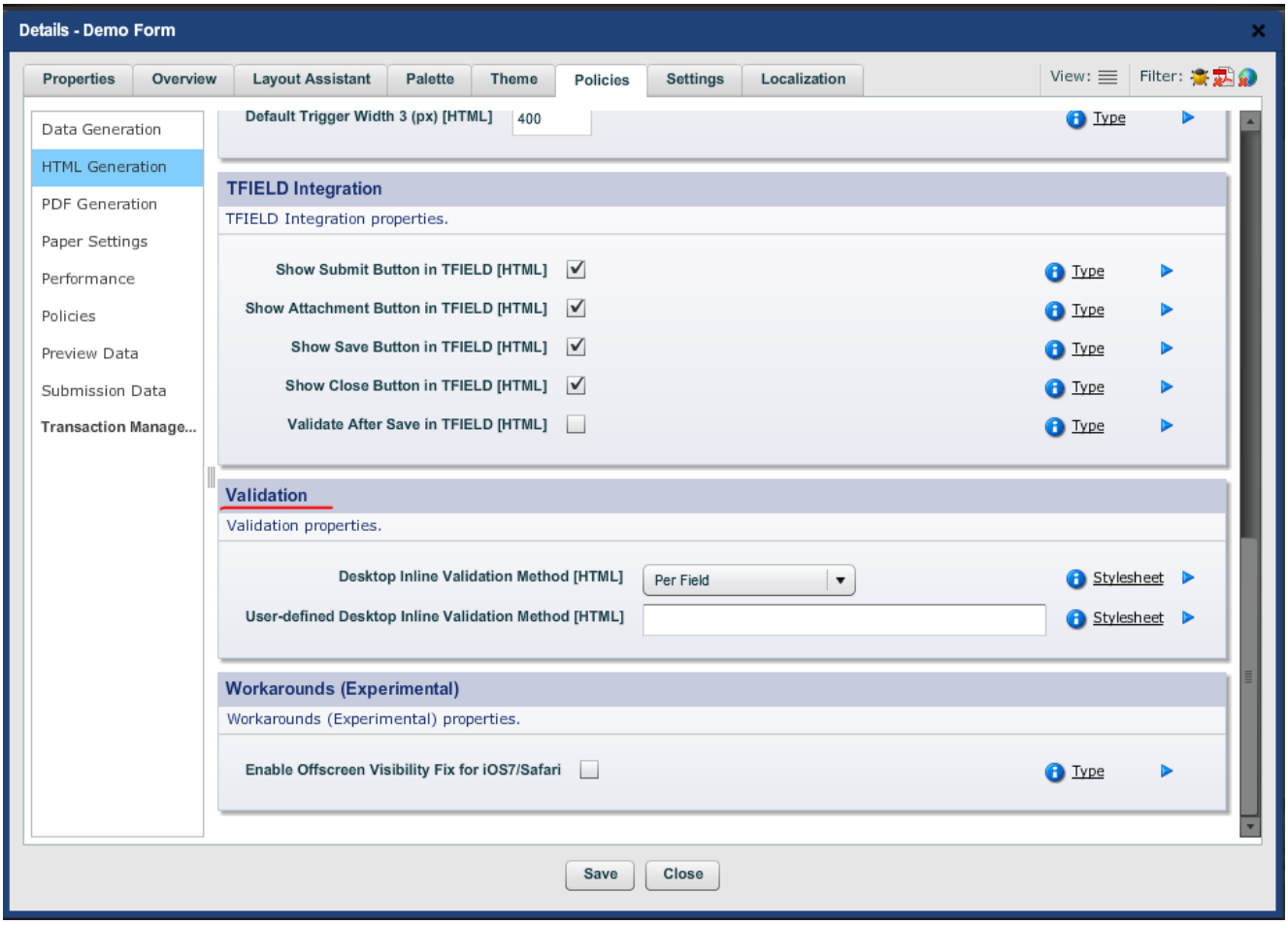


Form configuration

Inline Field Validation Method

This configuration changes where inline messages are displayed. You can find this setting in the following location:

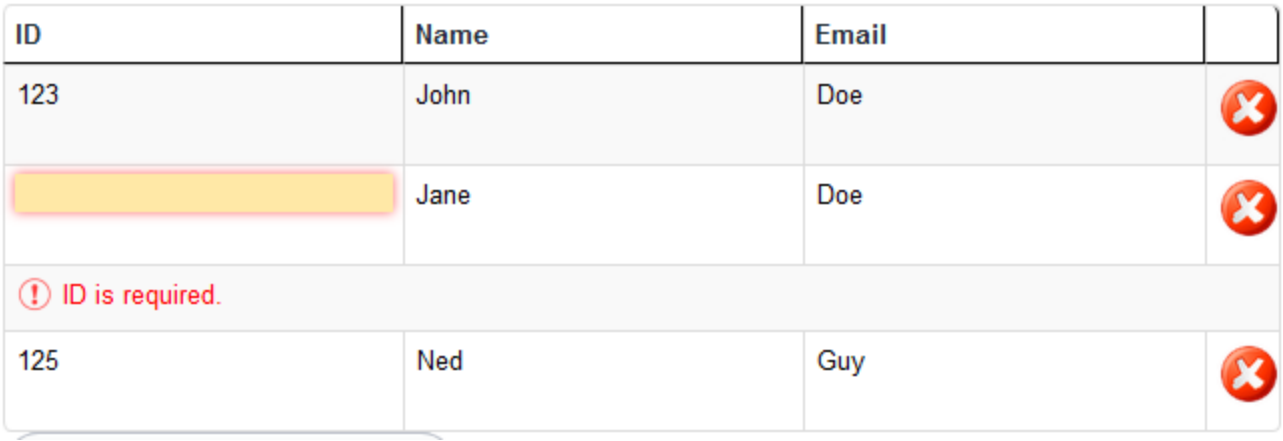
- Double-clicking the root-node of the form structure, and selecting the Policies tab, then the HTML Generation setting.



Changing this value has the following effect:

Note: the main difference between per field and per field block occurs when you have validation rules set on field displayed within a table.

'Per field' - validation messages appear after the appropriate table row.



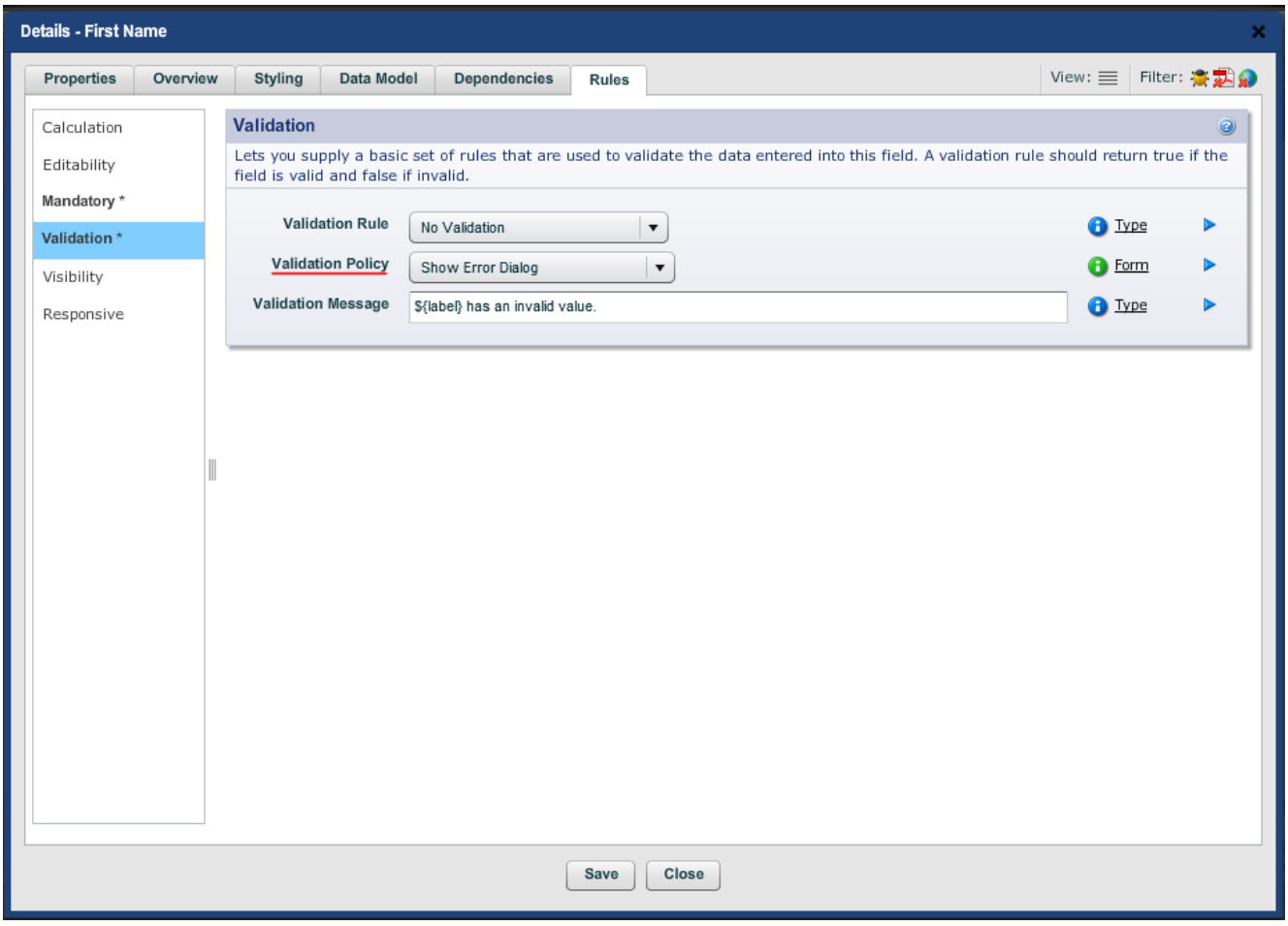
"Per field block' - validation message appear at the end of the table.

ID	Name	Email	
123	John	Doe	
	Jane	Doe	
125	Ned	guy	

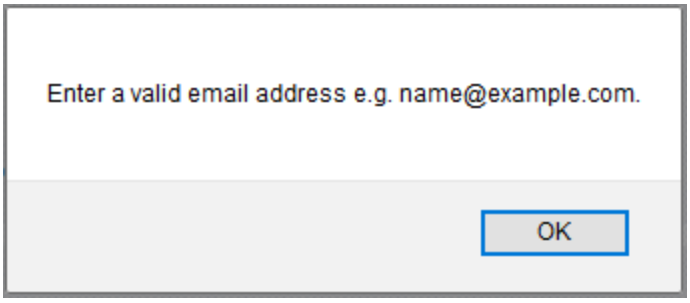
ID is required.

The Error Dialog

- To turn it on this advance property, go to the widget's property editor -> Rules -> Validation -> Validation -> Validation Policy -> Show Error Dialog".



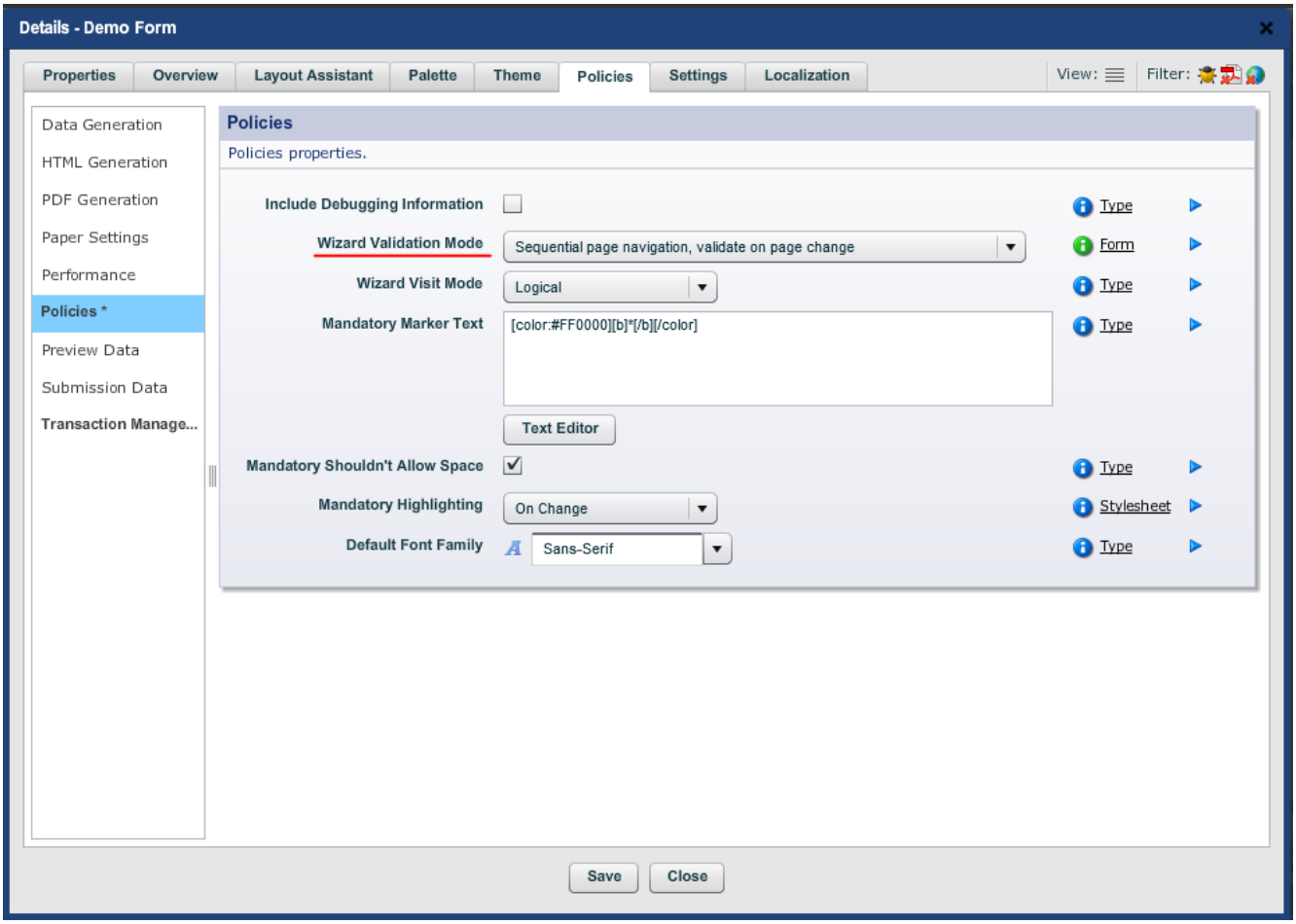
- If the form filler enters invalid information in the field (e.g. Email field), the following popup will become visible:



Page Navigation Policy

The form author can define if applicants should be able to freely navigate pages without interruption, or if applicants must meet all rules before proceeding to the next page. To change this setting:

- Double-click the root node of the form structure, and select the Policies tab, followed by the "Policies" item on the left.
- Change the Wizard Validation Mode setting (as below)



Note that a "sequential page navigation" Wizard Validation Mode can impede testing strategies - most authors only turn this on when the form is ready to deploy into production.

Mandatory Highlighting Policy

By default, mandatory rules are treated a little differently to validation rules. By default, the In-line error messages are **not displayed after focus change**. This is so that applicants can navigate around the page without being bombarded with a message for each and every field. This default behavior can be turned off, to discourage applicants from tabbing-out of fields without entering data. To change this setting:

- Double-click the root node of the form structure, and select the Policies tab, followed by the "Policies" item on the left.
- Change the Mandatory Highlighting setting (in red above)

Related articles

- How to stop the mandatory marker from wrapping
- Data Validation (Composer)
- Mandatory Marker
- Maguire Error handling on different platforms V4
- Field Validation in Composer

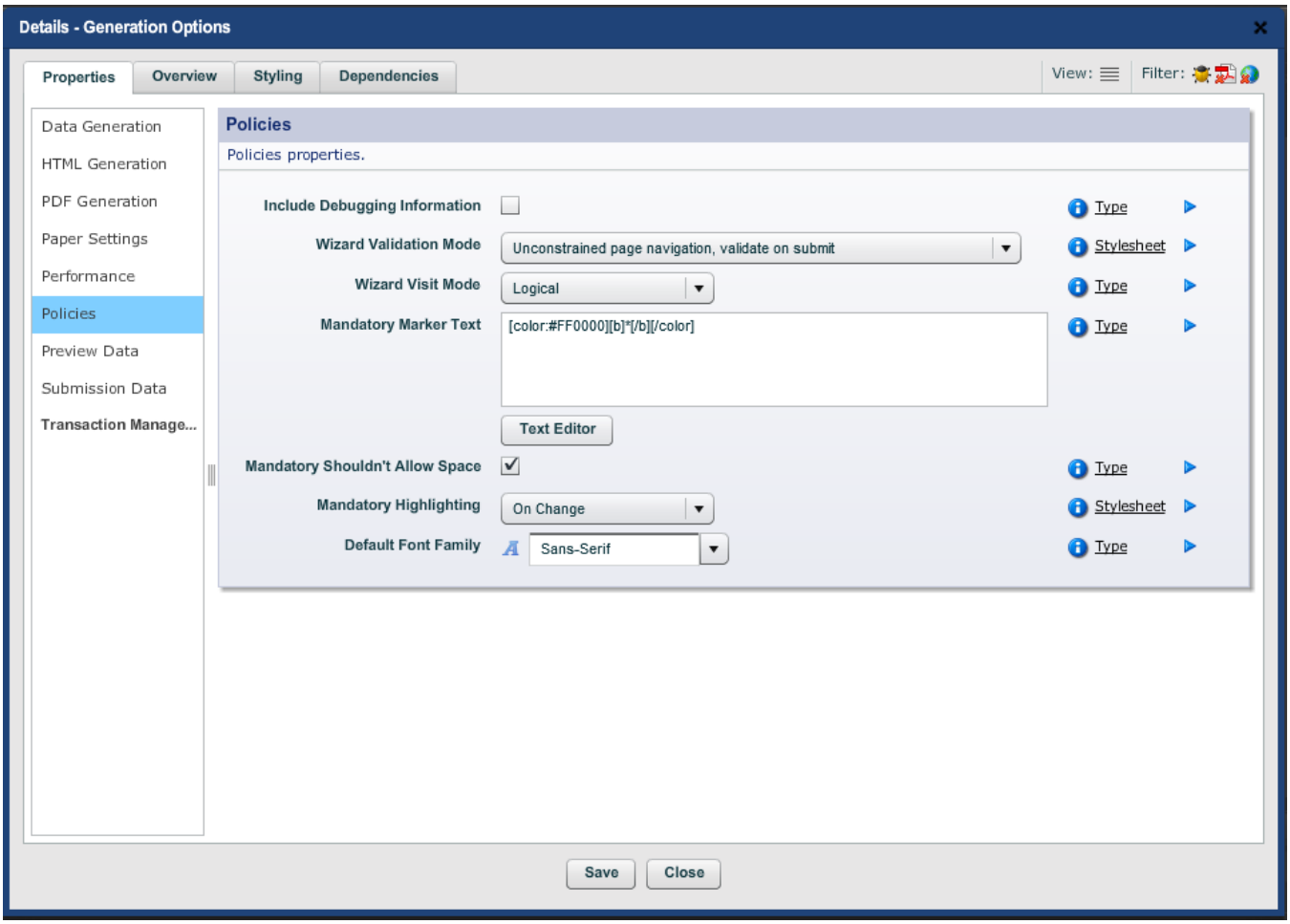
Mandatory Marker

Unknown macro: 'redirect'

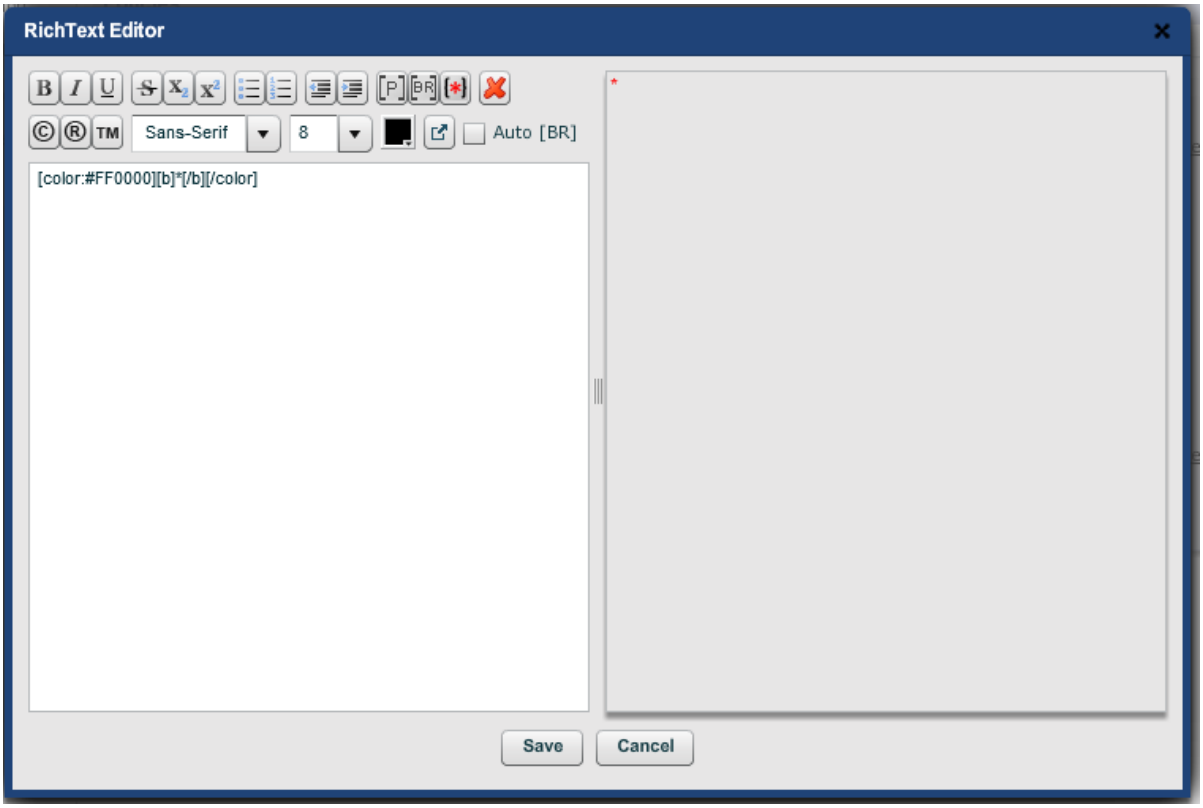
Mandatory Fields (aka Required Fields) are by default marked with a red asterisk (*) appended to the end of each fields caption.

This can be suppressed (if desired):

- Per Field
  - Properties > Rules > Don't Show Mandatory Marker
- Per Form
  - Nuts & Bolts > Form Options > Generation Options > Policies
    - Where the Rich Text editor allows you to alter and format the Mandatory Marker Text property the default setting is [color: #FF0000][b]*[/b][/color]
  - Deleting this text in the Mandatory Marker Text removes the Mandatory Marker from all fields in the form.



The Rich Text Editor now includes a special tag for inserting the mandatory marker for your use.



Related articles

- How to stop the mandatory marker from wrapping
- Data Validation (Composer)
- Mandatory Marker
- Maguire Error handling on different platforms V4
- Field Validation in Composer

Using Business Rules for Data Validation

Unknown macro: 'redirect'

Compatibility

Since	
Deprecated	

Introduction:

Composer offers some common data validation within the standard individual form field types. However, more complex data validation(s) are sometimes required. This article will demonstrate how to achieve a more complex validation by using a business rule.

Problem Description:

This problem was first described on the *Avoka Community Q&A Forum*. The *Composer* form was using the *Attachment Field* to upload attachments. However, the back-end system could not accept filenames that exceeded 60 characters in length. Therefore, the aim is to prevent any files being uploaded having filenames that exceed 60 characters in length.

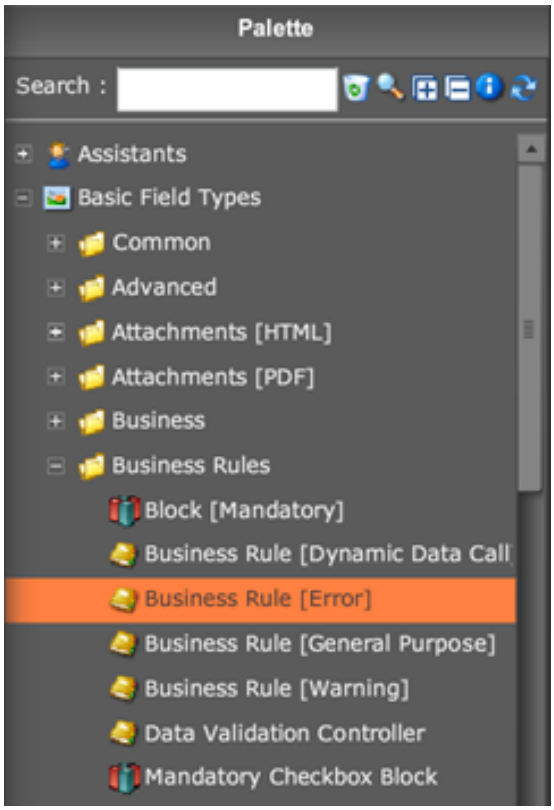
With the standard *Attachment Field* type, it doesn't offer any way of limiting the length of filenames uploaded via the *Composer* form. That is, the user doesn't enter the filename from the *Composer* form. Instead, the filename is returned to the form from the upload facility such that a rule for the filename field cannot be directly defined against this field. In this case a separate business rule is required.

Solution:

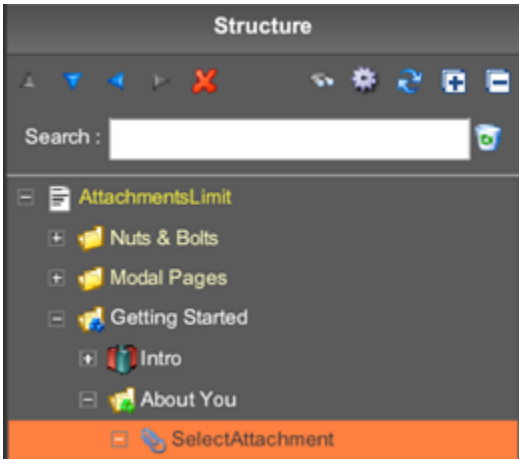
The following explains how to set up a business rule specifically for the problem described above:

Create a business rule for the *Attachment Field* type set up in the form:

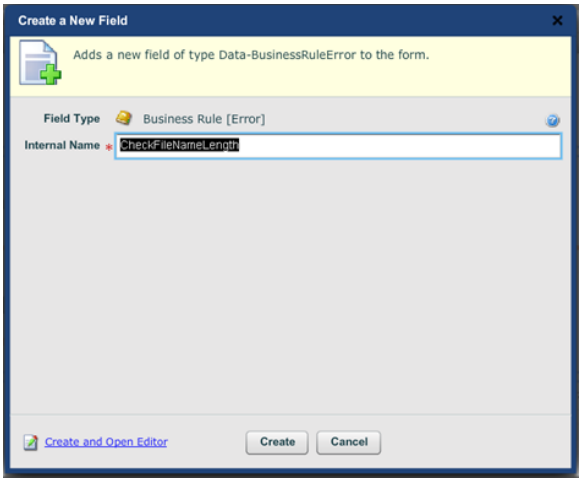
- First locate the *Business Rule [Error]* type which is under *Basic Field Types* > *Business Rules* in the *Palette* panel on the right-hand side.



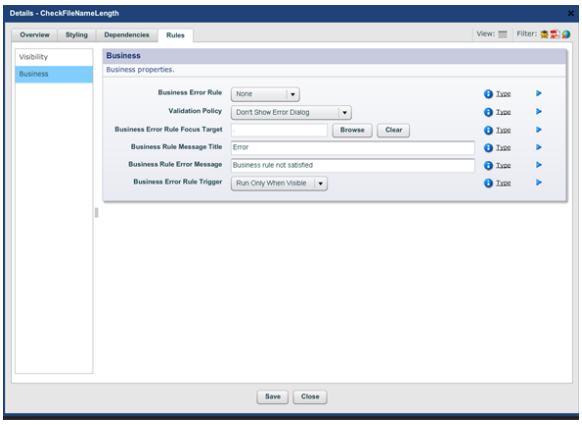
- To use the *Business Rule [Error]* field drag and drop this field to the *Attachment Field* we named *SelectAttachment* in the Structure panel on the left-hand side. It will be shown as a child of this *Attachment Field*. Alternatively, you may double-click on *Business Rule [Error]*. Before doing this, check that the *Attachment Field* in the *Structure* panel on the left-hand side is selected.



- A *Create a New Field* dialog will appear. Enter a meaningful *Internal Name* and click on the *Create and Open Editor* link at the bottom left-hand side of the dialog. A *Details* dialog will appear. Alternatively, if you clicked the *Create* button from the *Create a New Field* dialog, simply locate the new business rule on the *Structure panel* and double-click on it to open the *Details* dialog.

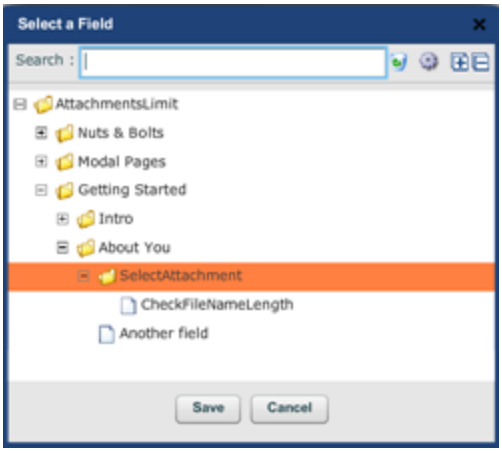


- On the *Details* dialog, click on the *Rules* tab from the tab menu and click on the *Business* rule from the panel on the left-hand side. This is where the business rule for preventing filenames exceeding 60 characters from being uploaded will be defined.

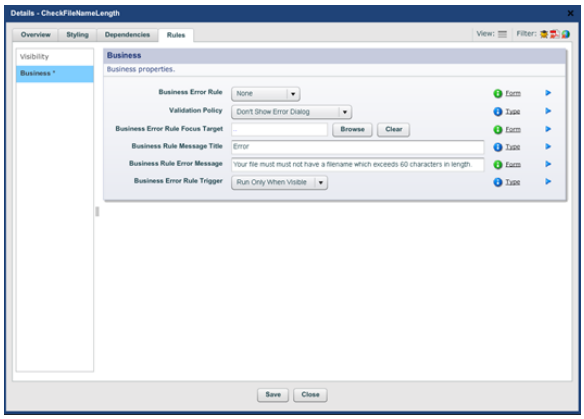


- The following steps will set up this business rule:

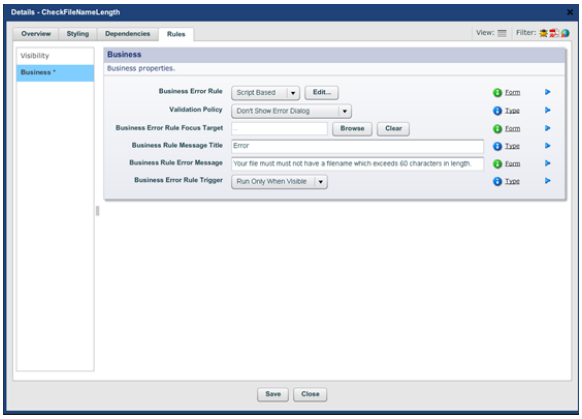
    - The business rule will need to point to a target field. That is, in this case the business rule is for the *Attachment Field* we've named *SelectAttachment*. Click on the *Browse* button next to *Business Error Rule Focus Target* field and *Select a Field* dialog will appear. Locate the *Attachment Field* and click on it. When the *Attachment Field, Select Attachment,* is highlighted, click on the *Save* button.



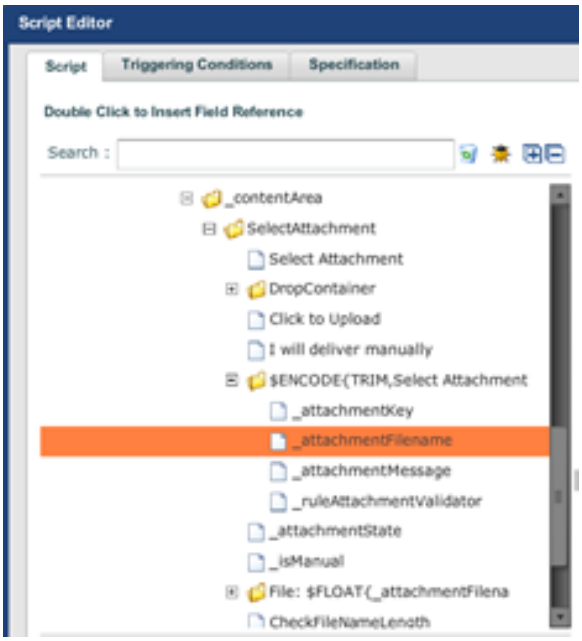
- Change the default *Business Rule Error Message* to something more specific and meaningful. This is the error message returned to the user and specifically for our example will be shown if the user attempts to upload a file with a filename exceeding 60 characters. So we changed the *Business Rule Error Message* to *Your file must not have a filename which exceeds 60 characters in length*.



- The final steps are to define the business rule in JavaScript code. Locate the *Business Error Rule* field at the top of the *Business* rule fields. Select *Script Based* from the dropdown and click on the *Edit* button that appears alongside it.



- The *Script Editor* dialog will appear. As was explained above, the filename is not entered directly into the Composer form but returned from the upload facility. The filename must then be located in the internal fields of the *SelectAttachment* field type. Click on the cog icon to S*how internal fields.* Locate and double-click on the *_attachmentFilename* to select it as shown below:

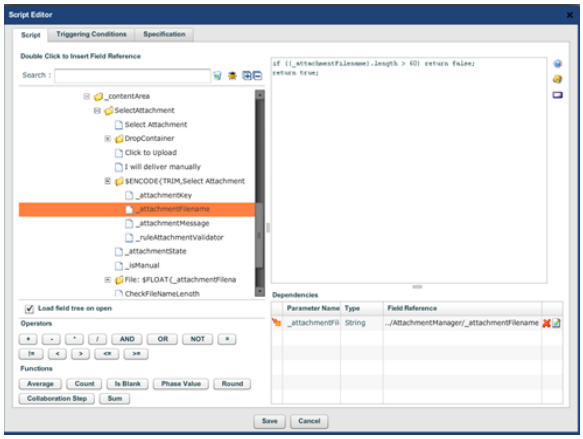


- Then for checking the filename length does not exceed 60 characters, the code shown below will return false when the filename exceeds 60 characters otherwise true will be returned. Whenever false is returned the error message entered in *Business Rule Error Message* will be displayed. Enter the following JavaScript code in the panel on the right-hand side:

CheckFileNameLength

```
if {{_attachmentFilename}.length > 60} return false;
return true;
```

- Save the changes by clicking on the *Save* button from the *Script Editor* dialog and then clicking on the *Save* button from the *Details* dialog



- Now we can test that it works by:
  - Save and preview the form. Attempt to upload a file having a filename which exceeds 60 characters in length and the following error message will be displayed:

Dynamic Data

Unknown macro: 'redirect'

- Data Lookup
- How to create a Dynamic Data lookup using test data
- Repeating and Nested data in Dynamic Data service

Data Lookup

Unknown macro: 'redirect'

Data lookups are implemented using a Dynamic Data service button widget, which links to a groovy data service set up in Transaction Manager.

The button is linked to the service definition by entering the Transaction Manager service name.

The button needs to be pointed to an input block, and an output block. The service matches the variables to the fields by name - field names in the input block must match the input variable names in the groovy script, and output parameter names from the groovy script must match field names in the output block.

The groovy script returns a JSON object containing the parameters returned from the service to the form.

For a more detailed walk-through of how to develop a Dynamic data lookup, refer to the article *Developing a Dynamic Data Lookup*.

Sample form: Composer production -> Cookbook -> Dynamic Data Service -> Dynamic Data Service, with the published form on Transaction Manager demo.

For a more advanced implementation of a Data lookup service, refer to the article *ABN Lookup Demo*.

ABN LOOK UP:

This document refers to the ABN lookup Demo form on composer production -> Cookbook for Clients -> Dynamic Data Service -> ABN Lookup Demo.

The ABN Lookup Demo form uses the ABN Lookup - Environment Service on Transaction Manager demo.

The ABN Lookup - Environment Service is an advanced ABN lookup service, which has some built in logic to handle things like suppressed legal names, cancelled ABNs, and safeguards against outages of the Australian Government ABN lookup service.

The logic used in this form and accompanying Dynamic Data service is based on the logic used by several Australian Government clients in PDF forms developed by Avoka in the past.

Related articles

- Developing a Dynamic Data Lookup
- How to create a Dynamic Data lookup using test data
- Repeating and Nested data in Dynamic Data service
- Data Lookup

How to create a Dynamic Data lookup using test data

Unknown macro: 'redirect'

Introduction

This article describes how a form author can create a Dynamic Data look-up using test data. This method can be used to create the UI before involving developers.

The advantages of following this technique are:

- Form authors can prototype the solution without involving developers
- When the prototype is finalized, the test data contains everything a developer needs to begin work on integration

The risks associated with this particular technique are:

- The form author could design something which is impossible to integrate (IE: impractical or insecure), and the UI will need to be redesigned. Mitigate this risk by involving developers early (test data design), or using alternate techniques such as defining the services first and UI second.
- There could be some iteration to accommodate changes required by the developer (we touch on this at the end of this article)

This guide assumes a simple scenario, where one service call is required to bring back all the data in one click. More complex scenarios (such as search and select or multiple service calls) can be extended from this basic example.

How it works

The Transact Dynamic Data ***widget*** is a special button which functions in the following way:

- When the button is pressed, it calls an arbitrary service on Transaction Manager.
- It uses an input block as parameters to that service call.
- It returns the results to the output block.

Therefore, you only need to define those three things, but you need to get the wiring right. Here's a few things you need to know:

- Input parameters and output parameters are passed to the Transaction Manager Service as JSON formatted data.
- In it's most simple form, JSON is a list of key-value pairs.
- The JSON key must be named the same as the field name (not the field label).

Thankfully, as a form author, you can leverage the Dynamic Data Assistant (Composer 4.0) to make all of the above easy.

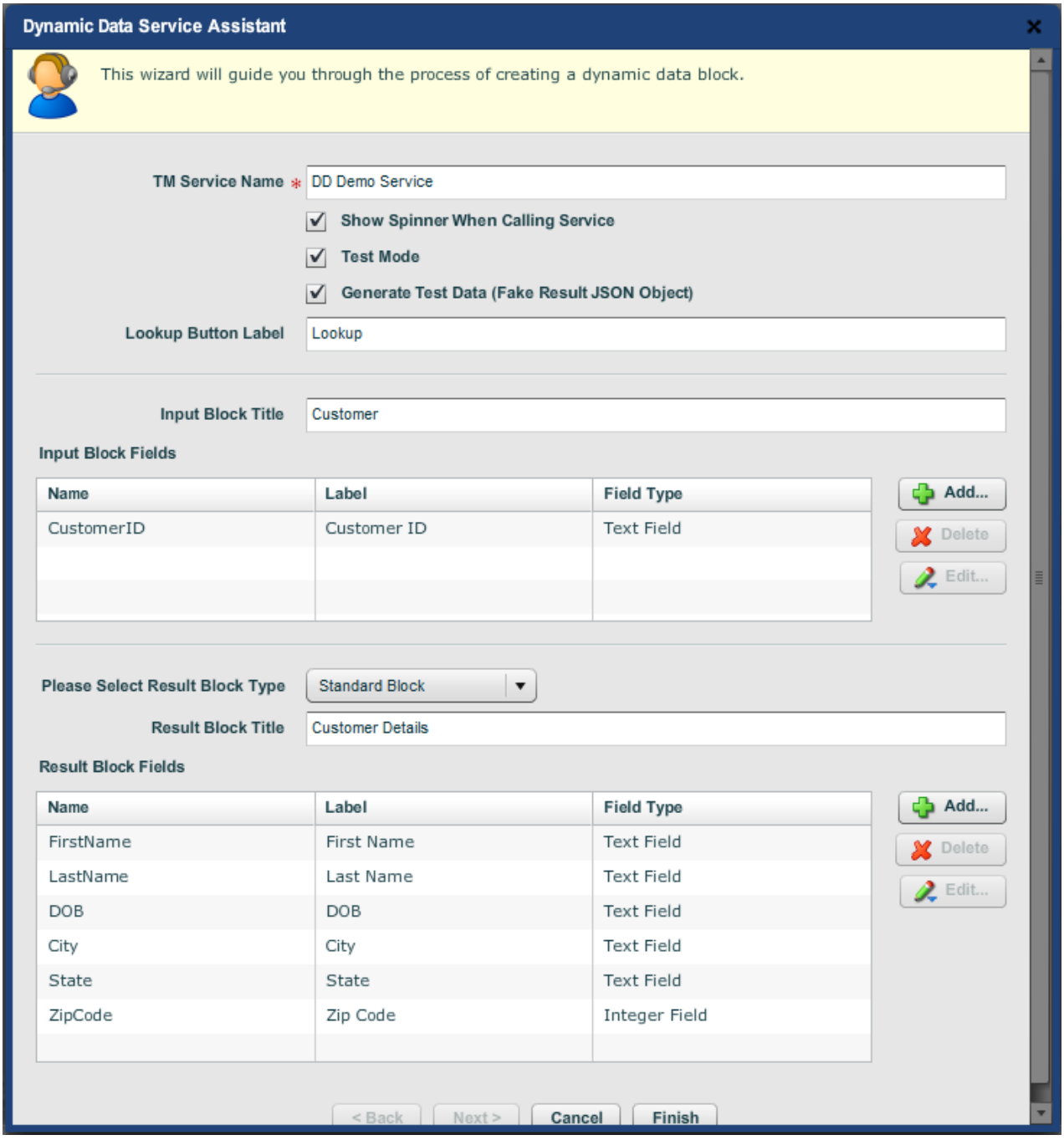
Building your first Dynamic Data section with the Dynamic Data Assistant

The Dynamic Data assistant assumes that nothing is built. It creates all the fields and test data for you. Just choose a place to put it and double click the assistant. Then all you need to do is:

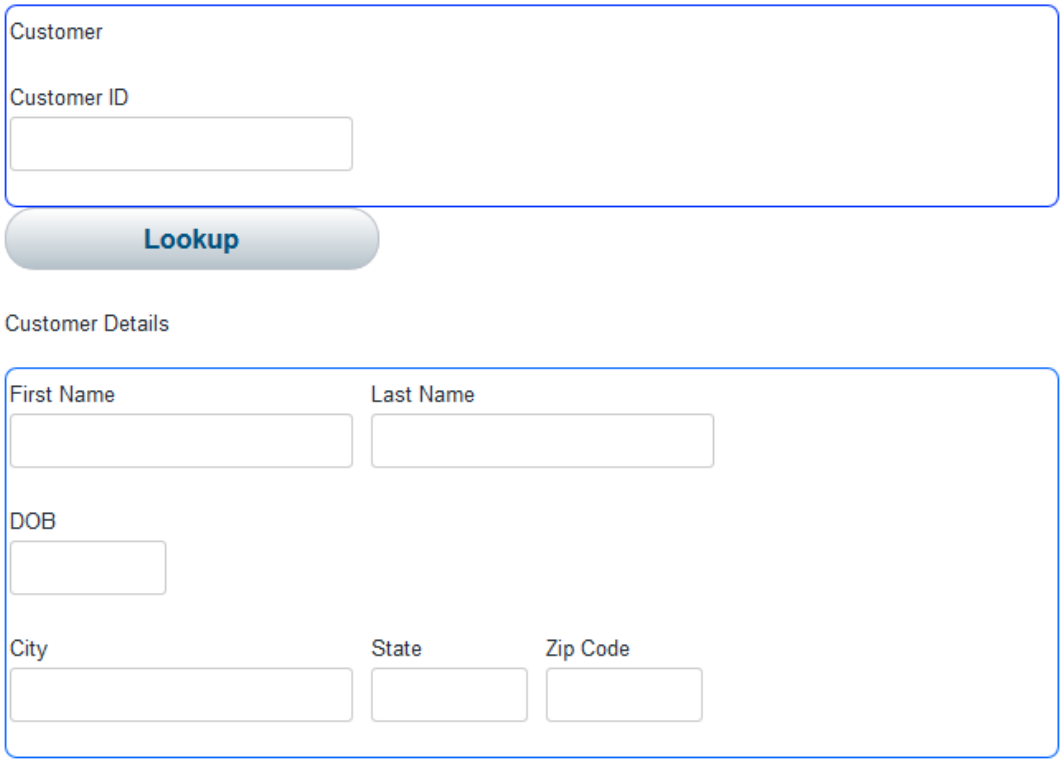
- Define the TM Service Name (This will be the name of the service on Transaction Manager)
- Select "Test Mode"
- Give the button a label
- Name the input block, and define which fields will be used as parameters to the service call
- Name the output block, and define which fields will be returned.

You can begin to define what field type is returned, or you can just return all data in a text field, and convert the field to a "Data Field" later. (Data Fields are hidden from the UI, but can be used by calculation rules or visibility rules in the form if the data needs to be reformatted or placed in more than one section of the form).

Here's an example of an initial design:



After some very basic formatting of the fields, here is the result in HTML:



Editing fake data

You can edit the fake data returned by following these steps:

- Edit the properties of the Dynamic Data Button ("Retrieve Customer")
- Under "Properties" > "Data", edit the "Test Data (Fake result JSON Object)":

> *[{ "AddressLine1":"Suite 25", "LastName":"Jones", "FirstName":"Henry", "City":"Springfield", "AddressLine2":"1024 Binary Road", "State":"NS", "DOB":"1970-11-29", "Zip":"90210" }]*

Beginners note: you can copy and paste the data into your favorite text editor (such as Notepad++) to see contextual highlighting (as JavaScript). You can also add as many new lines or spaces as you like, or re-order to make it easier to visualize.

Next Steps

As a form author, you will want to format the returned results, so that they look nice in context of your form. You can either:

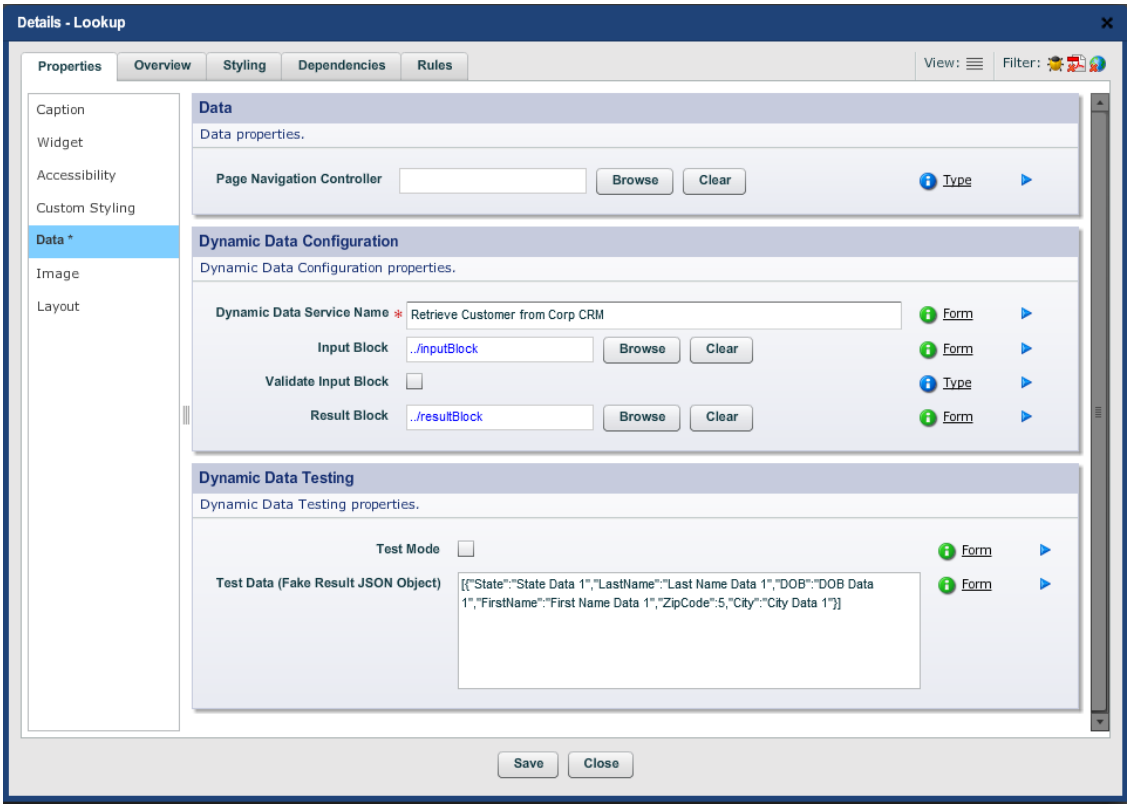
- Format the fields to suit your needs (Constrained within the block), OR
- Right click on the fields, and select "Change Field Type", and change to "Data Field":

Changing to a data field makes the returned result invisible, so that you can process the data to suit your specific needs. You can then use calculation rules to reformat or simply copy values.

Handing over to a developer

When you're happy with the look and feel of the UI, you can hand over to a developer to create a real service. Just edit the Dynamic Data button, and navigate to the "Properties" > "Data" section to retrieve the relevant information that can be used in the specification. The specification typically includes:

- The URL to the published form (This tells the developer where to go to develop the service)
- The name of the service (In this example, it's called "Retrieve Customer from Corp CRM"
- Sample JSON result data (The test data)
- The names of the input fields (not the labels)



Also, don't forget to disable "Test Mode" when you publish to Transaction Manager.

Editing the user interface

Sometimes it's necessary to take another pass at the UI, as a result of integration requirements. In this example we never added a system control field for exception handling. (What happens when no results are returned? What if a web service call fails or system-level authentication fails?)

To add a new data element to the Dynamic Data section, **you only need to do one thing**: Just add a field to the results block. However, you need to take care that the name (not the label) of the new field matches the expected JSON key.

Detailed scenario...

A typical scenario might progress like this:

- The developer might ask: "Where shall I put the system return information, such as service return codes?"
- You then need to agree a new JSON key-value pair. Let's call it '***system***'. The value will contain any error messages that the service returns. We can define return codes, or just put error messages in here.
- This means that all **you** need to do is drag a new field onto your form, in the results block with the name ***'system'*** (The name is important, but the label can be whatever you want)
- You should probably update your form test data to contain the new data (don't forget to turn test mode back on). Just add the following text before the curly brackets:

> *, "system":"500: An application error has occurred"*

Now, in reality, you would not want to show this error message to an end user, but:

1. It helps the developer debug the service as he is building it
2. You can build business rules into your form to deal with production issues appropriately
3. Any issues that had occurred are then saved in the transaction XML

Related articles

- Developing a Dynamic Data Lookup
- How to create a Dynamic Data lookup using test data
- Repeating and Nested data in Dynamic Data service
- Data Lookup

Repeating and Nested data in Dynamic Data service

Unknown macro: 'redirect'

Dynamic Data services are services stored on the Transaction Manager (TM) that Composer forms can access in order to retrieve data that can be used in the form. Typically dynamic data services access external web services to retrieve data to be used in a form, or to validate data entered into a form.

Currently, the standard 'TM Dynamic Data Button' widget in Composer is limited in that the fields in the Result Block must be in a flat format i.e., they must all be direct children of the result block, and any fields nested in other blocks within the result block will not have data populated by the Dynamic Data service. The implications of this are that nested data, such as data that contains repeating elements cannot be handled by the 'TM Dynamic Data Button' widget.

To get around this, you can use a basic push button on your form, and call the sfc.**getDynamicDataFromJson**() function to make the call to the TM Dynamic Data service, then handle the resulting JSON manually.

For example; a Dynamic Data service on TM that returns the following JSON:

```
{

    "start2":"bbb",
    "start1":"aaa",
    "array1":[
      {

          "item1":"xxx",
          "item2":"yyy",
          "item3":"zzz",

      },
      {

          "item1":"xxx",
          "item2":"yyy",
          "item3":"zzz",

      },

      {

          "item1":"xxx",
          "item2":"yyy",
          "item3":"zzz",

      }

   ],
}
```

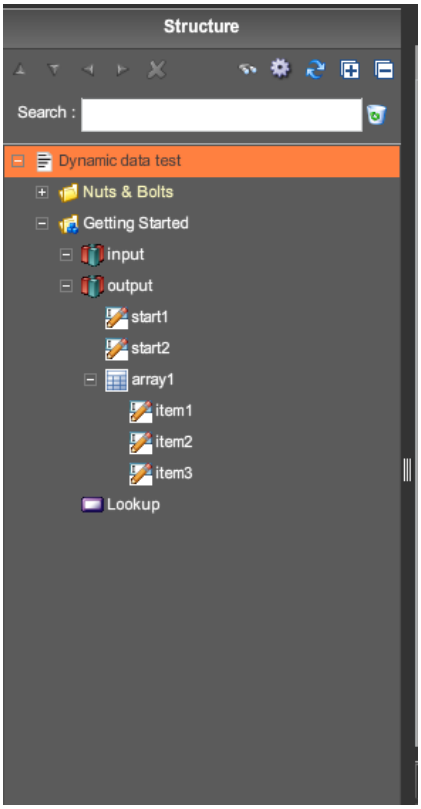
could be called and the resulting object imported into a form using the following code:

```
var inputJson = JSON.parse('{"test":"test"}');

sfc.getDynamicDataFromJson(me, "bbTest", inputJson,
    function(result) {
        var recurse = function(dataNode, formNode) {
            var outputChildren = sfc.gatherChildNodeInformation(formNode);
            for (var i = 0; i < outputChildren.length; i ++) {
                if (Object.prototype.toString.call(dataNode[outputChildren[i].nodename]) === '[object Array]') {
                    var im = sfc.findInstanceManager(outputChildren[i].node);
                    im.setInstanceCount(0, false);
                    for (var j = 0; j < dataNode[outputChildren[i].nodename].length; j ++) {
                        var newNode = im.addInstance();
                        recurse(dataNode[outputChildren[i].nodename][j], newNode);
                    }
                } else if (dataNode[outputChildren[i].nodename] != undefined){
                    sfc.setRawValue(outputChildren[i].node, dataNode[outputChildren[i].nodename]);
                }
            }
        }
        var jsonData = result.evtData.resultData;
        recurse(jsonData, {output});
    },
    function() {
        sfc.debug('lookup failed');
```

```
    }
);
```

This code will map the data returned in the JSON object to a field with the same name, in the same way that the standard TM Dynamic Data Button does, but with the exception that when it finds an array in the repeated data it will recurse into the array and map the fields within that object.

In order for the data to be mapped correctly, the fields in the form must match the structure of the JSON returned by the Dynamic Data service, i.e.,



The 'output' block must also be configured as a dependency in the button script, in the line:

```
recurse(jsonData, {output});
```

A working example of this form can be viewed here.

Related articles

- Developing a Dynamic Data Lookup
- How to create a Dynamic Data lookup using test data
- Repeating and Nested data in Dynamic Data service
- Data Lookup

Layout & Responsive Design

Unknown macro: 'redirect'

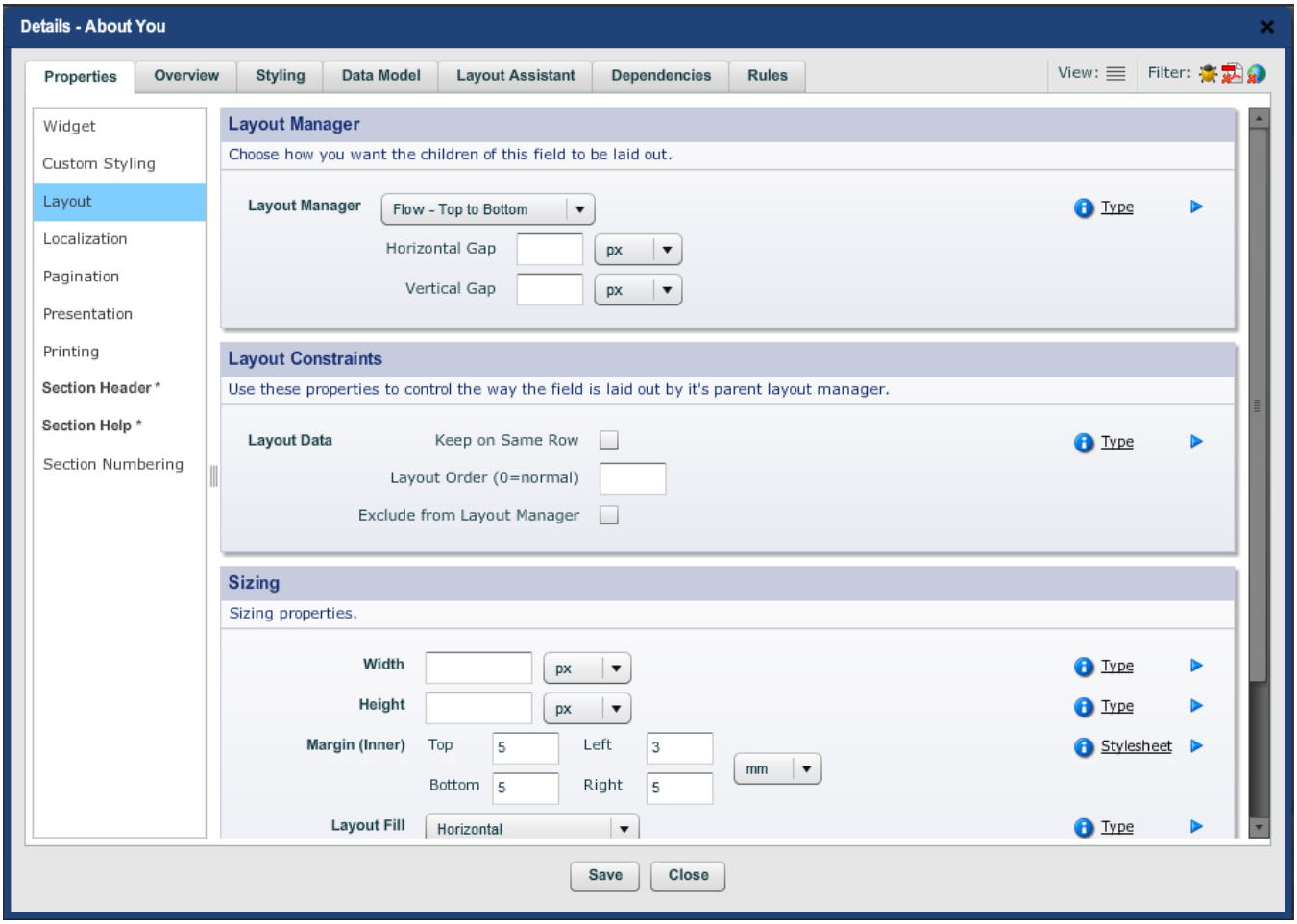
- Composer Layout Features
- Building responsive forms in Composer with the Maguire Template
- Layout Data options for fields in Composer
- Adjusting the Maguire Section Level 2 header positioning in Composer
- How to stop the mandatory marker from wrapping
- Abandonment Support V4
- Form Object using the Maguire template V4
- Tracking Code V4
- Turning "Go to" questions into "hide-show" logic

Composer Layout Features

Unknown macro: 'redirect'

First off it is very important to note, **don't always trust the wire frame.** Always test your form in the format your users will view your form in. It is best when creating forms for multiple formats to try to make the form wire frame look as close to the preview as possible. Consistency in the wire frame will carry over in PDF and HTML formats, and usually less layout issues.

Features:



Keep on same line

- Based on the parent before it
- Contingent on a horizontal layout fill
- Disappears on a vertical layout fill

Horizontal Layout Fill:

- Extends a fields width to 100%

Layout Manager:

Flow- Left to Right:

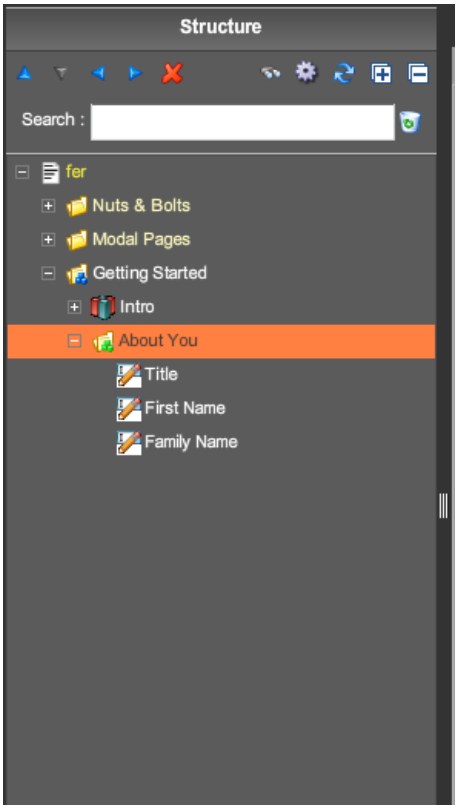
- Allows you to lay out several fields across the page with very fine-grained control over the size of each field by specifying the percent width of each field.
- Horizontal Layouts are always used in conjunction with a Vertical Flow Layout - generally the Horizontal Layout is used for each line of fields, with each line being laid out vertically.

Note: While we generally talk about percentages, those percentages need not to add up to 100: Composer will adjust accordingly.

The beauty of using percentages is that if you need to add another field onto the same line, move a field, re-size a field, or switch from portrait to landscape, all the other fields will automatically be adjusted to accommodate the changes and form maintenance then becomes easy. Also, the fields will retain their proportions on mobile devices with different screen widths.

Horizontal Layout

Horizontal Layout in the Hierarchy



Flow-Top to Bottom:

- A lot of forms are very simple - they just contain a series of fields going down the page. This is also the best layout for the narrower screens on mobile devices.
- To achieved this type of layout, you can use a Vertical Flow layout manager,
- Fields in a Vertical Flow simply go below each other down the page. The Vertical Flow is always the default manager on all sections and blocks.
- The Vertical Layout Manager does not change the width of any of your fields (unless you select Layout Fill). Your field width will be: The "natural" width of the field as defined in your organization style sheet; or the style-set that you associated with this field when you dragged it into the form.
- If you select Layout Fill, the layout manager will try to use as much of the line's space as possible and take into account the size of the screen or page, margins, borders and gutter areas, space allocated to menus, and the other fields on the same line.

Note: If you specify too many fields on the same line, this can create layout problems, particularly on smaller screens. The behavior of a line with too many fields will vary depending on the technology that is used to render it:

Recognizing Layout Issues:

In the Wireframe, your page will appear to be "chopped off" on the right hand side if the fields can't all be fitted onto the same line.

- In PDF forms, the fields that overlap the edge of the page will generally be wrapped around to the following line.

- In HTML, the fields that overlap the edge of the page will generally extend over the edge of the section. This can vary depending on whether the overall page itself is has a fixed or variable width.

Using Horizontal and Vertical Layouts together

You often can tweak layouts by mixing horizontal and vertical layouts.

Multiple fields per line

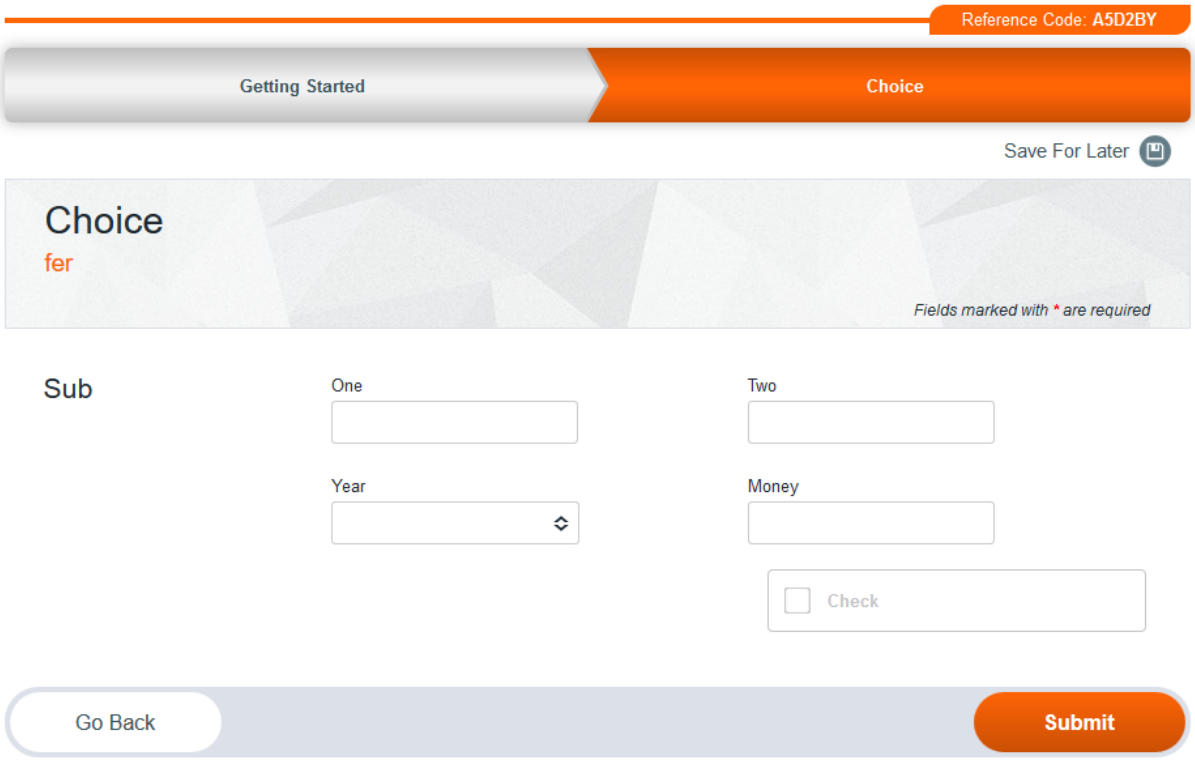
The simplest way to get multiple fields on the same line is to use a vertical flow layout, and using the "Keep on Same Line" property. However, this gives you very little control over the sizing of the fields.

For more control, use a horizontal layout within a vertical flow to finely control the widths of each of the fields in the single line. Using even multiples of a base size will result in fields on subsequent lines being lined up to each other. This is shown in the screenshot below.

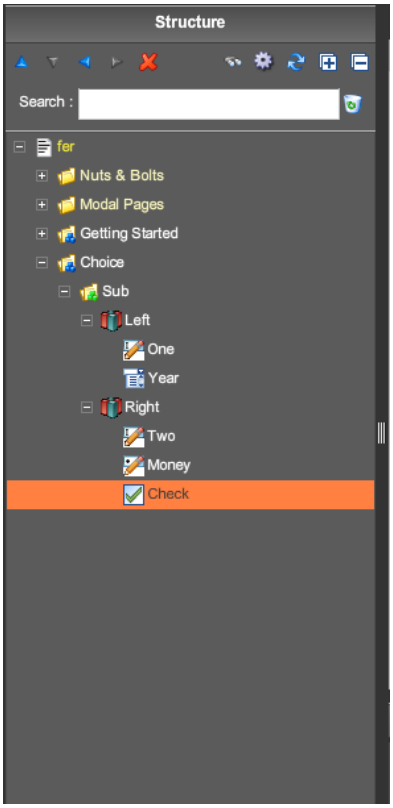
Two Column Layout

You can create a two column layout by using two blocks laid out within a horizontal flow layout, with fields inside them laid out horizontally.

This is shown in the screen-shot below:



Two Column Layout in the Hierarchy



Grid Layout

- Specify columns width relates to section.
- Only useful if you know the format needs to be a grid
- Specified on blocks or sections
- By default, each column in the grid has equal size, but you can also specify the width of each column by defining a percentage of the space available, or a fixed size in mm or inches. For example, in a 3 column grid, you may specify the column widths as "100% 60mm 100%". As in the example, these percentages need not add up to 100.

Note: You can use the Margin property to provide extra spacing between the two columns, or use another smaller block between the two.

Margins (brief)

Margin Inner- Controls the inner margin or padding on the inner div of the field
Margin Outer- Controls the outer margin of field

Expand to Fit Vertical

The Expand to fit vertical option is helpful on PDF forms mostly that do not utilize all responsive layout options. When you have longer labels or multiple fields in a block (div) this will allow wrapping to occur.

Horizontal and Vertical Gap

Horizontal and Vertical Gap increases the Gap between fields. This can be increased in pixels, mm, inches or points.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

Building responsive forms in Composer with the Maguire Template

Unknown macro: 'redirect'

Compatibility

Since	4.0
Deprecated	

Avoka Transact Composer enables form designers to build responsive HTML SmartForms - digital web forms that provide exceptional user experiences across all screen-sizes.

Much of what form designers need comes out-of-the box with the Maguire template, including:

- automatically re-layouts the form as screen size reduces;
- automatically swaps traditional navigation menus and validation handling for space saving techniques once screen size gets too small; and
- touch (large target) fields that are far easier to use with a finger or stylus.

With Composer, there is no need for form designers to learn about the technical details and new design paradigms required to build responsive forms.

This article introduces the few, simple concepts that form designers do need to understand to ensure that the content that they create works within the Maguire responsive framework.

Thresholds

Maguire provides three thresholds at which responsive rules kick in.

For the sake of clarity, this is described in the context of a large screen being reduced in size. Although, this might be exactly what happens in testing, by re-sizing a browser window, it is important to remember that in reality most users will only see a SmartForm in the one fixed screen-size of their device. Two in the case of devices that support screen rotation.

- **Threshold 1** - the largest threshold wraps section level two content under its heading to make full use of the screen width. The two column display above Threshold 1, with section level 2 headings taking up the left-hand column, provides better readability on larger screens
- **Threshold 2** - the threshold at which we are dealing with a small, handheld device, often a smartphone. This is where the greatest changes occur including introducing a fixed header, swapping navigation menus for a slide menu, wrapping fields into a single column layout and expanding their width to take up full screen width
- **Threshold 3** - the smallest threshold wraps the 'continue' and 'go back' navigation buttons so that they appear above each other when there is no longer room to display them side-by-side

The thresholds are defined in form properties as screen widths in pixels:

Structure > Top level form object > Double Click > Form

Should the default threshold values need to be changed, it is recommended that this is performed in a style-sheet and not as form over-rides.

Responsive Rules

Thresholds work by applying the default responsive rules that can be viewed under Structure > Nuts & Bolts > Responsive Rules

The naming convention describes what will occur as screen size reduces below the threshold e.g. objects set to Threshold2-Hide will hide when screen size is below threshold 2.

Typically, the default responsive rules will be sufficient for most SmartForms. It is strongly recommended that the default responsive rules are not modified and the creation of new responsive rules is avoided where possible.

Form designers can configure which rule is applied to a widget or block with the 'Responsive Ruleset [HTML]' property:

Widget/Block > Double Click > Rules > Responsive Ruleset [HTML]

The responsive rule of a block defines how that block behaves only. It does not apply to the children of that block.

Cases where the form designer may want to configure the responsive rule for a block include:

Separators

Additional visual separators may be required below Threshold 2 to maintain field grouping that would otherwise be lost due to their re-positioning as a single column. For instance, rows of fields in a repeating section.

To add separators, create widgets in the form and configure them with the responsive rule 'Threshold2-Show'.

Button Placement

By default, buttons that are located at the end of a row of fields will be re-positioned below the fields below Threshold 2. Even with the use of separators as described above, it may not be immediately obvious to the user, as to which group of fields the button belongs. It is often better usability to position the button at the top, right of the first field in the group.

To keep a button on the top right of its group of fields under Threshold 2, re-structure the fields into two sibling blocks - each block at the same level in the Structure, contained by the same parent. The first (upper) block contains the fields and the second (lower) block contains the button. Configure both blocks with the responsive rule 'None'.

Use relative widths as described in the Field widths chapter below to achieve the desired layout - typically with the button taking up a small amount of space on the right. Each of the blocks described can contain other blocks as long as they use relative widths.

White-space

Filler (invisible) objects used to achieve specific layouts can introduce unwanted white-space under Threshold 2 by either creating an empty row or preventing a field from filling the entire screen width.

To remove this white-space, remove filler objects and achieve the desired layout using the techniques described below in the Field widths chapter. If this is not possible, or too much work, then you can configure the filler objects with the responsive rule 'Threshold2-Hide'.

Screen resolutions

Form designers do not need to consider screen resolutions when setting threshold values, object heights and widths in Composer.

Screen resolution is a common concern for form designers when setting dimensions in pixels. Screen resolution is usually presented as width by height in pixels e.g. 1024 x 768 and sometimes by an equivalent shorthand e.g. XGA.

Particularly, as hardware vendors continue to build small screens with increasingly large resolutions, it is not uncommon to have a 4 inch smartphone in our hand that has an equal or greater resolution than the 24 inch monitor on our desk. Surely, this has to be considered as the physical display size of a 30px x 30px image on a form must be much smaller on the smartphone than on the monitor?

The surprising answer is no. The physical size of the image will be similar regardless of screen resolution.

The reason for this is that a pixel is not a pixel is not a pixel; the pixel used to describe screen resolution is not the same as the pixel used to define sizes in a form. Operating systems and browsers perform calculations, involving pixel densities, to ensure that the logical pixel used in web technologies is independent of the physical pixel of a screen. This has been specifically implemented to make the web developer and SmartForm designer's job significantly easier.

Field widths

Avoid using fixed widths wherever possible.

Fixed width fields are not responsive on screens larger than threshold 2; the width will not be reduced as screen size reduces, potentially resulting in content that does not fit within the available space and runs off screen to the right. This is of particular concern for fixed width fields that are on the same line as other fields as these are more likely to use up all available space.

Note that it can be difficult to visually identify the offending field(s); other relative width fields can appear to be over-sized due to the offending field(s) increasing the size of its containing block.

This is typically not a concern below threshold 2 as the default responsive rules take over control of field width.

Relative width layouts can be created using:

- Horizontal layout fill (Field > Double click > Layout > Layout Fill) ; or
- Grid layout manager (Block > Double click > Layout > Layout Manager); or
- Width (weight) (Field > Double click > Layout > Layout Data) when using the 'Flow - Top to Bottom' layout manager.

By default, relative width fields take up all the available horizontal space and share it equally with other fields positioned on the same row. Custom layouts can be achieved using:

- Width (weight) and grid column widths to make one relative field wider than another on the same row
- Springs and margins to introduce white space around a relative field

An exception to this are image resources which must have a fixed size.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

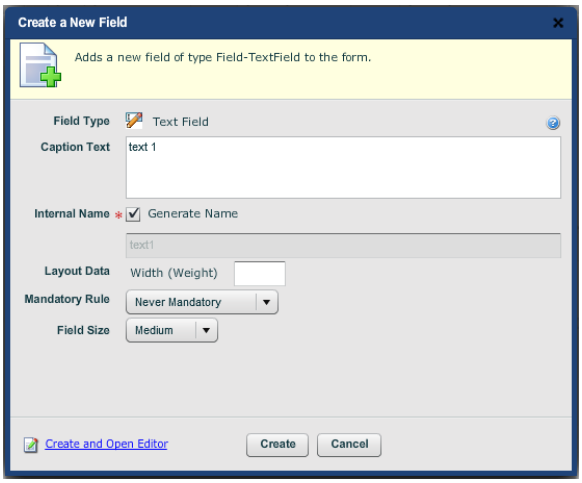
Layout Data options for fields in Composer

Unknown macro: 'redirect'

Compatibility

Since	4.0
Deprecated	

When creating a text field or a dropdown list in Composer, one is able to determine the location of the field in relation to other fields.



Keep on Same Row

The new field will be positioned on the same row as the field immediately preceding it within the container (either section or block) they both reside in.

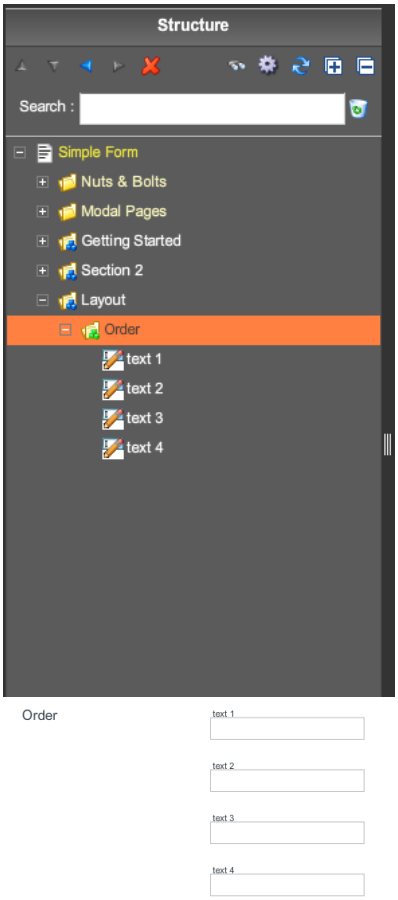
Layout Order

The default position of a field within the container (either section or block) it resides in, is determined by its relative location to other fields within the container.

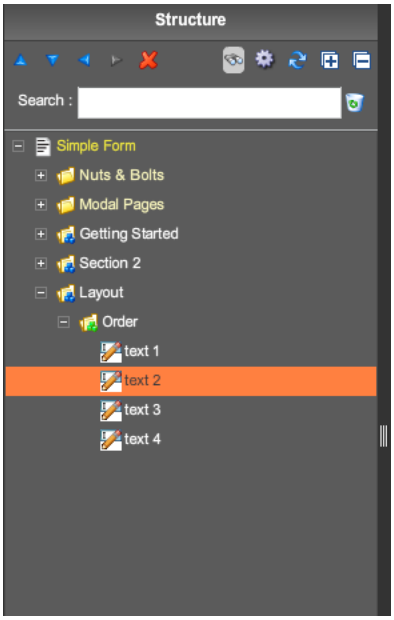
However this can be changed by assigning it a number larger than 0 in the 'Layout Order' property (found on the 'Create New Field' dialog or in the property editor > Properties > Layout). The actual value is not important; the size of the number relative to the number allocated to other fields is important.

Example

There are 4 text fields (text 1, text 2, text 3, text 4) which have been included using default positioning.

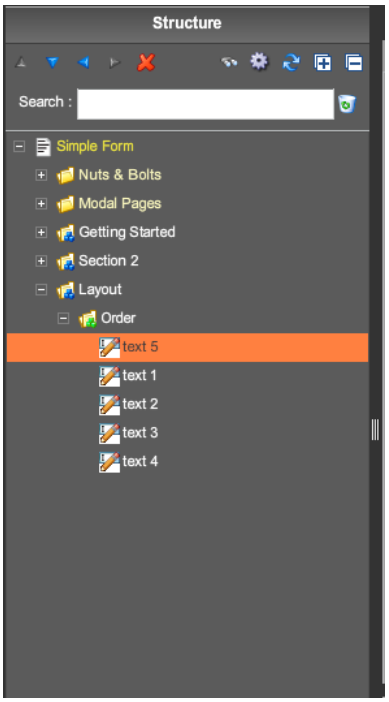


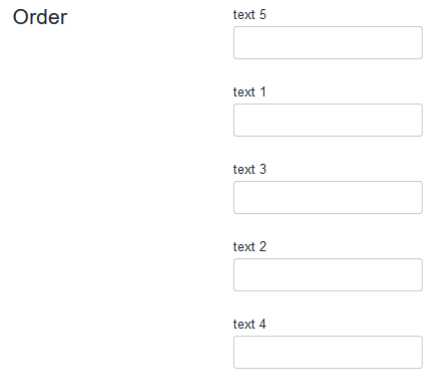
Layout order of 'text 2' changed to 2 (default is blank). Structure pane remains unchanged but text 2 is now moved to the end of the "Text Field's Section".



Exclude from Layout Manager

To illustrate how this property works, I am going to include another text field called 'text 5' with the 'Exclude from Layout Manager' property selected. Note that it is inserted at the beginning of the Section level 2.





If I then change the layout of how the fields within the section are positioned from 'Flow - Top to Bottom' to 'Flow - Left to Right', text 5 is excluded.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

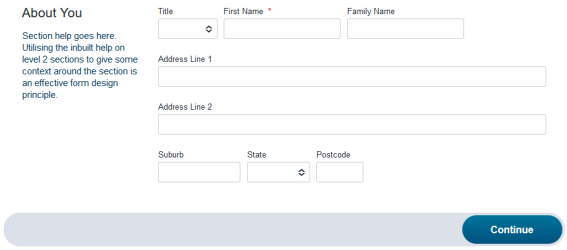
Adjusting the Maguire Section Level 2 header positioning in Composer

Unknown macro: 'redirect'

Compatibility

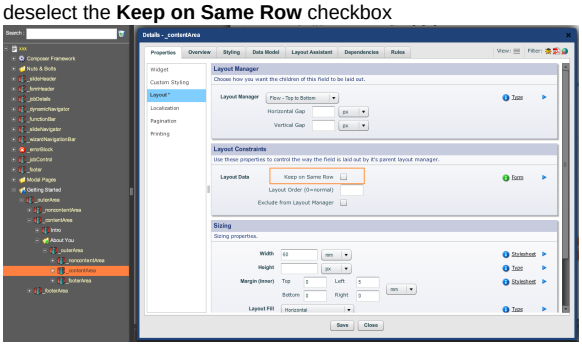
Since	4.0
Deprecated	

In the standard Maguire template, all fields are placed to the right of a Section Level 2.

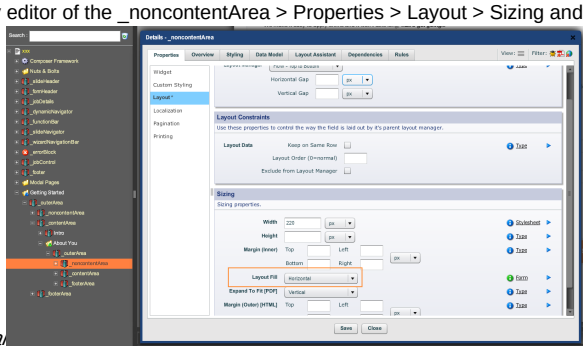


To adjust the header position such that it appears above the content:

1. Show Internal fields in the Structure pane
2. Expand the particular Section Level 2 and the _outerArea block within it
3. Open the property editor of the _contentArea > Properties > Layout > Layout Constraints and deselect the **Keep on Same Row** checkbox

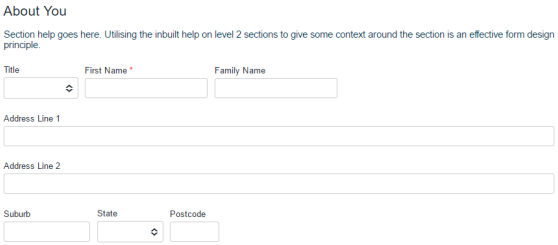


4. Open the property editor of the _noncontentArea > Properties > Layout > Sizing and select **Layo**



ut Fill = *Horizontal*

The Result



Related articles

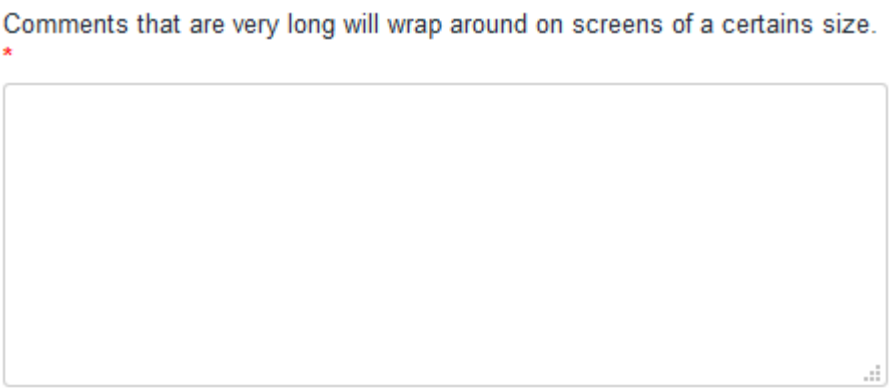
- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

How to stop the mandatory marker from wrapping

Unknown macro: 'redirect'

Generally this is not an issue, as the vast majority of field captions are short enough that the text never needs to wrap. On some occasions however, the text for a field's caption may be longer than the page is wide, especially on mobile devices with limited screen space.

Since the default mandatory marker is separated from the caption using a standard space character this can result in situations like the following:



Where, on very specific screen widths, the marker wraps onto its own line.

This can be avoided in HTML forms by replacing the standard space character with a 'non-breaking space' character. Those familiar with HTML will be used to using this type of space character by inserting the following reference into their HTML pages:

** **

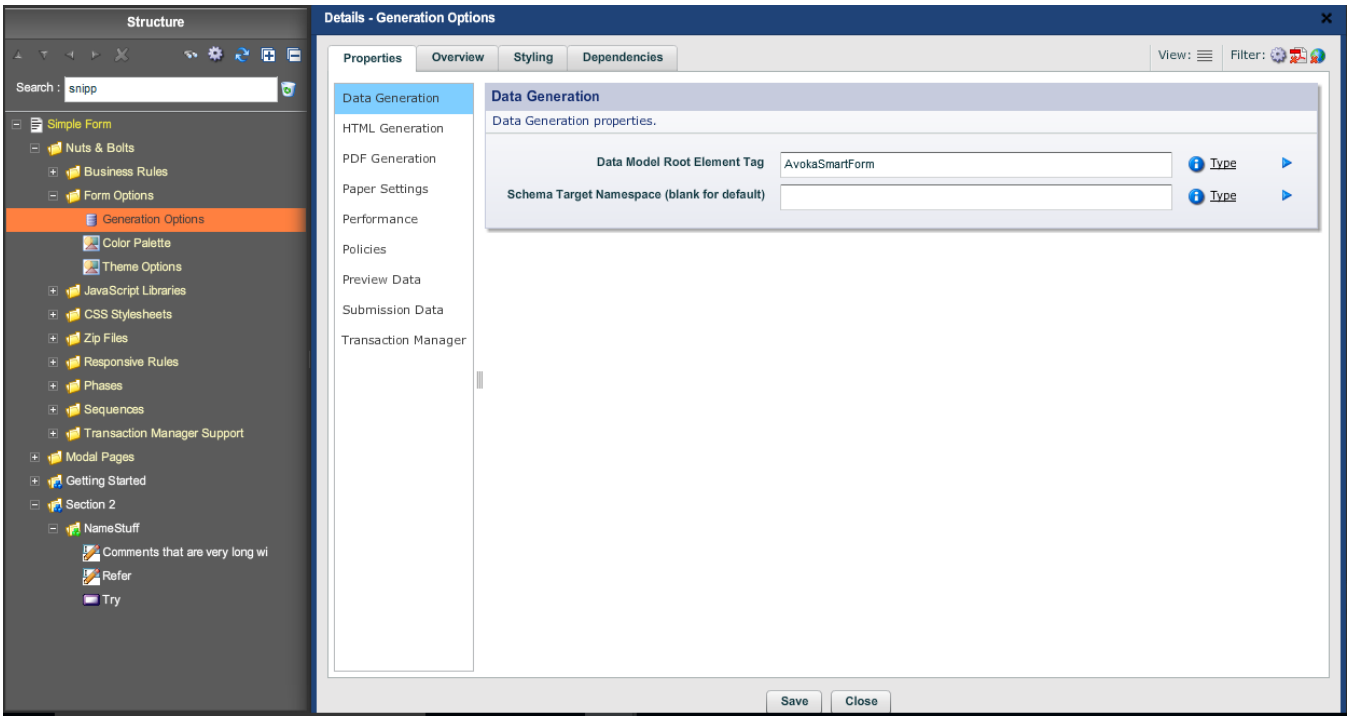
This is a special entity that is replaced with the real space character when the page is viewed in a web browser.

Since Composer forms are stored as XML we can't use the HTML non-breaking space, instead we need to use the XML version, which is:

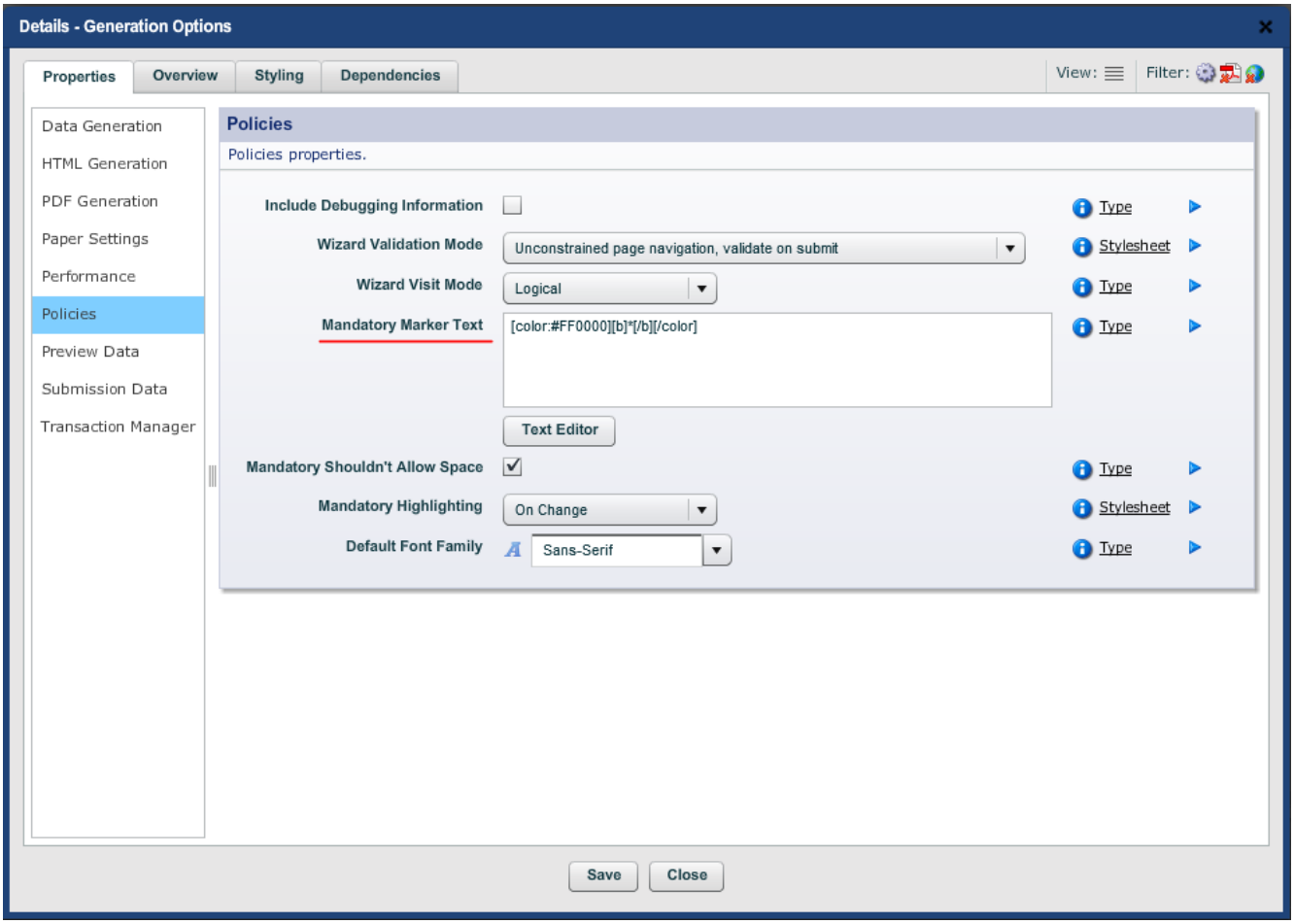
** **

This entity will be replaced with the HTML version when the form is rendered as HTML.

The text used as a mandatory marker can typically be modified in the form's **Generation Options** which can be found in Composer by navigating to **Form > Nuts & Bolts > Form Options > Generation Options**:



And then changing the property **Properties > Policies > Mandatory Marker Text**:

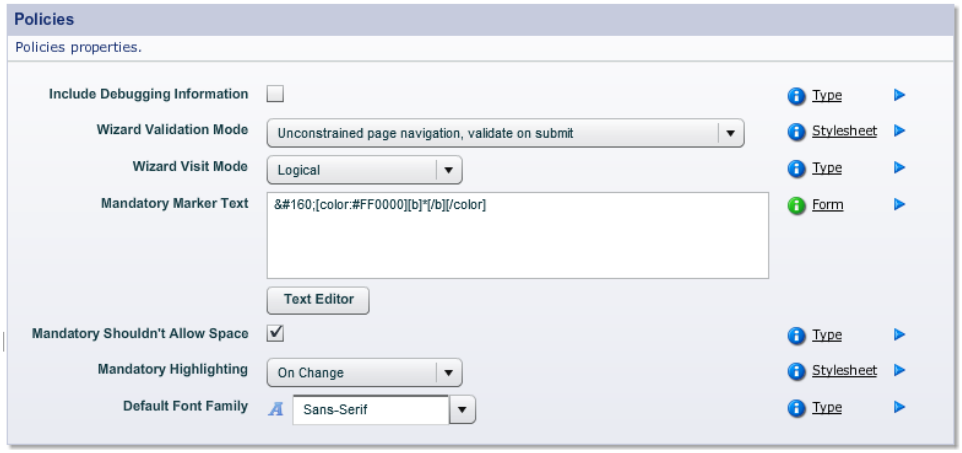


For this particular change however, there is one extra trick.

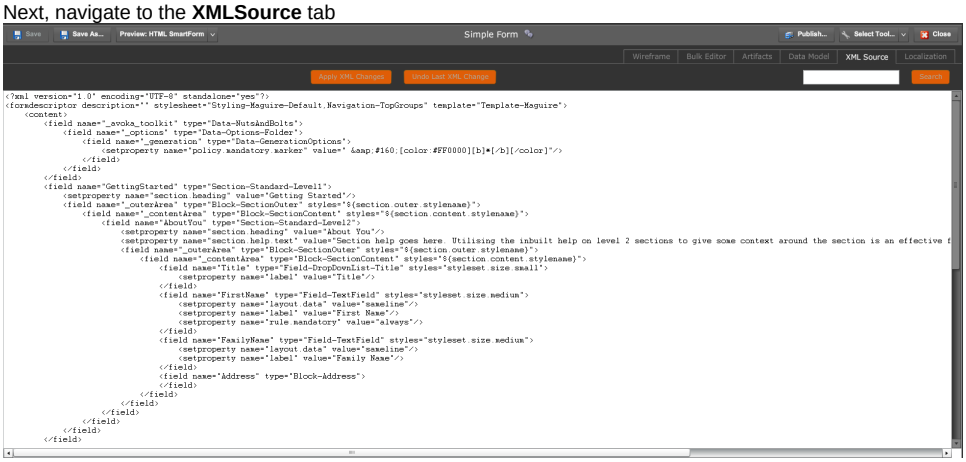
The '**&**' character is a reserved character in both HTML and XML, meaning the text editor will 'escape' it. So if we enter ** ** directly in the text editor we end up with ** ** in the XML, where **&** represents the '**&**' character.

As a result we need to do this in a few steps:

1. Delete the existing **Mandatory Marker Text**
2. Add the new text: ** [color:#FF0000][b]*[/b][/color]**



3. Save the properties
4. Next, navigate to the **XMLSource** tab



5. Find the line: <setproperty name="policy.mandatory.marker" value="&**amp;**#160;[color:#FF0000][b]*[/b][/color]"/>
6. Remove the added '**amp;**', so the line becomes: <setproperty name="policy.mandatory.marker" value="** **[color:#FF0000][b]*[/b][/color]"/>
7. Click **Apply XML Changes**

At this point you're done and the mandatory marker should now always stick with the last word of the caption. However it's worth noting that once you save the XML changes the line you edited will change to:

<setproperty name="policy.mandatory.marker" value=" [color:#FF0000][b]*[/b][/color]"/>

Note the missing ** **, this is to be expected as the XML editor is now displaying a real non-breaking space in place of the special entity.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)

- Addresses

Abandonment Support V4

Unknown macro: 'redirect'

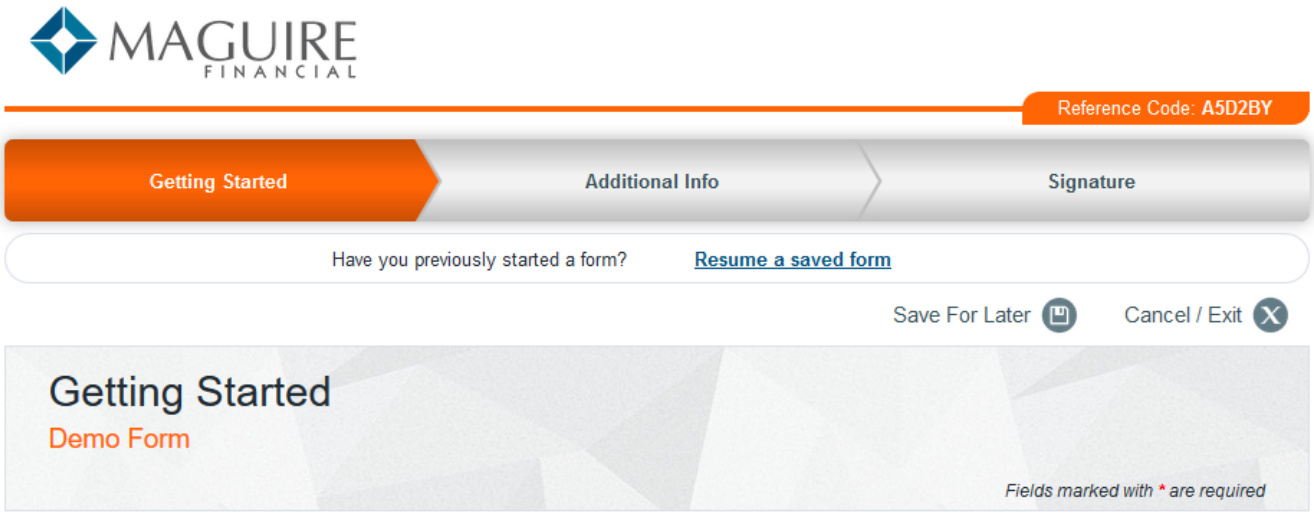
The most favorable outcome for a form, is where the customer completes and submits it. There are occasions where a customer may abandon the form because of difficulties completing the form, or due to technical issues.

Transaction Manager provides a list of abandoned transactions. These include transactions that have been abandoned by users i.e. where the user explicitly clicks a button to abandon the submission, or implicitly by closing the browser or simply never returning to a saved form. There may also be submissions marked as abandoned by administrators.

Enabling Abandonment Support in Composer

Cancel button

On the new form wizard, select the option "Include Cancel Feature". This will insert a "Cancel" button in the header.

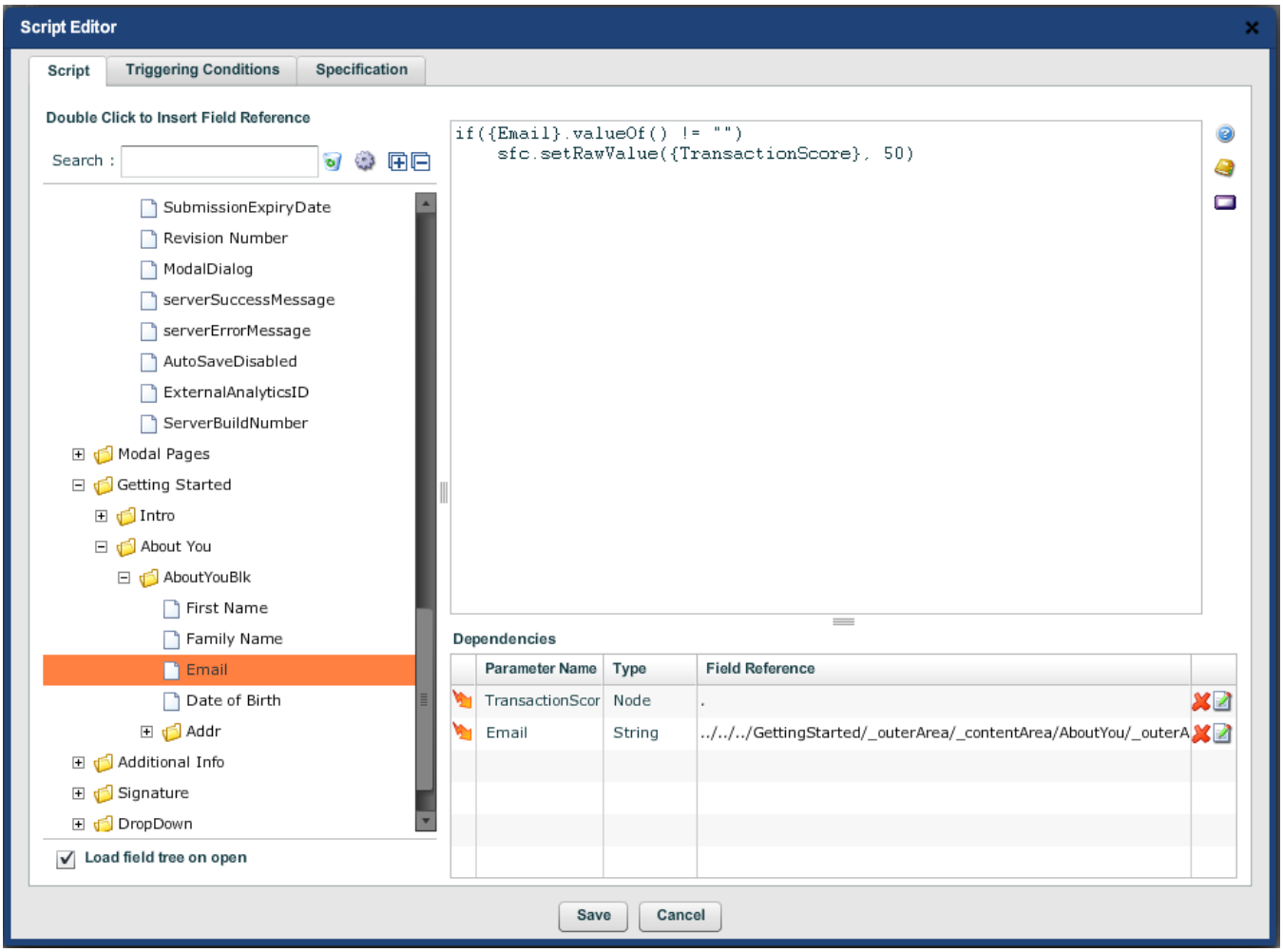


When the customer selects the Cancel button to abandon a form, this will be recorded as an explicitly "Abandoned transaction" (Abandon Type = "User") in Transaction Manager.

Transaction Score

This value can be set using a calculation in the form. For example, the calculation may score a minimum value if the user has entered their name and contact information, and then higher scores based on their net worth and demographics. Depending on your abandonment rules set in Transaction Manager, this will be used to determine whether a particular partially completed form should be followed up manually or discarded.

The calculation script is entered in the form's "Nuts & Bolts > Transaction Manager Support > TransactionScore > Rules > Calculation > Script Editor" (an example can be seen in the screenshot below):



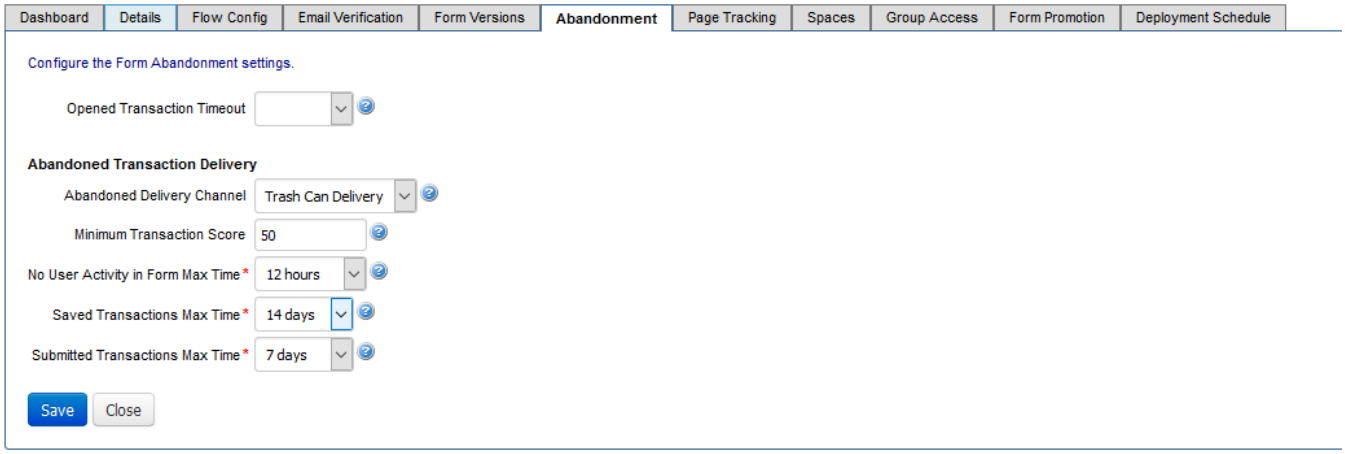
The values will be calculated against the appropriate fields as the form is being completed and is accessible from the Transaction Manager console "Operations > Abandoned Transactions" screen when the form is abandoned.

ID	Tracking Code	Form	Org.	Last User Activity	Anon. User	User / Contact Email	% Complete	Transaction Score	Abandon Type	Abandon Reason	Action
266	5DVX4L	Demo Form	maguire	20 Nov 10:37	✓			50	User		

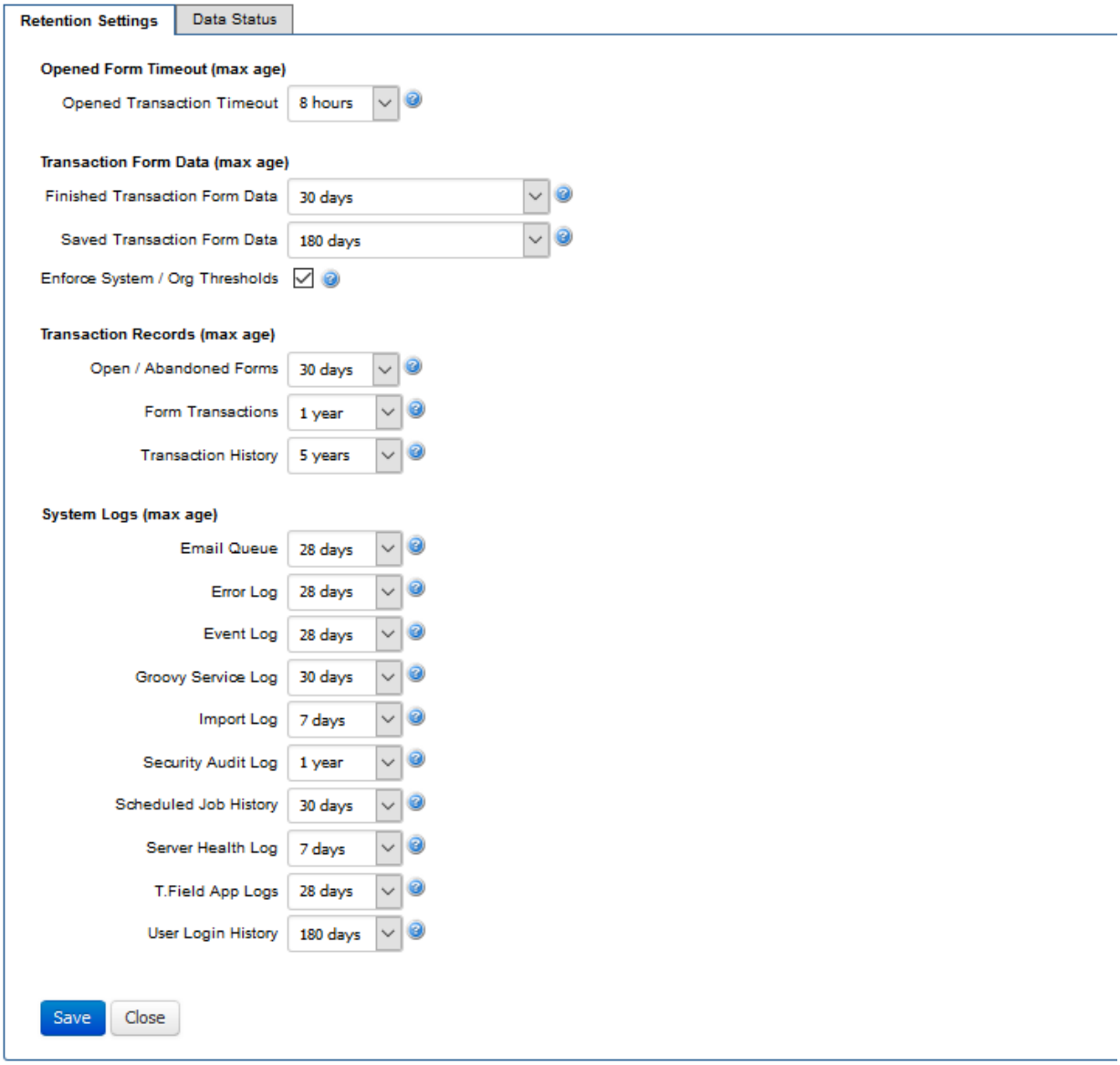
Depending on your security privileges, you may also be able to save an abandoned transaction using the button. This transaction will now appear in Operations > Saved Transactions screen.

Enabling Abandonment Support in Transaction Manager

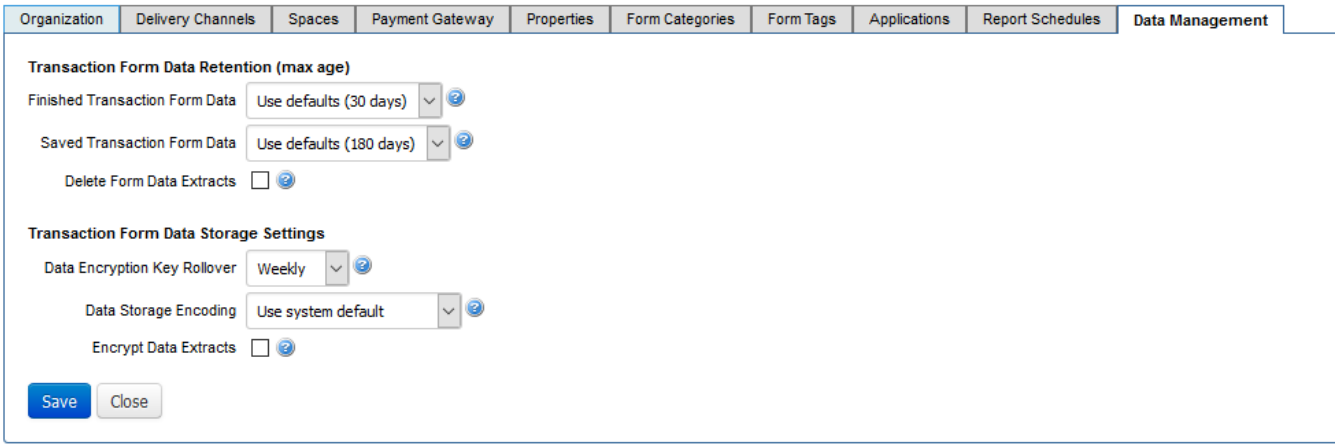
To customize the abandonment process for an individual form, use the form's dashboard > Abandonment tab.



Otherwise abandoned transactions will follow the policy set on the "System > Data Retention Management" screen.



This maybe customized per organization as well.



Related articles

- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Including Web Fonts in Composer forms
- Images in Composer forms
- How to style bullet points in a Composer form

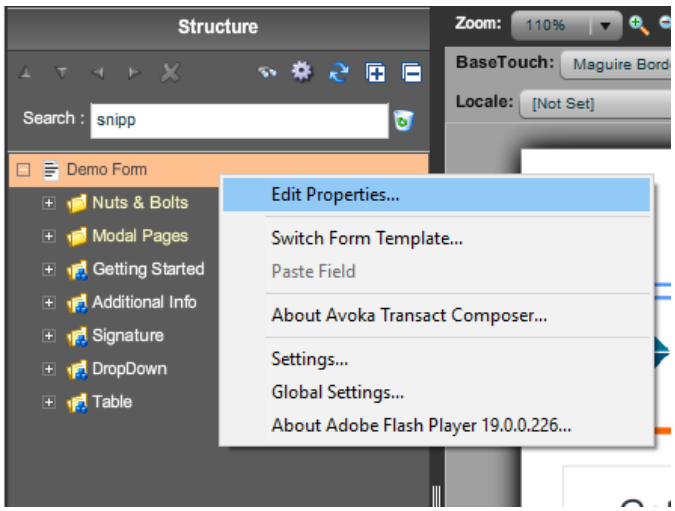
Form Object using the Maguire template V4

Unknown macro: 'redirect'

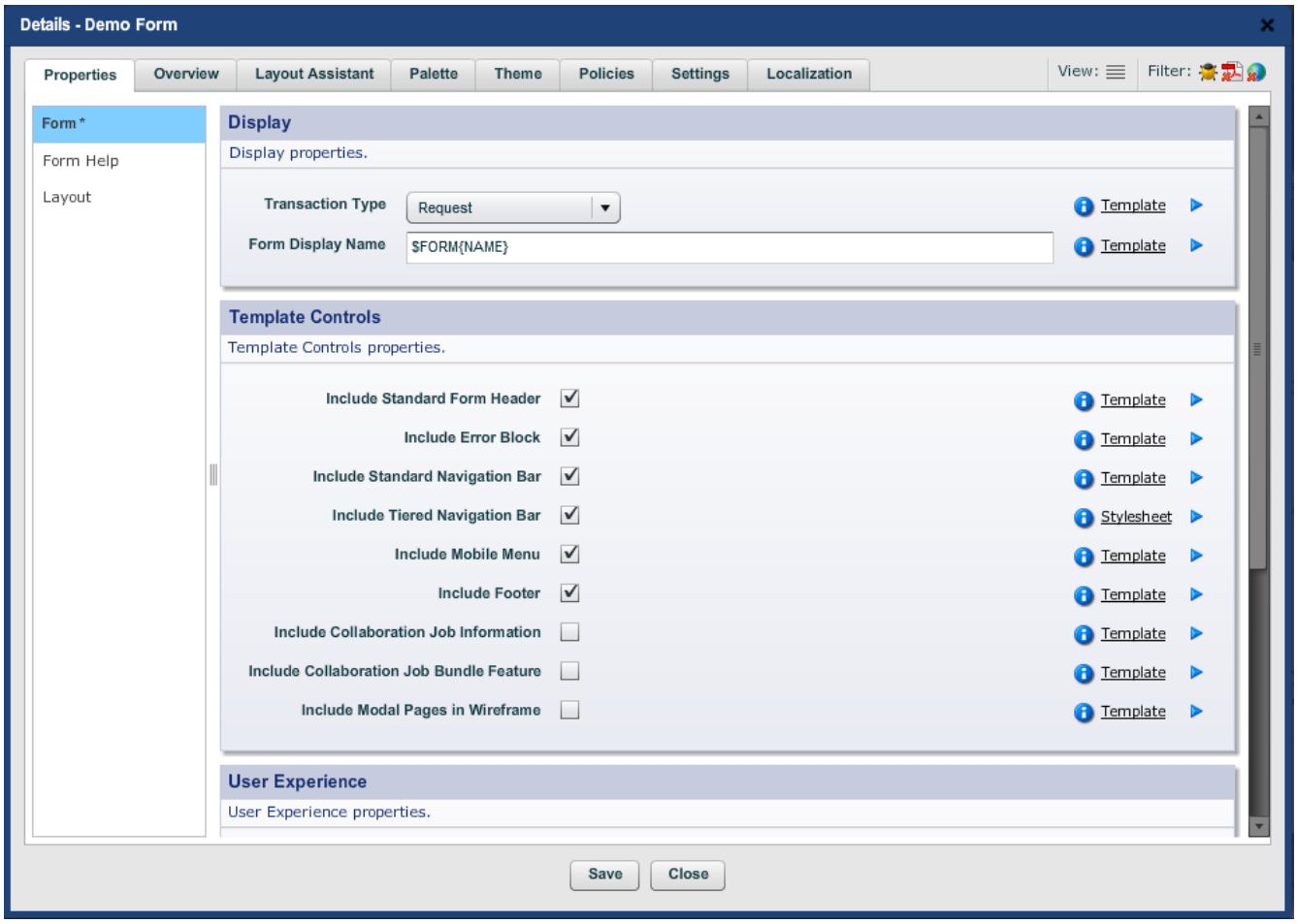
The 'Form Object' is found at the top of the Structure pane in the Composer UI.

Some properties maybe changed when creating a form in the New form wizard.

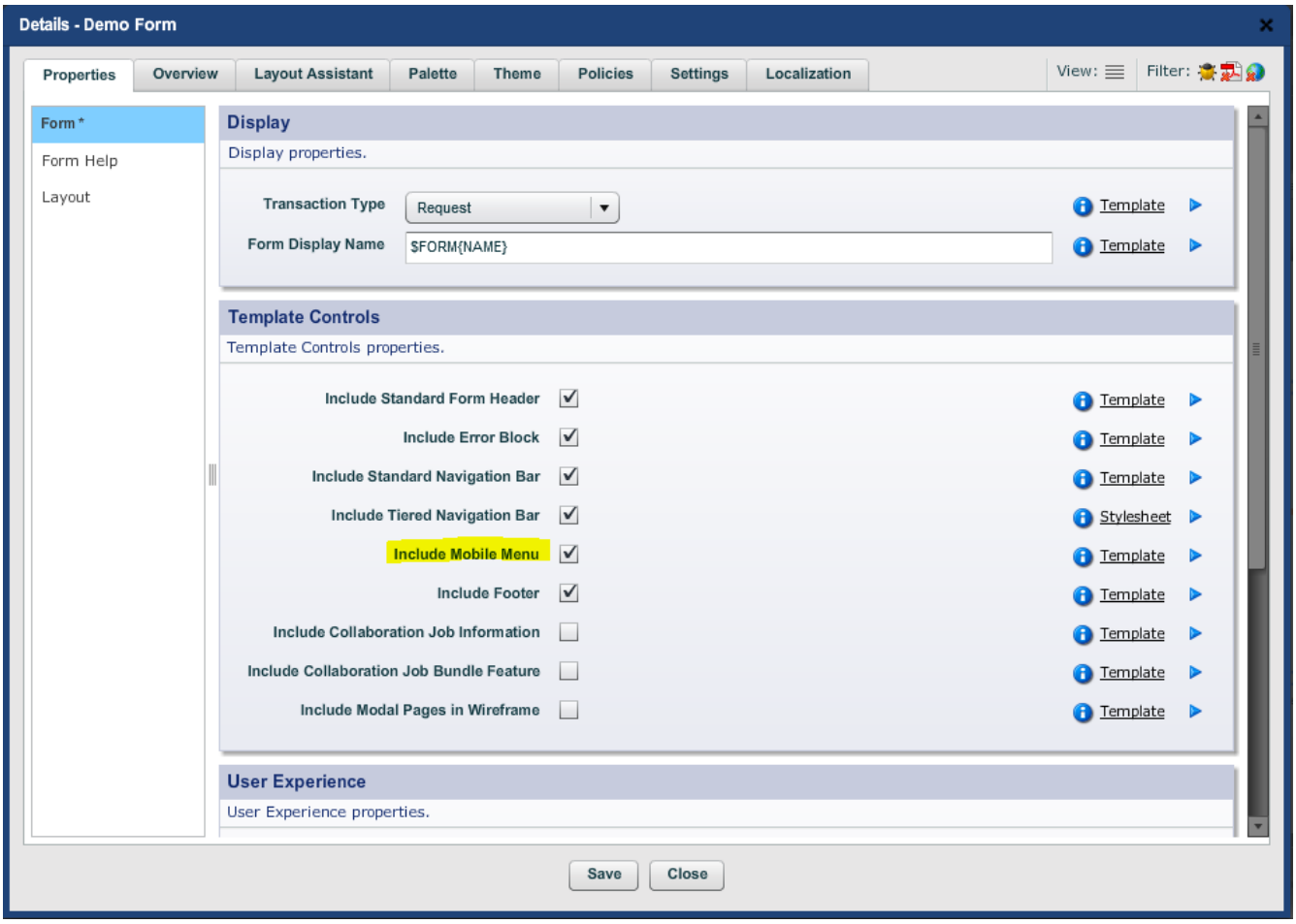
Here are a list of the properties found in the property editor > Properties tab > Form option and below is a Maguire form with various parts highlighted respectively:



Maguire form annotated.

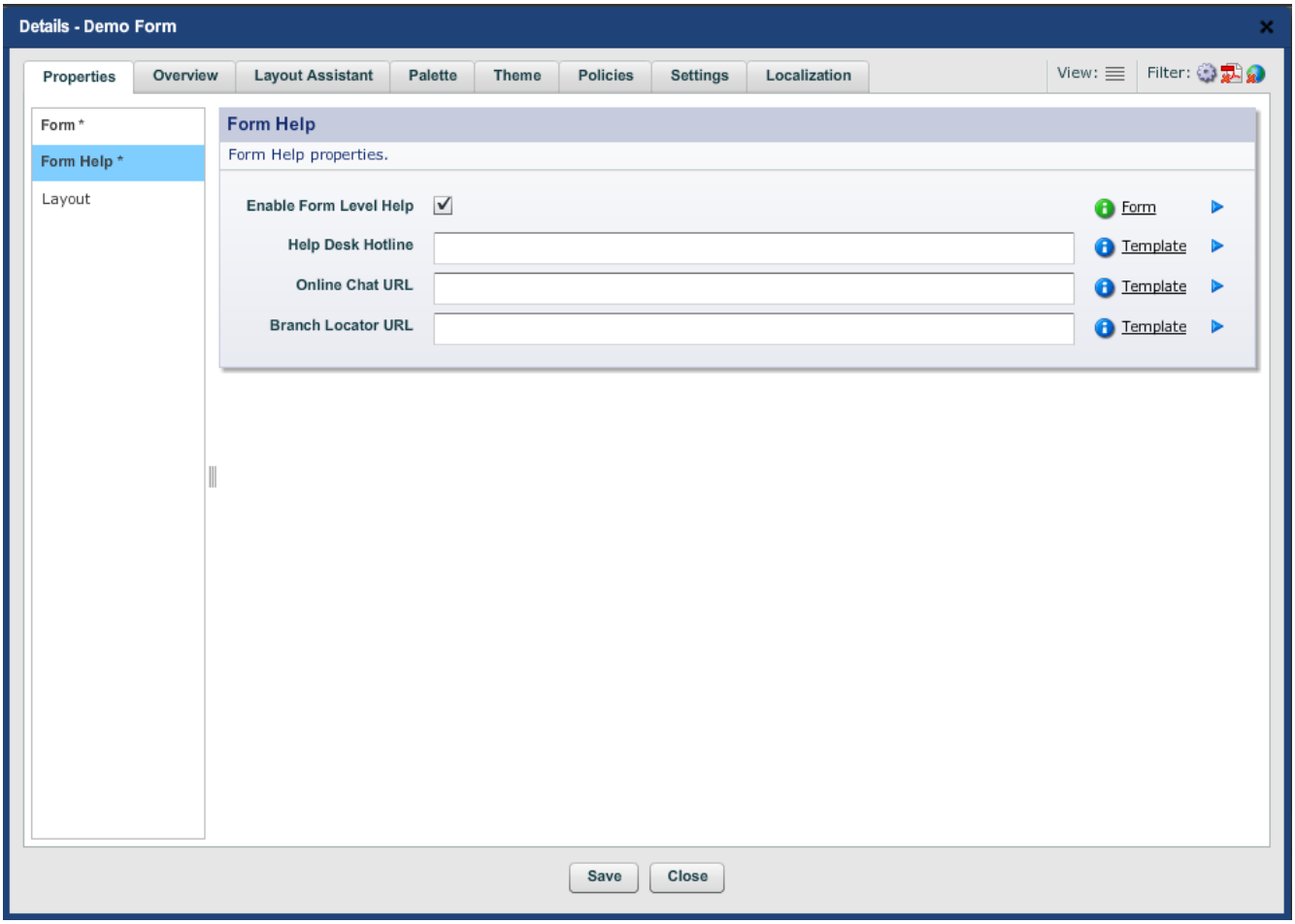


Mobile version of the Maguire form to show 'Mobile Menu' option:



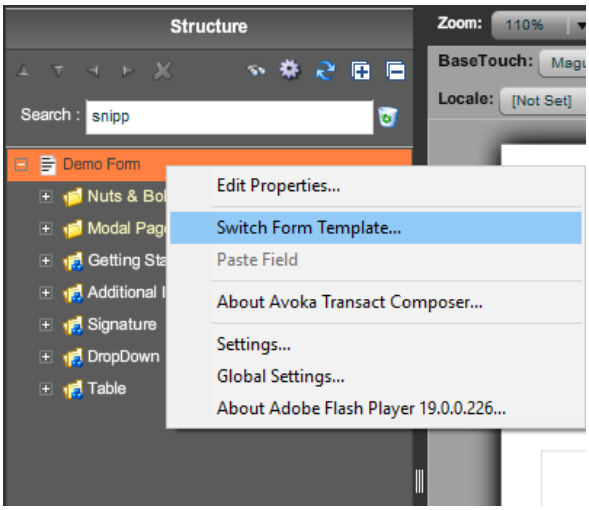
Update form help

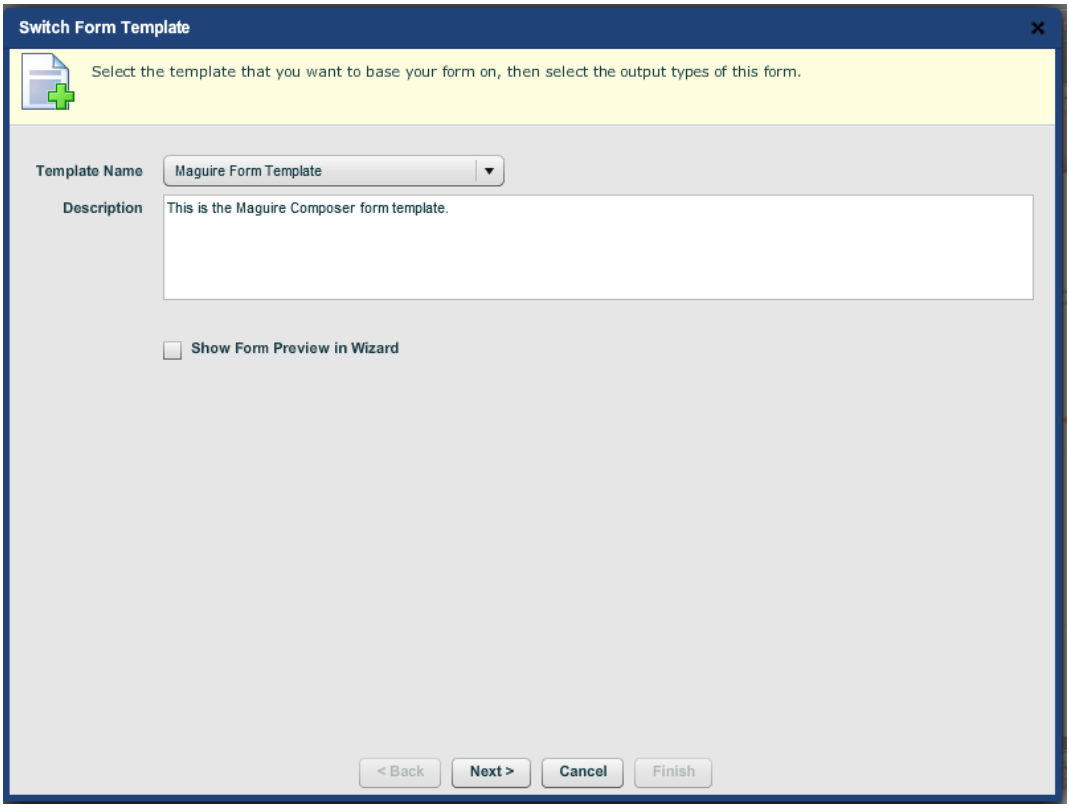
To update form help, Click the Form Help menu option in form properties.



Switch form templates

Right-click on the form object and select option 'Switch Form Template...'.





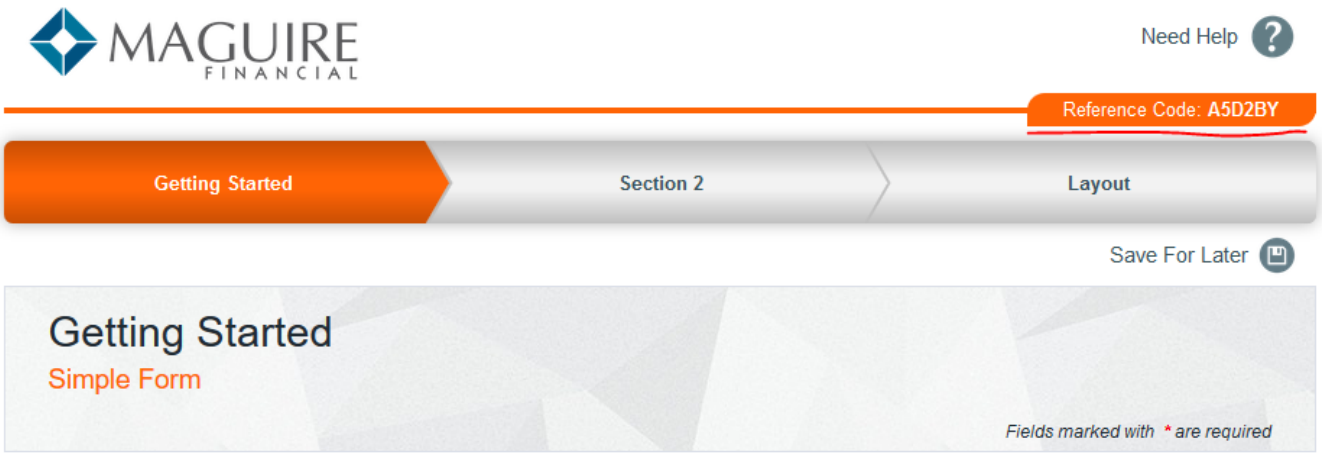
Related articles

- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Including Web Fonts in Composer forms
- Images in Composer forms
- How to style bullet points in a Composer form

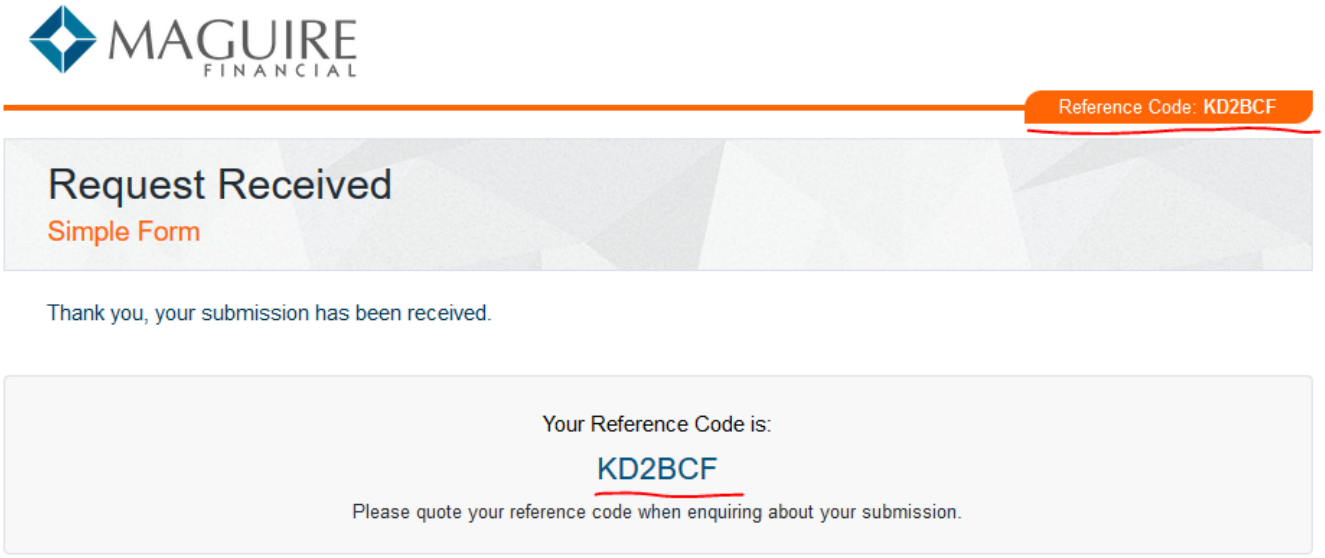
Tracking Code V4

Unknown macro: 'redirect'

The 'Tracking code' was introduced in version 4. It is a 6 digit alphanumeric code applied to an individual submission and is displayed on the form as soon as its rendered.



It may also be located on the confirmation page.



Purpose

The purpose of the tracking code is to enable cross channel support.

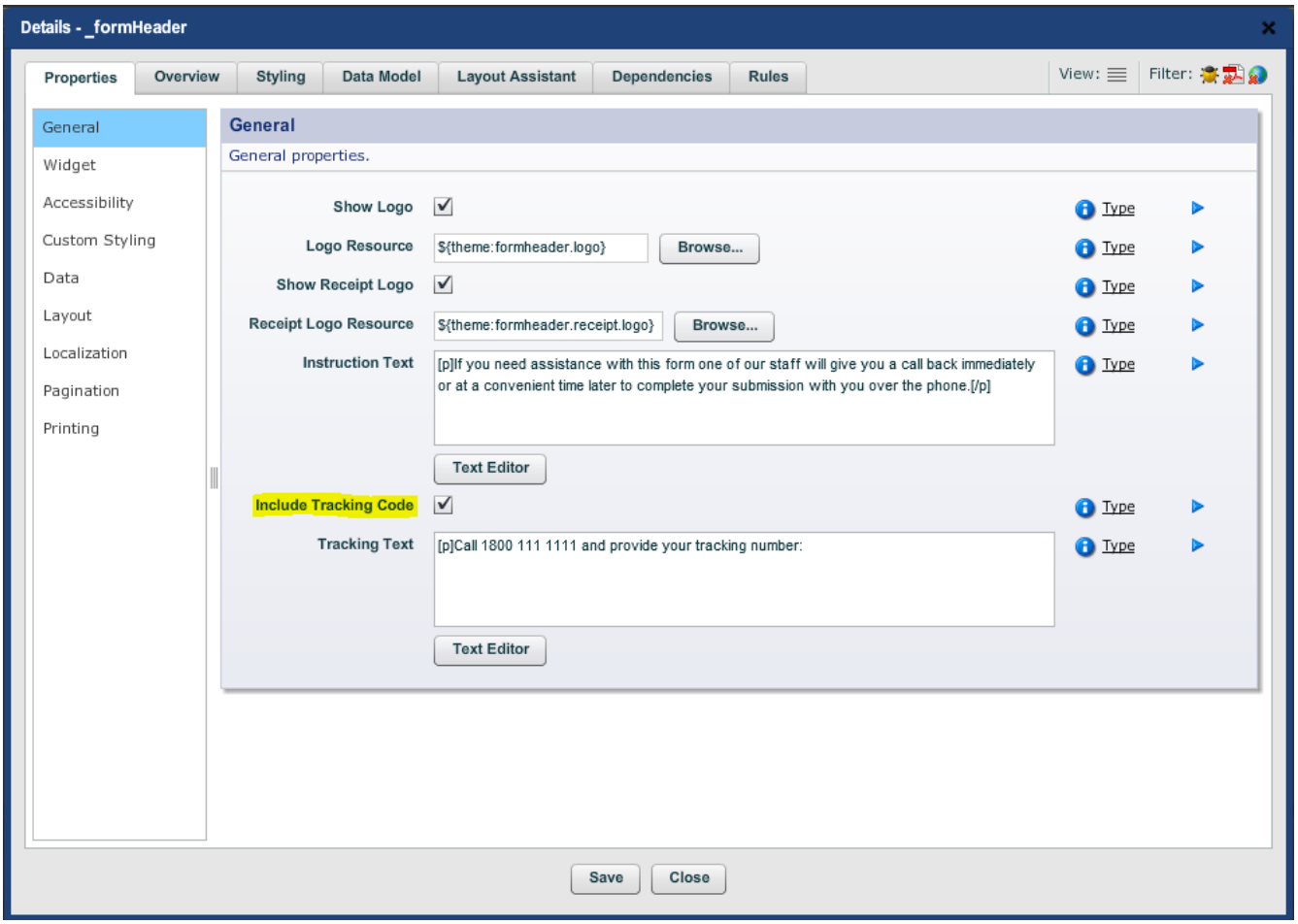
Example 1: The form filler partially completes a form using his/her phone, then saves it. When he/she gets to work they complete it on their desktop. To find the draft, they use the tracking number.

Example 2: It may be used by help desk operators to identify a submission.

Example 3: It may be used in anonymous form submission to enable multiple parties to complete a form (e.g. husband and wife).

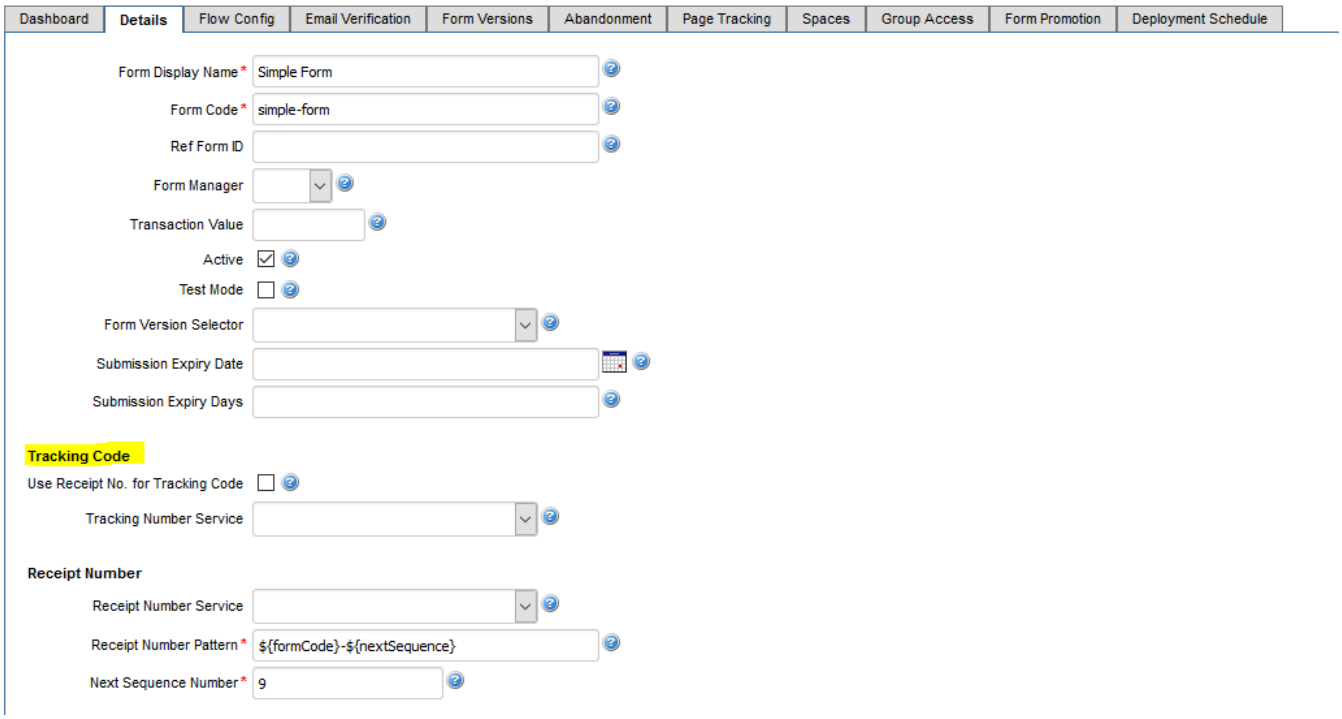
Composer form

The reason you see the tracking code is because there is a internal property called 'Include Tracking Code' on the _formHeader that is selected by default.



Transaction Manager

To influence the generation of the tracking code, open the 'Details' tab on the form dashboard.



You may use the Receipt No. as the tracking code (the receipt number has been around since the first version of Transaction Manager / Smart Form Manager / Form Center. It is assigned after a form is submitted and before the receipt is printed. It appears on the receipt) or you may create/use a 'Tracking Number Service to populate the tracking code.

Related articles

- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Including Web Fonts in Composer forms
- Images in Composer forms
- How to style bullet points in a Composer form

Turning "Go to" questions into "hide-show" logic

Unknown macro: 'redirect'

When organizations design paper forms, they often provide guidance to the user by adding "Go to" directions after specific questions. This type of logic is used for the same reasons as hide-show logic, to reduce the amount of information the the user has to complete.

An example page from a government form is shown below:

Composer rules, however, work in reverse. Instead of "go to" instructions, a rule is placed on the target question which specifies under what circumstances it should be visible. Composer rules point backwards to previous questions, whereas "go to" instructions point forwards telling the user where to go next. This is shown in the above sceenshot for Q97 using red "dependency" arrows.

How can you convert from "go to" instructions to Composer rules?

Tip #1: Build your form, creating all the questions. Remove all the instructional text. (You can of course do this a chunk at a time.) Remove the question numbers - this is because if you're using hide/show, some questions will be hidden and therefore their numbers will be skipped, and your users may become confused or anxious.

Let's use the above example, and work out the business rules one question at a time.

Q93. It's relatively obvious that Q93 should only be visible if the answer to Q92 is "No". This can be very simply implemented as a Composer visibility rule: "Q93 is visible if Q92:"No" is selected.

Remember that when you add a visibility rule, it means that the object will by default be invisible. It will only be visible if the rule is true. This means that all optional objects will be initially invisible, until the user starts answering questions.

Q94. Again, this is relatively simple - it should be visible if the answer to Q92:"Yes" is selected.

Tip #2: For each question, look backwards through the form and locate any previous questions that point to this question. For each "Go to" that you find, add that answer to the visibility rule. Sometimes this is slightly tricky, because the question number isn't explicitly stated, it may just be referred to as "the next question".

In Composer, the easiest way to put a rule on a radio button is to use the Script rule editor, and simply locate and double click on the specific radio button. This will insert the radio button as an "isSelected" or boolean value in your script as something like this:

Q95. You might think that this question is always visible. But in fact, it's a "fall through" rule - in other words, Q95 follows on from Q94, and should have the same logic. Q96 is similar. Either a) duplicate Q94's rule in Q95/Q96, or b) wrap Q94, Q95 and Q96 in a block, and put the rule on the block.

Tip #3. For "fall though" questions, either a) wrap them in a block with the previous question, or b) add the same rule as the previous question.

Tip #4. If you do find any questions are parts of the form that must always be answered - there is nothing to do. Relax.

Q97. This is a more complicated rule, but the process is similar - look through all previous questions and find all references to Q97. In this case, Q88 and Q93 both refer to Q97. The rule would look something like: "Q97 is visible if: Q88:"Yes" is selected or Q93:"Children Pay Rent" is selected or Q93:"Other" is selected".

Tip #5. For more complicated rules, simply use "or" (|| in JavaScript) to combine all the options together.

Now let's consider a slightly complicated scenario. You don't necessarily need to follow this in order to build these rules successfully, you just need to implement the tip below - but it's useful to understand why.

Let's assume we've implemented the scenario where a user had answered as follows:

- Q92: No. (Q93 would be displayed.)
- Q93: Other (Q97 would be displayed.)

Now let's assume that the user goes back and changes the answer to some previous question that hides Q92. Using a different rule that we haven't discussed in this article, Q92 is now hidden. But because the user answered "No" to this question when it was visible, Q93 will still remain visible. This can cause serious confusion and errors in logic in the form.

Fortunately, there is a very simple way to solve this problem, and similar problems like it.

Tip #6. Whenever you define a visibility rule, turn on Advanced Properties (using the small gear icon in the Property Editor), and select "When Hiding - Clear Data (Includes Grayed Out)". This will ensure that whenever the rules causes this question to hide, its data will be cleared, which will in turn hide all other dependent questions.

Tip #7. Sometimes form owners insist that the question numbers are retained. Users can become confused when this is done, because the questions will no longer be sequential. One way to overcome this is to use an Editability rule rather than a Visibility rule. Remember to enable the "On read-only, clear data" option similarly to the visibility rule.

If you follow these simple tips, you should be able to quite easily convert paper forms with "go to" instructions into more intuitive Composer forms.

Note: An astute observer might notice that Q89 and Q91 both direct users to skip questions 92-98 under all circumstances, resulting in these questions never needing to be answered. There are no instructions further up in the form that affect this. This appears to be a logic error in the paper form.

Related articles

- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Including Web Fonts in Composer forms
- Images in Composer forms
- How to style bullet points in a Composer form

Payments

Unknown macro: 'redirect'

Businesses often require payment interface functionality within a form. The steps below detail the procedure to follow in order to configure payment interface functionality within your smart form. Note that Transact does not store, process or transmit payment card data; any payment card data interaction is directly between the end user's browser and the payment gateway or tokenisation provider. Avoka recommends that when enabling Transact's payment interface features, you ensure appropriate security, access and change control measures are in place to prevent accidental or malicious change of configurations.

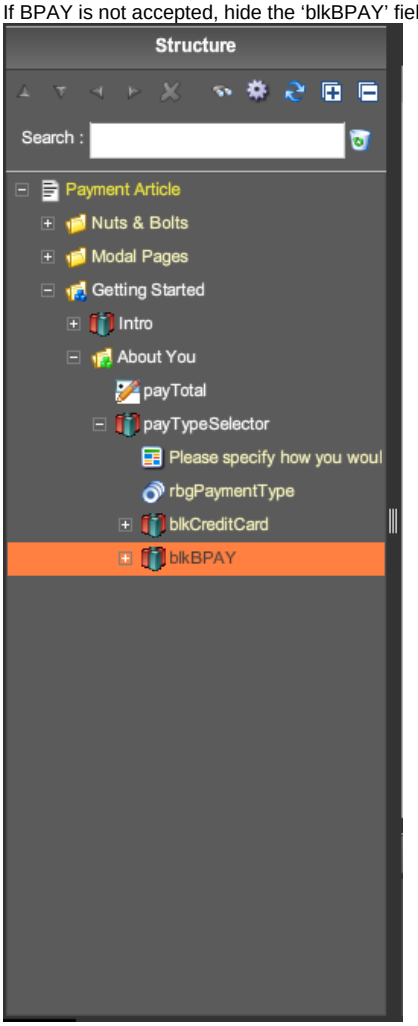
Configuring Payments in Forms

Unknown macro: 'redirect'

Businesses often require payment functionality within a form. The steps below detail the procedure to follow in order to configure payment capability within your smart form.

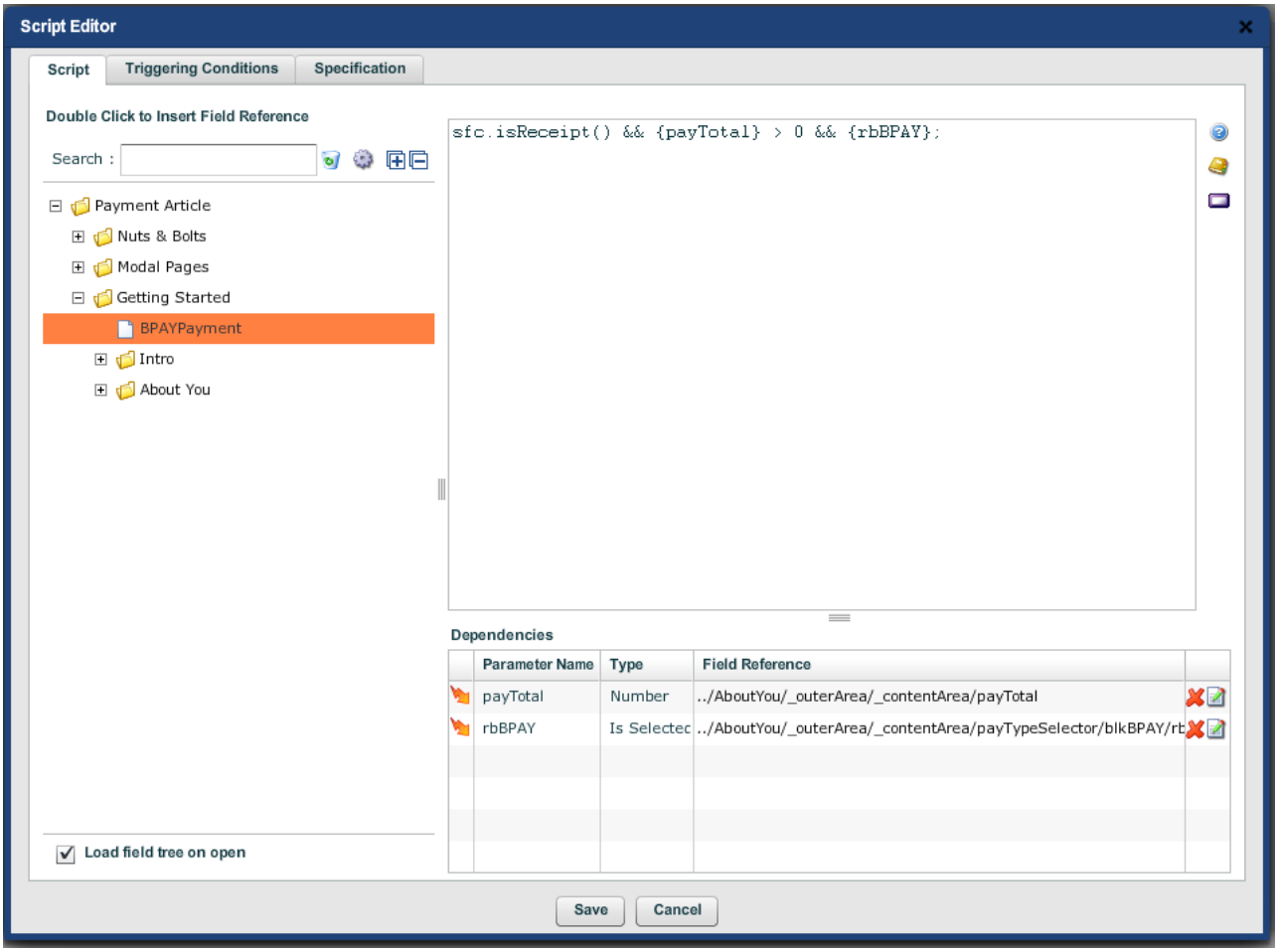
Composer

1. Include both a 'TM Payment Total' widget and 'TM Payment Type selector' in the form.
2. The 'TM Payment Total' field indicates to the form filler what payment is required and is usually a calculated value i.e. not reliant on the user to enter payment information. To create a calculation script, go to the property editor > Rules > Calculation property.
3. The 'TM Payment Type selector' field enables the form filler to select either a credit card payment or a BPAY payment.
4. If BPAY is not accepted, hide the 'blkBPAY' field (see the sample form).



5. If BPAY is accepted the receipt must be configured to display a BPAY biller code and reference number details as well as the amount due. This can be accomplished as follows:
    - Add a block at the beginning of the form (see section level 2 called 'BPAY Payment' in the sample form) and set it to display only on the receipt using the following code in the visibility rule:

        - *sfc.isReceipt() && {*TM Payment Total field *} > 0 && {*BPAY radio button*};*



Fields referenced in the visibility rule above

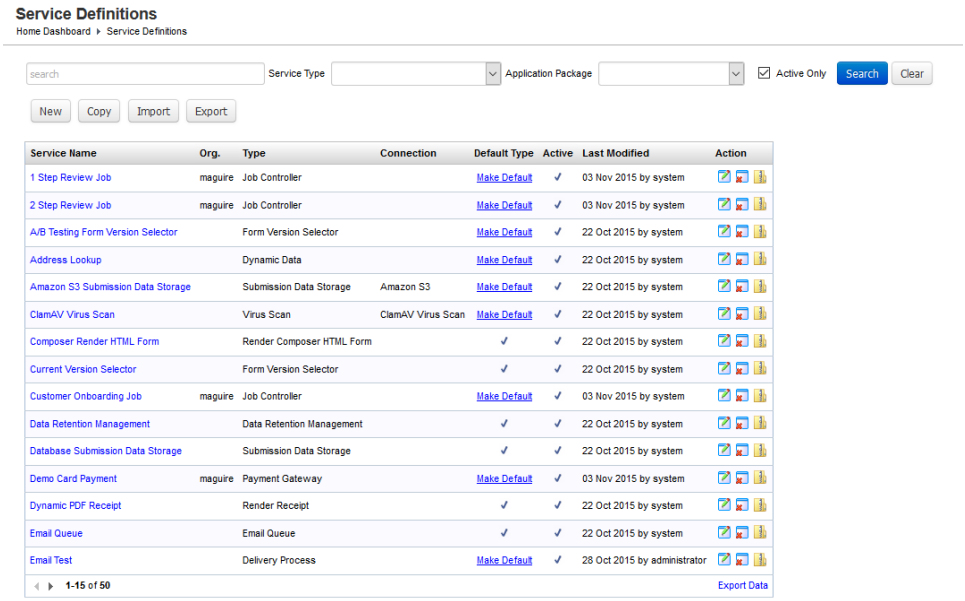
- This code ensures that the block will only display when the form is in receipt mode AND when there is a payment required AND when the user has indicated that they will make the payment using BPAY.

- The 'biller code' and 'reference number' are set using form properties in Transaction Manager. This means that the same form can be used by different organizations, without all of the payments made by users being paid to the same place.

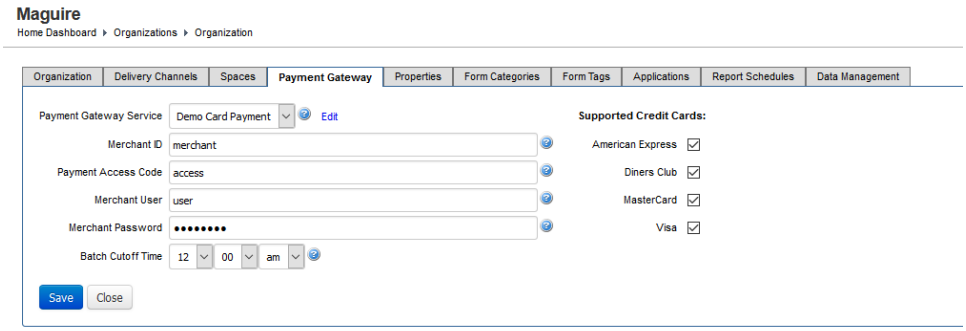
Transaction Manager

For payments to work, a payment gateway has to be setup in Transaction Manager.

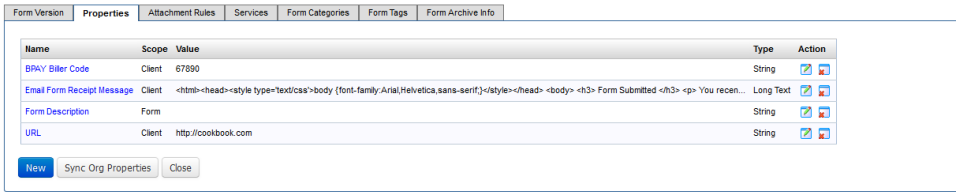
1. A new definition can be is created in the System > Services Definitions page if an appropriate gateway does not exist (see Transaction Manager documentation for more information).



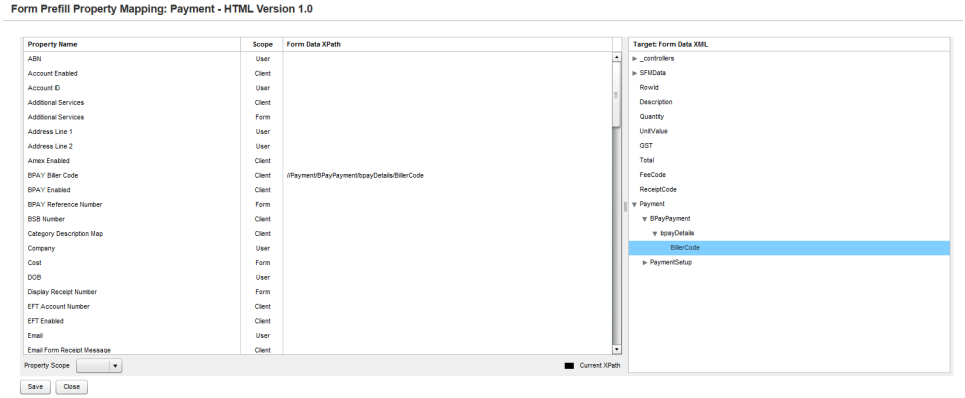
2. Then associate the payment gateway service to the relevant organization using Forms > Organizations > Payment Gateway tab.



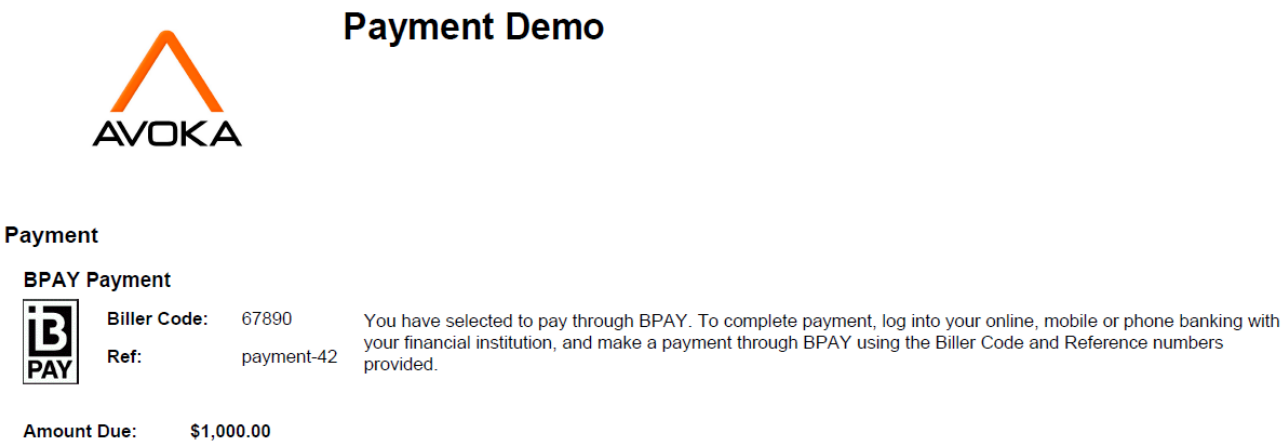
3. Complete the details required including which credit cards may be used.
4. In the Properties tab, create a 'Biller Code' property if it doesn't exist.
5. Open the form and include the property at form version level.



6. Map the properties to the biller code and biller reference number fields in the form.

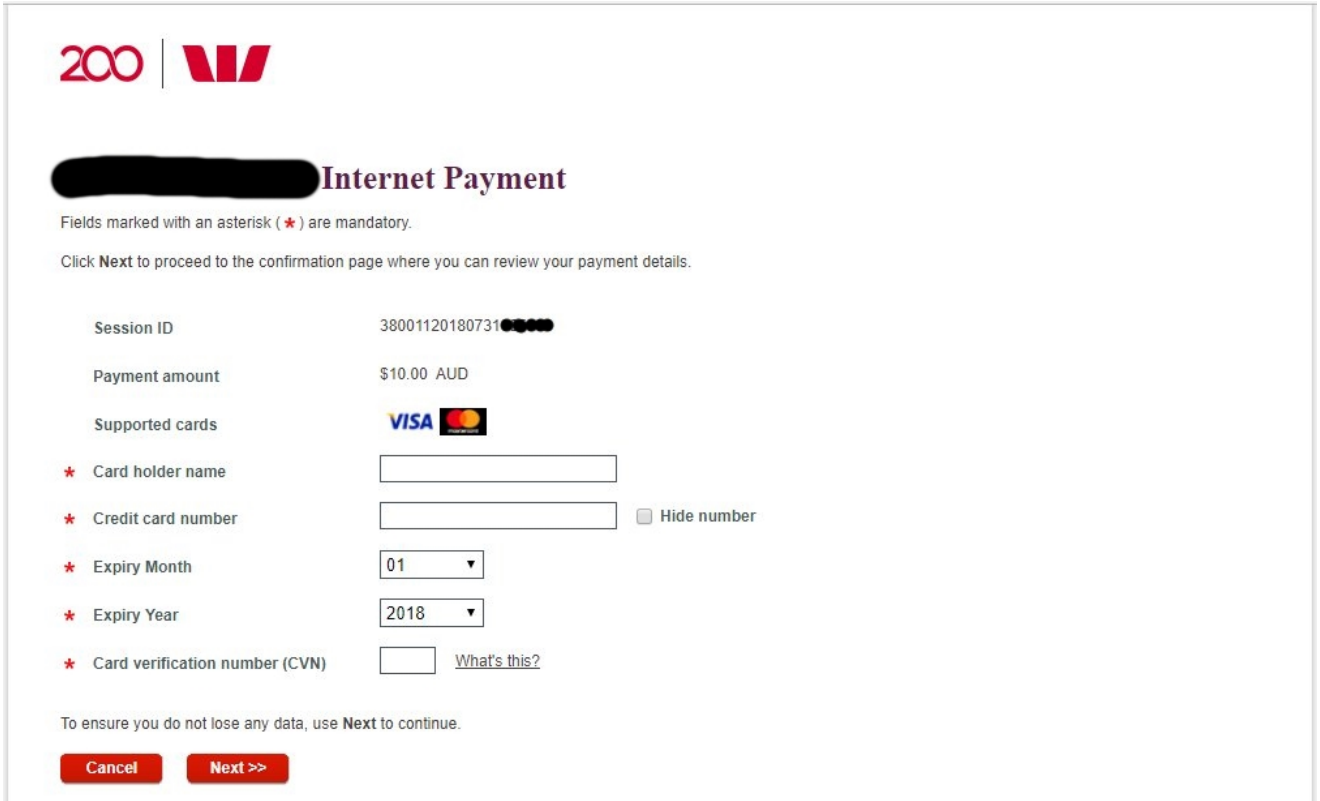


7. Resulting receipt:



Result

When the form is submitted, if the 'TM Payment Total' field has non-zero value and the 'Credit/Debit Card option' is completed, Transaction manager will either redirect the user to the payment providers website and payment page for the form filler to enter credit card information (see below) or a confirmation page in the case of BPAY.



Related articles

- Transaction Manager Payment Reconciliation and Reporting
- Configuring Payments in Forms
- Configuring Payments in Maestro

Transaction Manager Payment Reconciliation and Reporting

Unknown macro: 'redirect'

This article explains the process of generating reports from Transaction Manager that can be used to reconcile payments from a payment gateway.

Payment gateways generate daily reports which contain the payment transactions taken by the gateway each day. Reconciliation involves matching the payment taken by the payment gateway with Transactions recorded in Transaction Manager. Payment gateway reports normally contain, as a minimum, a unique reference number and an amount.

Important Concepts

Transact Manager Payment Log

Each time a payment attempt is made by Transaction Manager a new Payment Log entry is created to represent this attempt and this will include failed and cancelled payments.

Each Payment Log entry has a unique Payment Log Key which is used as a reference to link the payment attempts to the transactions. You can view the payment log through the Transaction Manager - > Operations -> Payment Transactions menu option.

Payment Gateway Receipt Number

Payment Gateways generate a unique reference of their own for payments and return this to Transaction Manager on a successful payment. This is stored in the Payment Log 'Do Merch Txn Ref' field.

Batch/Settlement Dates

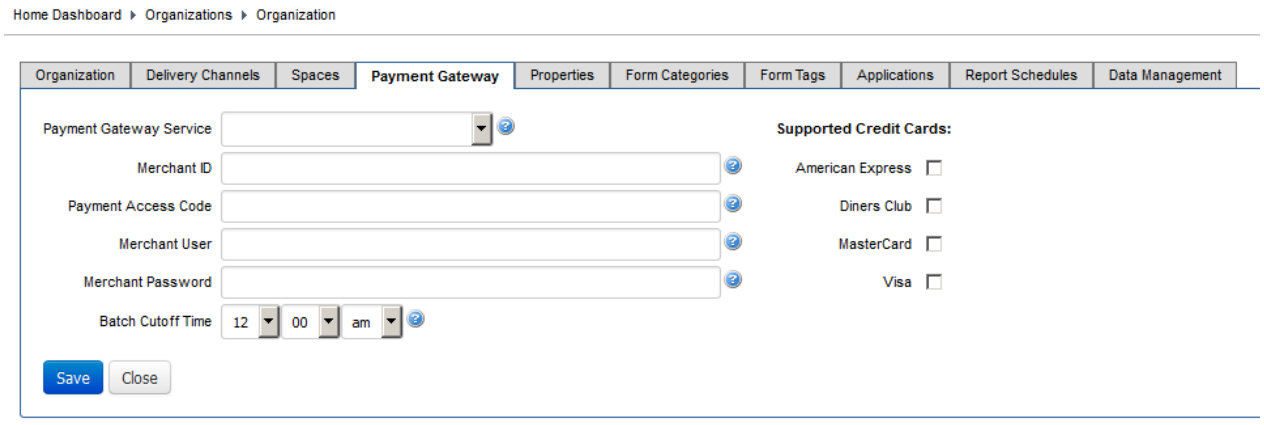
Payment Gateways have a concept of Settlement Date. This is normally defined as the day that the transaction is actually processed by the bank and is affected by the banks working day. This value is returned and stored in the Payment Log 'Dr Batch No' field.

This is a important field and will determine which daily report the transactions will appears on from both Transaction Manager and the payment gateways perspective.

Batch Cutoff

The batch cutoff is the time of day where transactions processed after this will be settled on the next business day by the bank. For example, if the banks cut-off time is 6:00 PM then a transaction processed on the 26/10/2015 at 8:00 PM will receive the settlement date of the 27/10/2015.

This is configured in Transaction Manager under the Payment Gateway configuration under each organisation.



Payment reporting (Payment Daily)

The out of the box report to use for payment reconciliation's is the 'Payment Daily' report. This report takes the following parameters to generate a list of payments taken for a particular batch (settlement date).

Field Name	Data Name	Comment
Organisation	client_name	User select this on TM to indicate which organization they what the report to run on. For a scheduled report this is the organisation that the schedule if configured against.
Batch	batch_id	For example, if it's set to "20151023", the report will show transactions that occur on 2015.10.23. For a scheduled report this is automatically calculated at the time the report is run and will be calculated to represent the last complete settlement day.

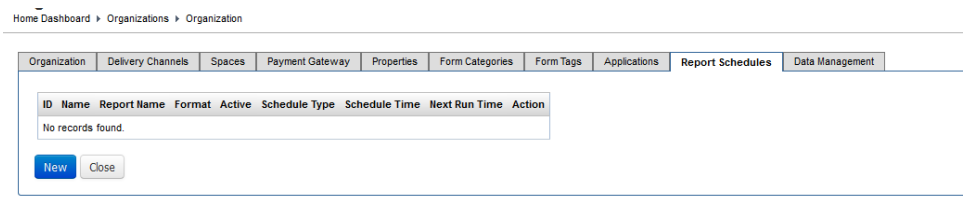
This report is a BIRT report and can be customized if required to include more information. For more information on updating BIRT reports please read the " TM BIRT report writer's guide" and "TM set up guide".

Scheduling Reports

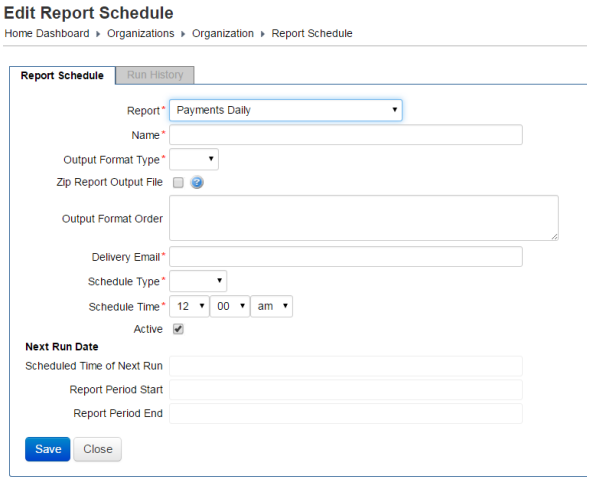
This section describes how to schedule a payment report to run periodically.

Step-by-step guide

1. Set up the report schedule on the organization level: TM->Forms->Organization-><organization name>.
   In the organization tab, go to "report schedule", click new.



2. Choose the desired report from the drop down list. Fill in the name, output format type, delivery email, schedule type, schedule time. Click save.



   NOTE: Please pay attention to the schedule run time as this will determine the batch that the report is generated for. If run time is before the daily cut-off time, it will report on the data for the previous day.
   For example, a report run at 6 AM on the 27/10/2015 with a cut-off time of 6 PM will generate data for the batch for the 26/10/2015.

3. If csv is your chosen output format type, you can specify/order the columns by using the data names and separate them by a comma. However, the data names are different from the headers

   See the table for the mapping between the headers(field names) and the data names.

Field Name	Data Name
Organisation	client_name
Batch	batch_id
Submission ID	submission_oid
Receipt Number	submission_receipt_no
Payment ID	payment_oid
Payment Time	dr_timestamp
Payment Receipt	receipt_no
Requested Amount	total_requested
Confirmed Amount	total
Total Requested Amount	total_requested_sum
Total Confirmed Amount	total_sum

Related articles

Content by label

There is no content with the specified labels

Signatures

Unknown macro: 'redirect'

- About Electronic Signatures
- Click-through Agreements
- Scribble Signatures (Sign-On-Glass)
- Understanding and integrating with Docusign
- Wet Signatures

About Electronic Signatures

Unknown macro: 'redirect'

There is a range of methods to implement electronic signatures in Avoka Transact. These patterns demonstrate some of these approaches.

Introduction to Electronic Signatures

Signatures are so entrenched in our culture that a reminder of what signatures are actually used for can be very helpful. Some background can be found here:

- http://en.wikipedia.org/wiki/Digital_signatures
- http://en.wikipedia.org/wiki/Electronic_signature

Why Signatures?

Signatures (either wet signatures or electronic signatures) can be used for the following purposes:

Purpose	Explanation
Ceremony or Assertion	Adding a signature (or historically, a seal) to a document adds a sense of gravity and importance to the whole process. Most people will treat a document and the information that they are providing with more care and respect if they are asked to sign it. Studies have shown that people are more likely to tell the truth if they put their signature at the bottom, or committed their name to it, they are more likely to act in accordance to it
Integrity	On paper documents, the integrity of the data is ensured due to the fact that printed words are difficult to modify. If a hand-written modification is made, often all parties are required to initial the modification.
Identity	A signature can be used to identify the user who signed the document (1:1 Biometrics). In fact, this is rarely used, and the user's name and other details are generally also collected.
Non-repudiation	Non-repudiation means that a person who has signed some information cannot at a later time deny having signed it. Historically, this is because signatures are difficult to forge. In other words, non-repudiation means that you can say "You can't weasel out of this, I have a document with your signature on it".

Avoka Transact Signatures Patterns

An electronic signature is "any electronic means that indicates either that a person adopts the contents of an electronic message, or more broadly that the person who claims to have written a message is the one who wrote it".

Avoka Transact provides several different types of electronic signatures, which range from extremely simple to quite sophisticated.

Each of the different patterns will be assessed against each of the Signature requirements: ceremony, identity, integrity, and non-repudiation.

Carefully consider your needs

Think very carefully about what your needs are. All electronic signature solutions add a level of complexity to the overall transaction to the customer. Consider asking the questions such as:

- What is the worst that can happen if...?
- Is my company policy about digital signatures appropriate for this use case?
- Does the legal department who are demanding particular technological solutions aware of the implications of their decision?
- Is this use case concerned about ceremony, identity, or non-repudiation?

Always use the simplest pattern that will satisfy your needs.

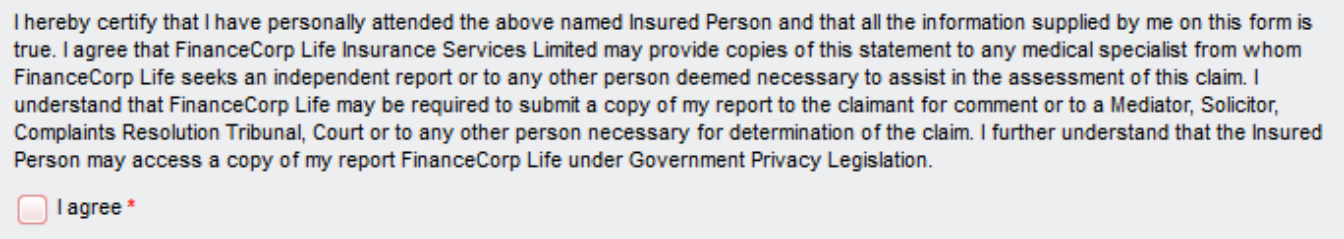
Related articles

- Signatures
- Wet Signatures
- Understanding and integrating with Docusign
- Scribble Signatures (Sign-On-Glass)
- Click-through Agreements

Click-through Agreements

Unknown macro: 'redirect'

The simplest way to implement an electronic signature is not to have one. In many cases, it's sufficient just to get a customer to perform a click-through agreement, without any real signature at all. With this strategy the user must manually select an input control (e.g. checkbox) or button to indicate their agreement to pro-forma or personalized passage of text:

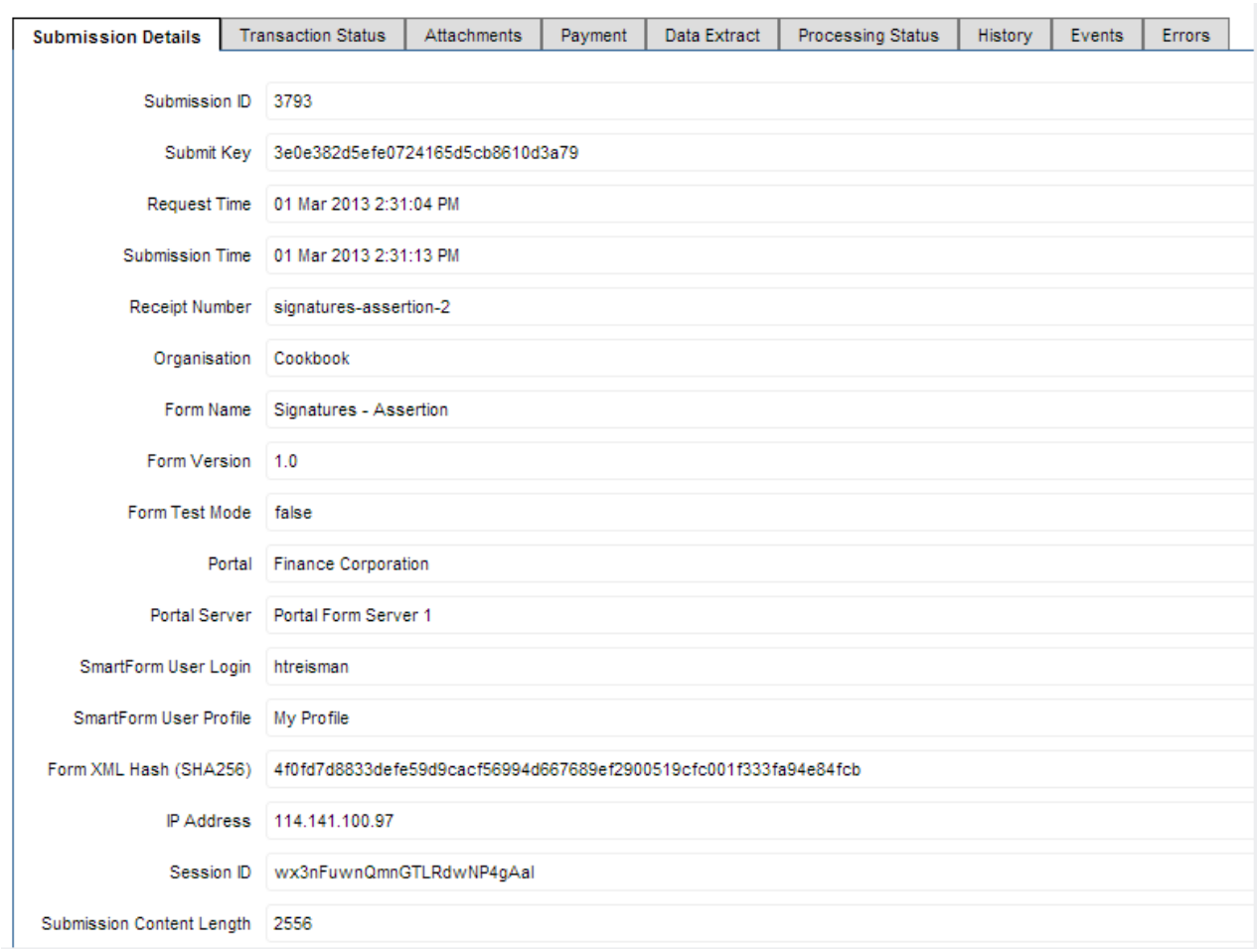


It is important to ensure that the click-through ceremony is mandatory and that the user cannot progress past this point without having indicated their agreement.

Criteria	Complies
Ceremony	Yes. The user feels like they are entering into a signature-type ceremony.
Integrity	Partially. Transaction Manager computes and stores the hash of the submission data. This could be used as evidence that the data has not been modified.
Identity	Partially. If the user is a registered user, then they would have had to sign in using their username and password. They may also have had to go through email verification, depending on setup. If the user is an anonymous user, then identity is not verified, but we do have some information about them and their IP address. There is an optional step where we can verify email address after submission.
Non-Repudiation	Partially. The combination of Ceremony and Identity give us some level of non-repudiation.

Pros	Cons
• Extremely simple • Works well on any device • All done within the form, no additional steps	• May not provide the level of integrity and identity (and therefore non-repudiation) that are required.

Note that Transaction Manager maintains a full audit trail and submission details about every submission, including the user's login identity, date and time of submission, and even the IP address of the user. This can be used as proof that a particular user did complete a particular transaction.



Reference

- http://blog.ericgoldman.org/archives/2010/02/clickthrough_ag_1.htm

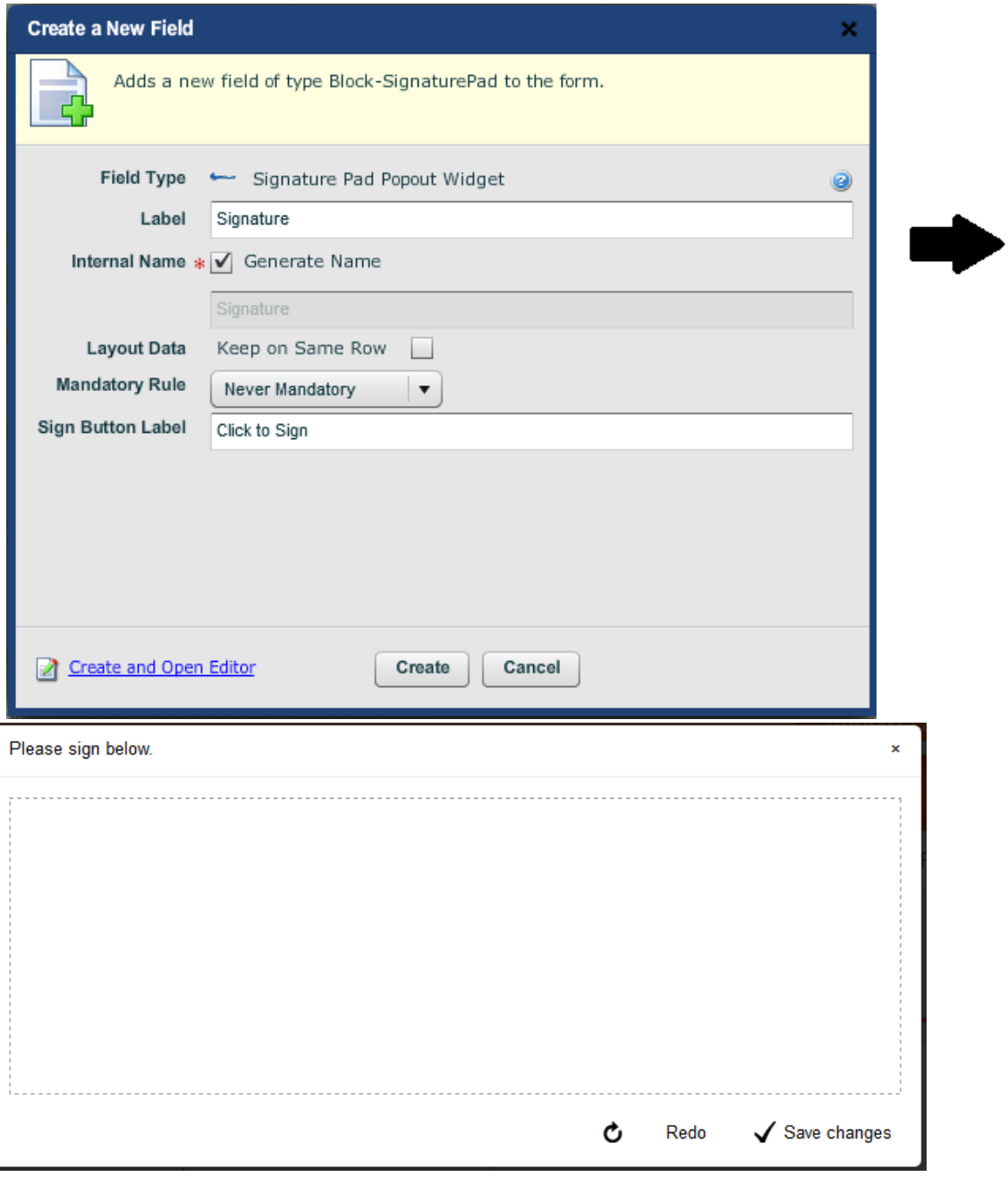
Related articles

- Signatures
- Wet Signatures
- Understanding and integrating with Docusign
- Scribble Signatures (Sign-On-Glass)
- Click-through Agreements

Scribble Signatures (Sign-On-Glass)

Unknown macro: 'redirect'

The simplest way to replicate existing paper signatures is to use the signature pad popout widget. This provides a very similar experience to signing a real piece of paper.



It is useful on devices with touch-input, and allows users to sign-on-screen using your finger or stylus but can be used with a mouse too.

Criteria	Complies
Ceremony	Yes. The user actually performs a signature-type ceremony.
Integrity	Partially. Both user and organization have a copy of the flattened receipt that contains an image of their signature. Transaction Manager computes and stores the hash of the submission data. This could be used as evidence that the data has not been modified.
Identity	Partially. Same as Assert pattern.

Non - repudiation	Partially. The combination of their scribbled signature, Identity and Integrity give us some level of non-repudiation.

Pros

- Extremely simple to implement
- Feels like a real signature that people are familiar with
- Very convenient, even for multiple signers at a single ceremony.

Cons

- May be difficult to do for mouse users
- Some people may feel that their scribble signature doesn't look much like their real signature.
- Some people may be concerned that a generated signature isn't a valid signature (even though this is not in fact a legal issue - generated signatures are legally binding in most jurisdictions.)

Any-device compatibility

- Desktop: It works, but not a great user experience
- Tablet and Phone: Works really well, with a stylus, OK with finger

Notes

- You can have as many signature pad popout widgets on a form as required.
- The signature is saved as an image in the XML and is automatically reproduced on the PDF receipt.

Related articles

- Signatures
- Wet Signatures
- Understanding and integrating with Docusign
- Scribble Signatures (Sign-On-Glass)
- Click-through Agreements

Understanding and integrating with Docusign

Unknown macro: 'redirect'

Docusign Overview

This article will help you understand how to enable form transactions to use the DocuSign service to have one or more signatories sign a PDF receipt before a submission is marked as complete.

The article is focussed on a simple form complete and sign by two signatories delivery process.

The diagram below describes the sequence of events in a two signature flow.

Docusign bundles documents that need signing in to what they call an **envelope**. When an event happens like a document has been signed the status changes. There are 2 ways for 3rd party systems to integrate with DocuSign to determine when the envelope status changes .

1. **Expose a call back rest endpoint**. Docusign calls this endpoint when the status changes. This is Docusign's preferred method.
   The issue with this approach is On Premise TM installs would require you to make a firewall rule that exposes the rest endpoint to the internet.
2. **Polling of a Docusign rest service**. This works for On Premise as well as Hosted systems.

The integration in TM uses **Polling**. The Docusign integration uses a delivery service with check points. DocuSign does not want it's servers to be overwhelmed by excessive polling. They have a terms of service that the minimum polling interval for an individual envelope is 15 min where the polling is automated. The polling interval is determined by the service parameter **retryDelayMin**

Creating a new Transact Manager docusign delivery service uses a transact template. This will generate generic Groovy script code which can be customised to your needs.

Docusign - The Demo

This demo assumes you already have a working Docusign account and are aware of the account details.

The Docusign demo consists of two solution components:

1. Composer widgets to format the Docusign signature panels
2. Transaction Manager (TM) Checkpoint Delivery process

To make life simple in this demo we will just have one signatory but the concept is the same for adding additional signatories.

The demonstration will work like this:

1. Fill and submit the form with Name and Email address parameters.
2. Recipient receives email notification of signature request and link. Recipient proceeds to the Docusign website and completes signature experience using DDocusignfacilities. A Docusign account is not required.
3. Upon confirmation, Docusign sends email notification to the Docusign account holder and signatory that the signature process has finished.
4. TM polls Docusign to see if the form has been signed. Once an acknowledgement of signature is received the delivery process is complete and the submission is marked as complete.

Developing a Docusign Demo

Developing a demonstration using Docusign is a relatively straightforward process:

1. Drop Composer widgets onto the desired form
2. Deploy Form and configure Docusign delivery process
3. Hook the delivery process to your organisation
4. Hook the delivery process to your form

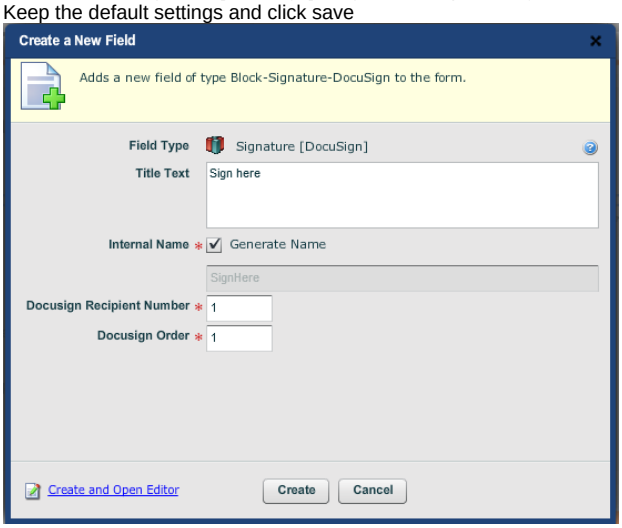
The Composer Widget is reasonably simple, consisting of only a name to appear as a label on the form next to the Name and Email address fields which are used at runtime to identify the signatory.

The Docusign Checkpoint Delivery Process uses Service Parameters to control behavior. The demonstration uses default values for Docusign account and password, and a default period for Polling the completed signing process. Once signing has completed, the Delivery Process will deliver and finalise the submission.

So, lets get down to business:-

Part 1 - Composer

1. Log onto Composer
2. Create a new form in Composer - just use defaults to create a basic form
3. Search for the Signature[Docusign] widget and drop it onto your form
4. Keep the default settings and click save

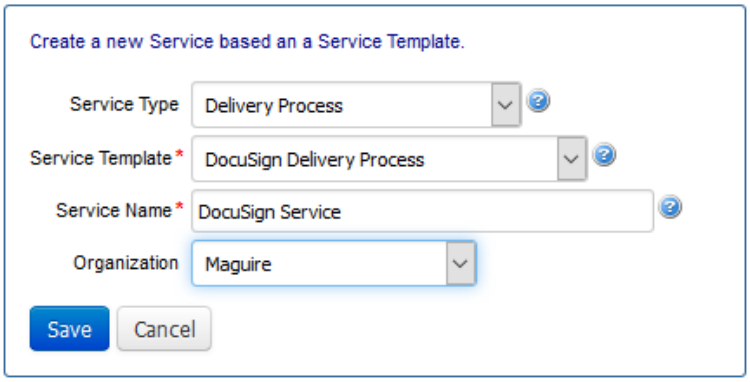


5. Make sure you also add a Submit button to the form
6. Save your form and publish to your test TM instance
7. Simple huh?

Part 2 Manager

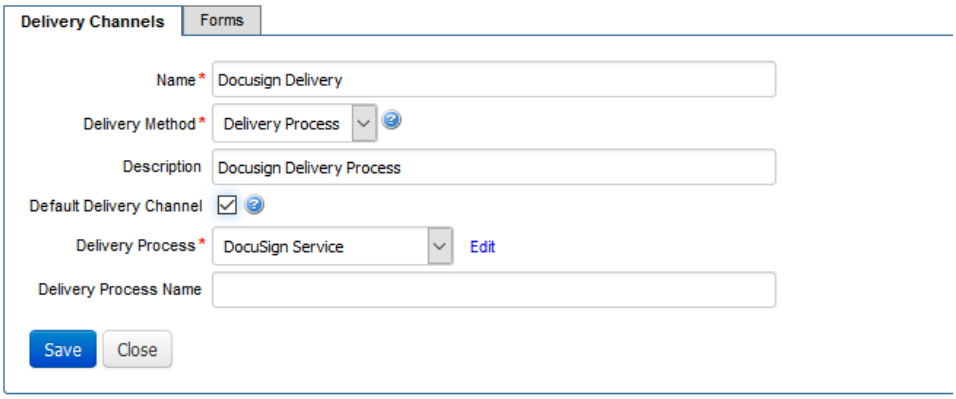
OK this part is a little more technical but is a one time set up. Once you have it configured you can have as many forms as you like using the Docusign delivery process.

1. Log onto your test TM instance
2. Go to System->Service Definitions
3. Click on New and create a service with the settings below, make sure the organisation is the same one you published the form to



4.
5. Click on the service's Parameters tab
6. You will see the default Avoka docusign parameters, you will need to change these for your own docusign account details, this is all Parameters with a Docusign prefix
7. The template process is set up to be the final delivery process so the Final Delivery Process parameter points to an "empty" Trash can delivery process. You can use this parameter to point to another delivery process once the Docusign process is complete. E.g. you may wish to pass the form submission data to one of your internal systems
8. As mentioned above, retryDelayMins must be greater than 15 mins.
9. Finally we need to hook your form up to use your new delivery process
10. Navigate to Forms->Organisations
11. Click on the Delivery Channels tab and click new

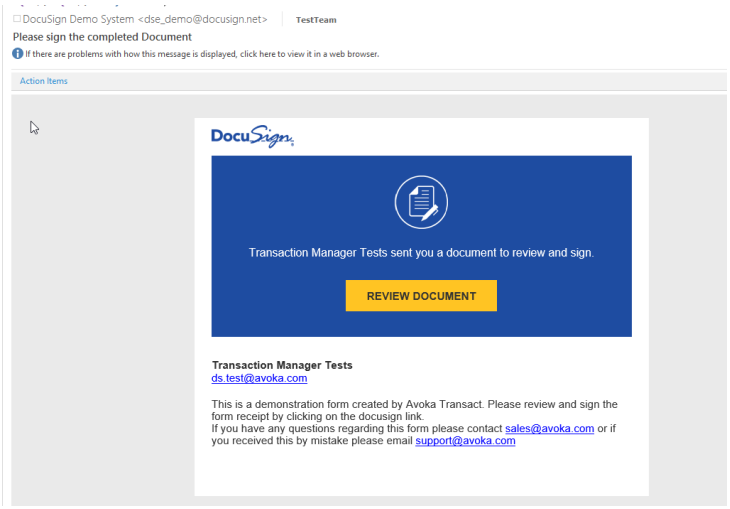
12. Use the settings below



13. Save
14. Navigate to Form-> Forms
15. Open your form and go to the Details tab
16. Look for the Delivery Channels section and select your delivery channel as the Production and Test Delivery options.

Now it is time to test your form! Render your form, fill in your email and name and click submit.

If you navigate to Operations-> Form Transactions you can monitor your submission's progress. You should receive a Docusign email such as the one below and once you have signed your receipt you should see the submission move to a Delivery Complete state. N.B this will all take a few minutes dependent on how often your delivery process is scheduled to run.



You can retrieve your signed docusign document by opening your form from Operations-> Form Transactions and clicking on the attachments tab.

Related articles

- Signatures
- Wet Signatures
- Understanding and integrating with Docusign
- Scribble Signatures (Sign-On-Glass)
- Click-through Agreements

Wet Signatures

Unknown macro: 'redirect'

In some cases, you need a real "wet" signature on a piece of paper.

You can still achieve the benefits of electronic data capture using the following flow:

1. Have users fill in the form electronically. This signature section will say "Not required to sign" as shown in the sample.
2. When the user submits, they will print out their receipt. The receipt will contain the proper wet signature block, as well as a receipt number in both human readable and barcode format.
3. The submitted data will be delivered to the organization, but it will be put into a holding queue and not processed.
4. The end user will sign the printed copy, and send it in by fax, mail or email.
5. When the received wet signature is received, the paper receipt will be matched up to the electronic submission in the holding queue, and if everything matches, it will be released to fulfilment.

Because the data has already been captured electronically, when the signed "paper" is received, there is no need to re-key any data. All that is required is to verify that the signature has been received, and the paper form is filed. After that, the process can revert to being completely electronic.

Sample form: https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=signatures-wet3

Criteria

Criteria	Complies
Ceremony	Yes. This is identical to a regular wet signature signing ceremony.
Integrity	Yes. The company owns a paper copy of the document. (Note that if the paper form is submitted electronically or by fax, integrity is much harder to prove.)
Identity	Transaction Manager provides the same levels of identity verification as for the Assertion pattern (login, audit-trail). To be honest, holding a user's signature doesn't really improve the level of identity, unless the organization already owns a sample signature, and verifies the new form against that signature. Few organizations do this in practise.
Non-repudiation	Yes. The organization holds a signed paper copy of the document.

How does the barcode work?

- The printed barcode records the Avoka Transact Receipt Number - a unique identifier for the submitted data.
- A barcode reader is just an alternate keyboard. When scanning a barcode, it 'types' the Avoka Transact Receipt Number wherever the cursor is.
- It is up to the back end system to provide a user interface to retrieve a pending application from a database. This is often done in a BPM or ERP system.

Pros

- Identical to current wet signature flows, with the added advantages of:
  - The end user does the data entry instead of a team of data entry operators.
  - Data is more likely to be correct since it's being typed rather than data entered from a hand-written form.
  - A SmartForm provides a better user experience.
  - Legally has the same status as the existing paper process.

Cons

- Considerably more complicated process for the end user
- The organization is required to implement the "hold while waiting for signature" queue.

Any-device compatibility

- Desktop: It works well
- Tablet & phone:
  - Most people don't connect a printer to their phone or tablet. You will need to download and Email the receipt yourself.
  - You could use a PDF editing application such as Adobe Reader (or 100 other apps that are built to sign a PDF), but the issue is then that you can't upload a file in iOS

One way to address this issue, is to make the wet signature a part of the delivery process. This way, we Email the PDF which needs to be signed and posted in. This makes more sense from the user perspective, as most people will need to refer back to the Email to lookup the address to put on the front of the envelope.

Notes

- You can have as many wet signatures on a form as required, and the printed form can be distributed to the signers just like a regular paper form.
- You may want to include a TxM Submission Receipt block above the first section. This will be invisible during data entry process, but contains the receipt number and barcode when the receipt is generated.

Related articles

- Signatures
- Wet Signatures
- Understanding and integrating with Docusign
- Scribble Signatures (Sign-On-Glass)
- Click-through Agreements

Styling & Branding

Unknown macro: 'redirect'

- Composer Style sheets
- Alternative touch button styles supported by the Maguire Template
- Creating a custom Composer skin
- CSS3 Web Fonts
- CSS Styling with Custom Classes in Composer
- How to add Custom Fonts to Composer
- How to style bullet points in a Composer form
- Images in Composer forms
- Including Web Fonts in Composer forms
- How to publish forms in multiple brands

Composer Style sheets

Unknown macro: 'redirect'

There are three different style sheet levels in Composer; Section, Theme, and Custom. Each is used to control a different part of the form.

Section is used to determine the structure of the sections in the form, i.e., the numbering system used for the section headings, the indenting around the sections, etc.

Theme is used to determine the colours used in the form

Custom is used to adjust individual widgets, i.e., changing fonts, layout of text fields and radio buttons, or changing the spacing between fields.

The purpose of breaking up these categories is to make each style sheet more reusable. By breaking them up you are able to re-use a section style sheet with a different colour scheme, without having to build a whole new style sheet from scratch.

Style sheet precedence

Each style sheet overrides the one preceding it. The Section style sheet is the base style sheet applied to the form. Anything in the Section style sheet can be over-ridden in the Theme style sheet, which can also be overridden in the Custom style sheet. If you are making a change to a style sheet, and it is not coming through to the form, ensure that it isn't being over-ridden in a different style sheet.

Creating a new style sheet

Style sheets are maintained on an organisational level. To create a new style sheet, navigate to an organisation, then switch to the 'Style Sheets' tab at the top. At the top of the tab, click the 'New' icon' Give the style sheet a name, and select the category. You also have the option to make a copy of an existing style sheet to use as a start point.

Editing a style sheet

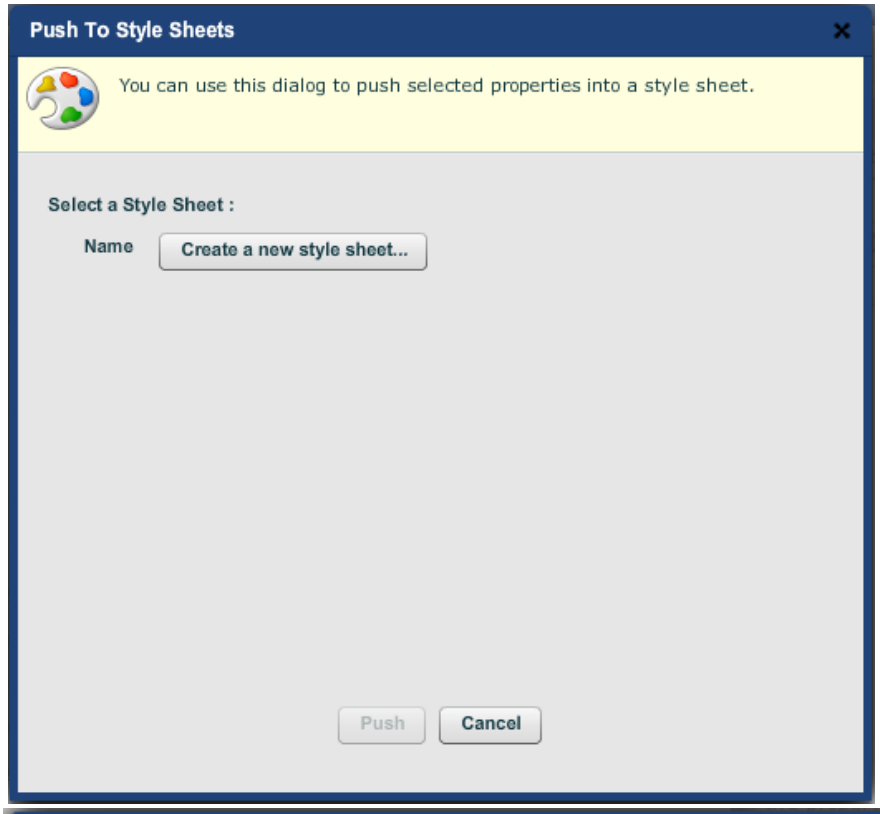
There are several ways to edit a style sheet:

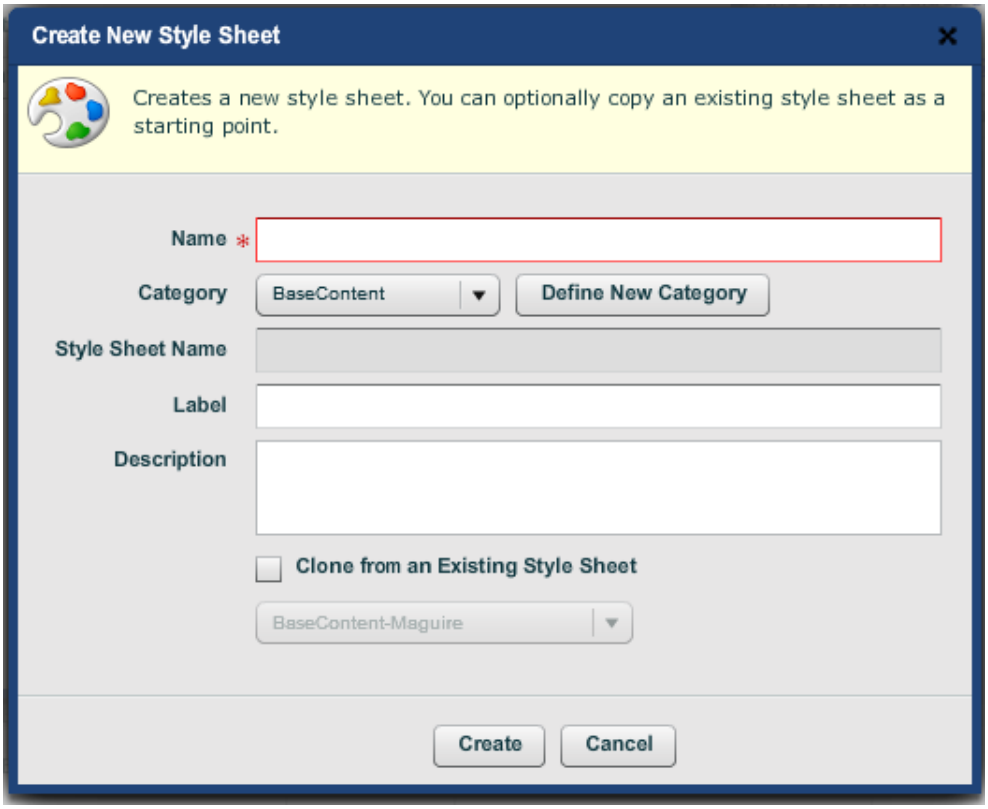
Directly through the 'Edit Style Sheet' interface

By clicking the 'Edit' button for a style sheet, the 'Edit Style Sheet' interface opens. Each style appears in a list of styles to the left, and their properties and values appear on the right. To edit the style sheet, find the appropriate style and property, and edit the value. Alternatively, it is possible to modify the XML directly by switching to the 'XML Source' tab. A good way to do this is to build the widgets you wish to style in a new form, manually change the style, then switch to the XML source tab at the top right, and copy the XML from the form into the style sheet.

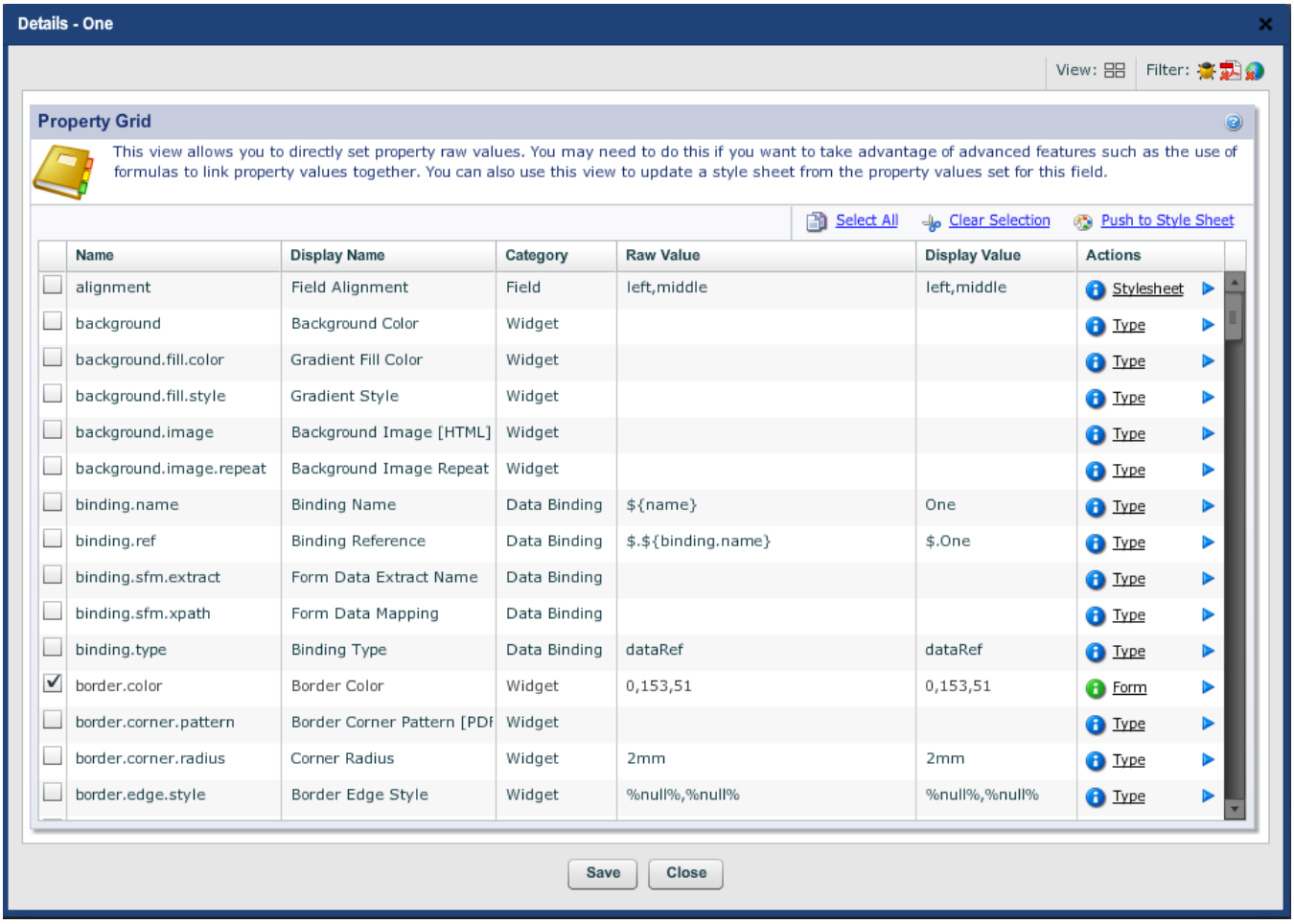
Using 'Push to style sheet'

When styling a form manually through the form builder, the interface will indicate where styles are being applied from. If a style is being applied from the form, and not a style sheet, it is possible to push it to the style sheet, so that it is applied to the rest of the form.





To do this, switch to the Grid view by pressing the button at the top right of the window, then check the appropriate properties, and click the ‘Push To Style Sheet’ button.



Select the style sheet you wish to apply the style to, and click ‘Push’. Checking the ‘Remove selected properties…’ checkbox will remove the styling from the field, and pull it from the newly modified style sheet instead.

Typed style vs named style

A ‘Typed Style’ refers to a style that applies to all widgets of a particular type. Typed styles tend to be more general purpose, and changing a typed style will make more widespread changes to the form. Named styles refer to a sub-type of a typed style. Named styles will override typed styles.

For example, a typed style of ‘Text’ will refer to all text objects in the form. If you wanted to change the font used in a form, the ‘Text’ typed style would be the appropriate place to make the change. There may also be a named style in the form called ‘style.level1.headertext’. This object would also be included in the typed style ‘Text’. If you wanted to change the colour and size of the level 1 headers in the form, you would use ‘style.level1.headertext’. This would allow you to make changes to every instance of a ‘Header 1’, but wouldn’t be applied to every single text object in the form.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

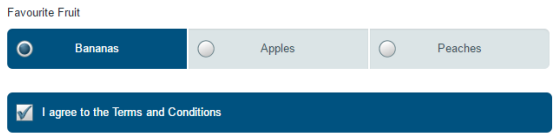
Alternative touch button styles supported by the Maguire Template

Unknown macro: 'redirect'

Compatibility

Since	4.1
Deprecated	

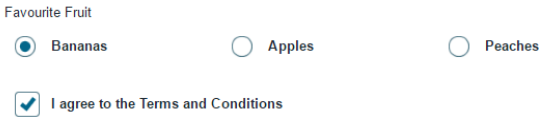
The standard touch button styles (applied to radio button groups and checkboxes) in Maguire forms have been designed to be highly intuitive, responsive and touch friendly, however they may not be consistent with every organizations preference and branding.



Maguire 4.1 introduces 2 new out-of-the-box touch button styles that are available for use in brand implementations and demos.

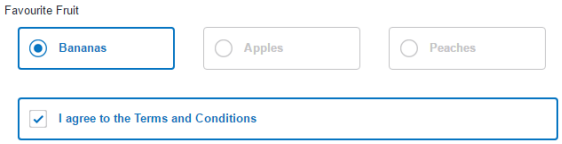
Naked Touch Buttons

The 'Naked' styling options presents touch buttons without the background shading, using icons that are more representative of a classic radio button or checkbox. Radio button captions are left aligned so as to be visual aligned with the selection icon. Selected options are indicated only by the icon change.



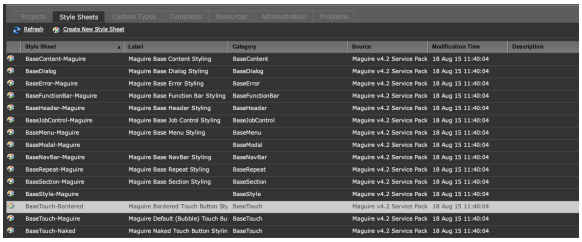
Bordered Touch Buttons

The 'Bordered' touch button style presents a styled border around each touch button. Radio buttons are not shown as being visually grouped like the default Maguire styling other than their proximity to each other. Font and border color highlighting is used to indicate the selected option, in addition to the icon change.



How do I test out these new styles?

Testing the new styles is very easy. Simply navigate to your Composer organization's 'Style Sheets' tab and modify the filter to 'All' so that you can see style sheets provided by sources outside your organization. If your organization is using the Maguire package then you should see 3 stylesheets in the category 'BaseTouch'. Note the name of the one you wish to try out in your form:



Open your Maguire based form and view the XML Source, then simply add the style sheet name to the front of the 'stylesheet' property of the top level 'formdescriptor' element, ensuring that a comma separates all style sheets listed in the property:

```
<formdescriptor description="" stylesheet="BaseTouch-Bordered,Styling-Maguire-
Default,Navigation-TopGroups" template="Template-Maguire">
```

Click the 'Apply XML Changes' button and preview your form with the updated touch button styling.

Alternatively, you could create your own template that exposed the BaseTouch category selector in the Composer design UI by setting visibility="visible". This will allow you to modify the touch button styles of the form without modifying the raw XML. Template inheritence can be used to apply this visibility change but remain in the inheritence tree for the Maguire Template as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<formtemplate name="Template-Maguire-NakedTouch" extends="Template-Maguire"
label="Maguire Naked Touch Template" sortorder="50"
             description="This a sample template that extends Maguire and
uses naked touch button styling">
    <stylesheets>
        <category name="BaseTouch" visibility="visible"/>
    </stylesheets>
</formtemplate>
```

Note: the simplest way to create an inherited template in Composer is to add it to a data pack and import it to your organization.

How do I apply one of these new styles to my brand?

These styling options are deliberately not provided as selectable options in the Composer UI as it is fully expected that most organizations will standardise on styling across their brand and will not wish to allow form designers to select different styles in the design process. Rather, the preferred styling option (or variant thereof) will be fixed in the organizations template by simply setting a different 'BaseTouch' stylesheet option.

The Maguire template sets the default BaseTouch style sheet to 'BaseTouch-Maguire' as shown below:

```
    <stylesheets>
        ...
        <category name="BaseTouch"       label="Base Touch"        default="Ba
seTouch-Maguire"       visibility="hidden"/>
        ...
    </stylesheets>
```

This default can be over-ridden in organization specific templates by replacing this default with the preferred style sheet option of 'BaseTouch-Naked' or 'BaseTouch-Bordered'. Template inheritence can be used to apply this style sheet change but remain in the inheritence tree for the Maguire Template as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<formtemplate name="Template-Maguire-NakedTouch" extends="Template-Maguire"
label="Maguire Naked Touch Template" sortorder="50"
             description="This a sample template that extends Maguire and
uses naked touch button styling">
    <stylesheets>
        <category name="BaseTouch" default="BaseTouch-Naked"/>
    </stylesheets>
</formtemplate>
```

Note: the simplest way to create an inherited template in Composer is to add it to a data pack and import it to your organization.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

Creating a custom Composer skin

Unknown macro: 'redirect'

Creating a custom skin allows you to create a set of display rules for previewing a form in Composer. For example, we built a skin for a client to allow them to create a review version of a form, which had rules to override the display logic in the form, so that all fields were visible.

Creating a custom skin

The best way to create a new skin is to export an old one. To do this, navigate to Libraries, select the latest release version, and right click and select 'Export…'. In the exported zip, navigate to 'Skins', and copy one of the skins to a new location. In the skin xml file, change the skin 'name' and 'label' properties.

Modifying a custom skin

Skins are essentially a list of property overrides that are applied to a form. These overrides appear in the following format:

```
<module name="Block">
    <property name="rule.visibility" value="always"/>
</module>
```

This example overrides the visibility rule for all 'Block' objects in the form, so that they are always visible.

To modify the skin, add property values to the appropriate modules. To find what modules are available, refer to the library export – anything defined in the 'blocks', 'sections', 'types', or 'widgets' folders is a module, and the properties defined in the xml definition file (or the xml definition of any of the objects that it extends) can be set in the skin.

It is also possible to override named styles in a skin. E.g.,

```
<style name="style.wizard.menu.header">
    <setproperty name="layoutfill" value="horizontal"/>
</style>
```

The easiest way to do this is to copy the style tags directly from the style sheet, then modify the property values.

Making a new skin available

The template used by a form dictates which skins are available. To make the new skin available, you must modify the template to include the new skin. Skins are defined as 'technologies' in the template. To include the skin, make a copy of the technology that matches the new skin the closest (e.g., for a PDF skin, use the 'pdf-interactive' technology), and modify the appropriate properties. The properties you will need to modify are:

- name – a unique name for this technology
- output-file – the filename that the form will preview to
- label – the name the skin will display as when previewing in composer
- skin – this should match the 'name' property in the skin xml definition

Next you will need to import the skin and the template into an organisation. To do this, export an organisation, extract zip file, and delete everything except for the organisation.xml file. Create a new folder called 'skins', and a folder called 'templates'. Move the skin to the 'skins' folder, and the template to the 'templates' folder. Zip the organisation, and reimport to Composer.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

CSS3 Web Fonts

Unknown macro: 'redirect'

Adding web fonts can provide some great flexibility and creativity to your forms and user portal web pages. An example of how to achieve this is outlined below.

Some additional resources and further reading on this topic include:

- http://www.w3schools.com/css/css3_fonts.asp
- http://www.w3schools.com/cssref/css3_pr_font-face_rule.asp
- http://css-tricks.com/snippets/css/using-font-face/
- http://www.w3.org/TR/css3-fonts/

Important disclaimer: Adding web fonts can add additional page load delays to your form and/or web page. The web browser needs to interpret the HTML content and then determine which resources it needs to fetch in order to render the page. It is highly recommended to profile the page load time using freely available 3rd party tools such as **PageSpeed** https://developers.google.com/speed/pagespeed/

And now the example:

Step One: Obtain the source files for the font that you wish to use. True Type Font (TTF) is widely supported by most modern browsers, but it can be useful to also include Web Open Font Format (WOFF) and Embedded OpenType (EOT) as well. For the purposes of this example, I've obtained the sample TTF file, orangejuice2.ttf, from: http://www.dafont.com/

Step Two: Deploy the font file as a Transact user portal resource, e.g.: <portal base>/resources/font/orangejuice2.ttf. This ensures that the font file will be available to anyone attempting to access your form.

Step Three: Create your CSS font face definition(s):

```
@font-face {
  font-family: "orangejuice";
  font-style: normal;
  font-weight: normal;
  src: url("/maguire/resources/font/orangejuice2.ttf") format("truetype");
}
```

You can specify multiple font formats as:

A bold type:

```
@font-face {
    font-family: "orangejuice";
    font-style: normal;
    font-weight: bold;
    src: url("/maguire/resources/font/orangejuice2-Bold.eot?#iefix") format("embedded-opentype"), url("/maguire
/resources/font/orangejuice2-Bold.woff") format("woff"), url("/maguire/resources/font/orangejuice2-Bold.ttf")
format("truetype");
}
```

An italic type:

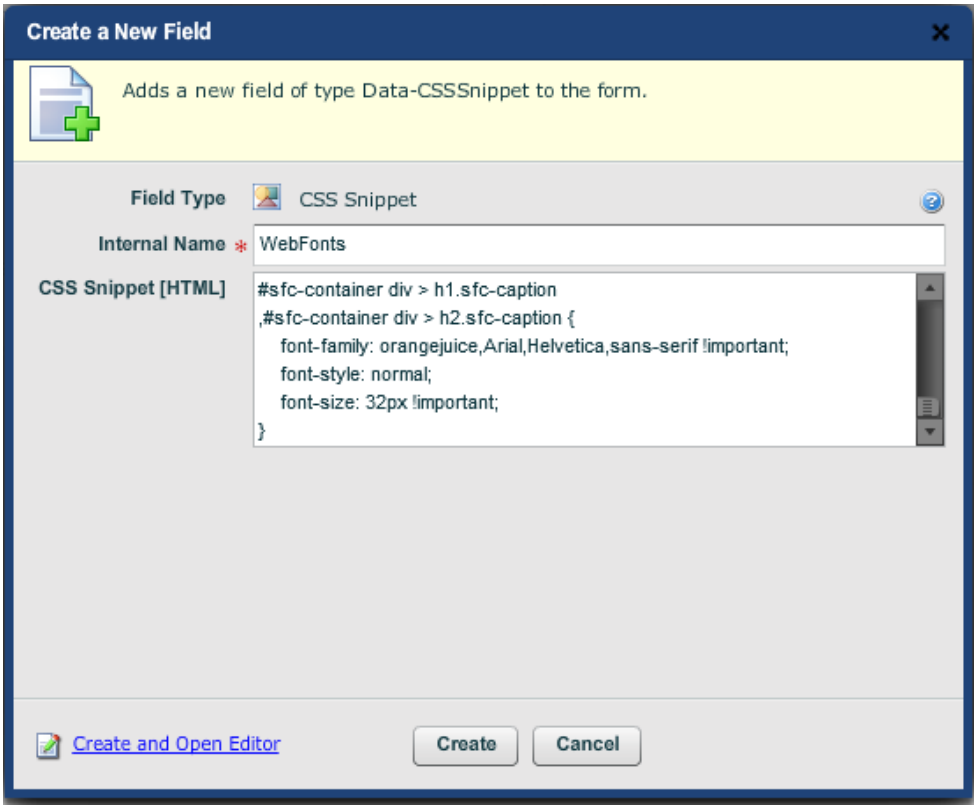
```
@font-face {
    font-family: "orangejuice";
    font-style: italic;
    font-weight: normal;
    src: url("/maguire/resources/font/orangejuice2-Oblique.eot?#iefix") format("embedded-opentype"), url("/maguire
/resources/font/orangejuice2-Oblique.woff") format("woff"), url("/maguire/resources/font/orangejuice2-Oblique.
ttf") format("truetype");
}
```

Step Four: Specify the font-family attribute in your CSS classes (you will most likely need to inspect your HTML content to ensure that you're overriding the correct CSS class path):

```
/* override the default heading font */
#sfc-container div > h1.sfc-caption
,#sfc-container div > h2.sfc-caption {
    font-family: orangejuice,Arial,Helvetica,sans-serif !important;
    font-style: normal;
    font-size: 32px !important;
}
```

Note: It's important to also include some fallback font options (in this case Arial, Helvetica, ...) in case the web browser is unable to obtain or display the web font you have specified.

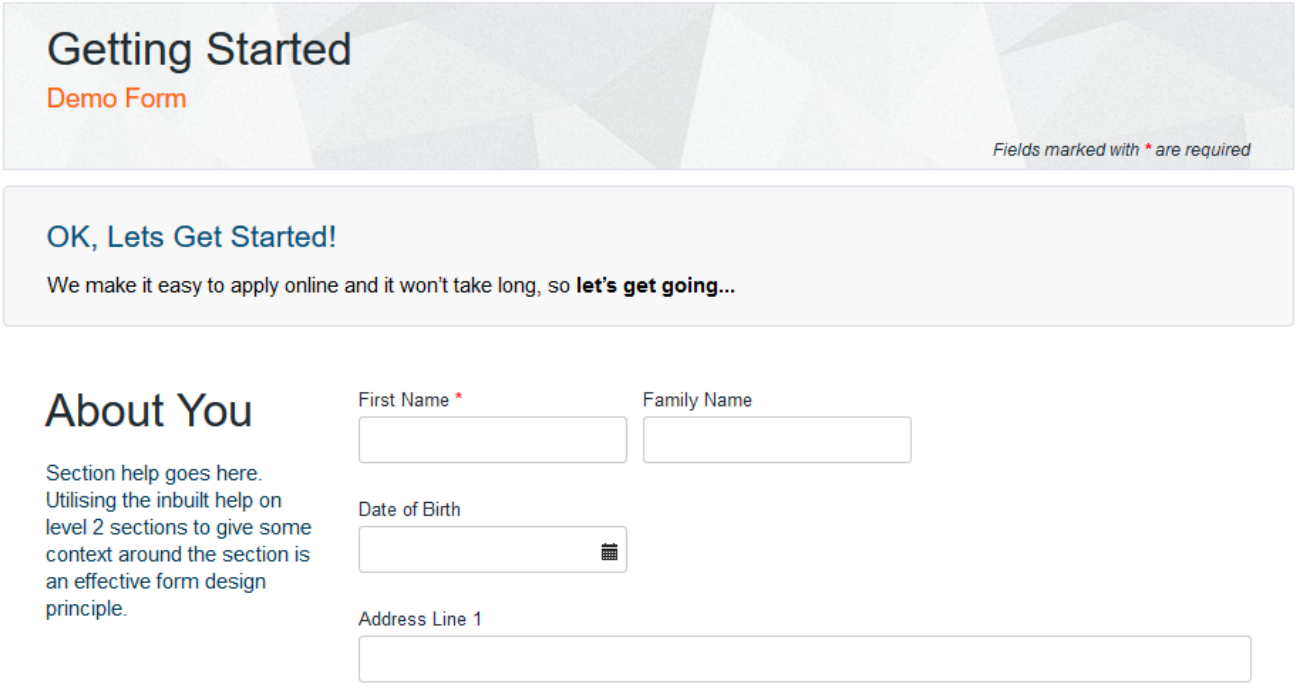
Step Five: Add your CSS to your form, deploy and test



In this example, the CSS web font details have been added to the form's *CSS Snippet* style sheet.

Note: In some circumstances it can be preferable to provide this in an external CSS resource file so that both the Composer form and Transact user portal HTML content can refer to the same definition (and you only need to maintain it in one place).

And finally, the result:



Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

CSS Styling with Custom Classes in Composer

Unknown macro: 'redirect'

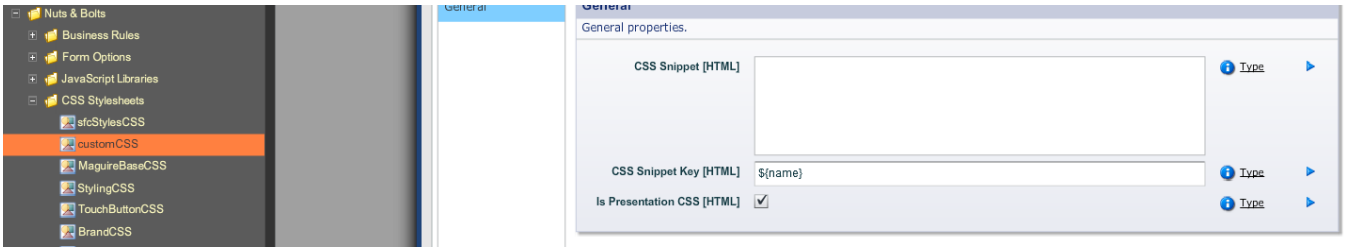
To indivdually style a widget, one can use custom classes.

Step-by-step guide

Whatever entered into the Custom Classes property is inserted into the widget's css classes at run-time.

```
<div class="style-346 sfc-ui-fill layout-608 sfc-bindable testclass sfc-rule-default sfc-mandatory" data-sfc-name="Surname"
data-uid="/BasicDetails/_outerArea/_contentArea/BasicDetails/_outerArea/_contentArea/Surname" data-guid="568">…</div>
```

You can now write css to impact widgets with that specific class.



Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

How to add Custom Fonts to Composer

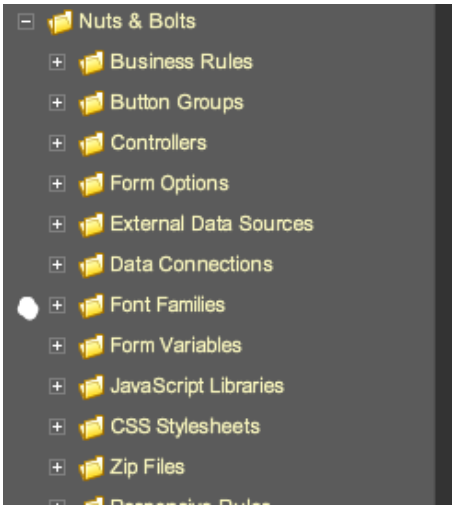
Unknown macro: 'redirect'

Occasionally there may be a need to use a custom font when styling Composer Forms to add to the form presentation and comply with a company Style guide.

The following article describes the process to add the font family to Composer so that forms can use the new font for form presentation and rendering as PDF.

Steps

1. Open your form in Composer and go to Nuts & Bolts > Font Families



Please note that Avoka does not support embedding custom fonts in PDF at this point in time.

This is not just a Composer limitation, as we only pass an XDP to TM and an XDP cannot embed fonts. In addition, my understanding is that TM PDF receipt rendering is not set to embed fonts. The TM team can provide more information on this functionality.

2. You will need to add your font to the appropriate Font Family.

Sans-Serif

In typography, a sans-serif typeface is one that does not have the small projecting features called "serifs" at the end of strokes.

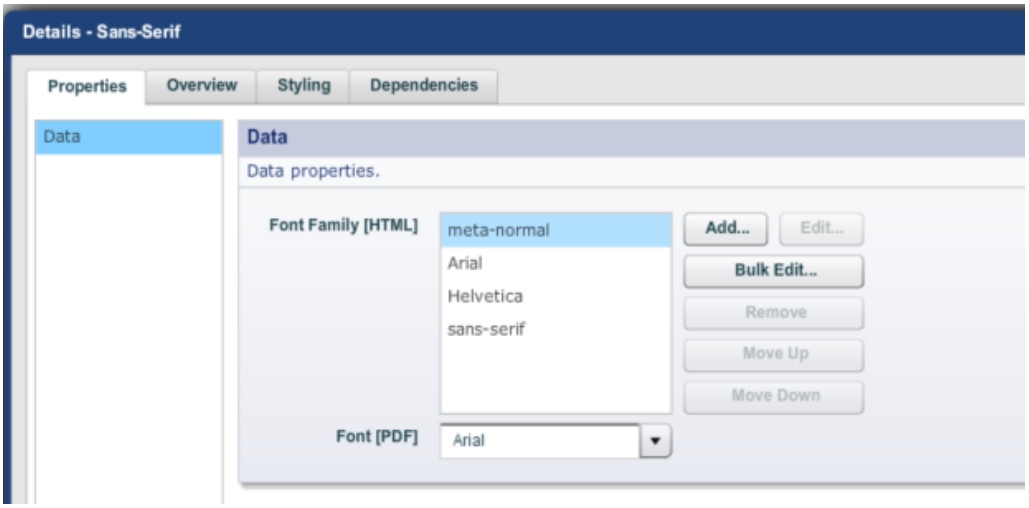
Serif

In typography a serif is a small line attached to the end of a stroke in a letter or symbol.

Monospace

A monospaced font, also called a fixed-pitch is a font whose letters and characters each occupy the same amount of horizontal space.

To add, click the font family you want to add

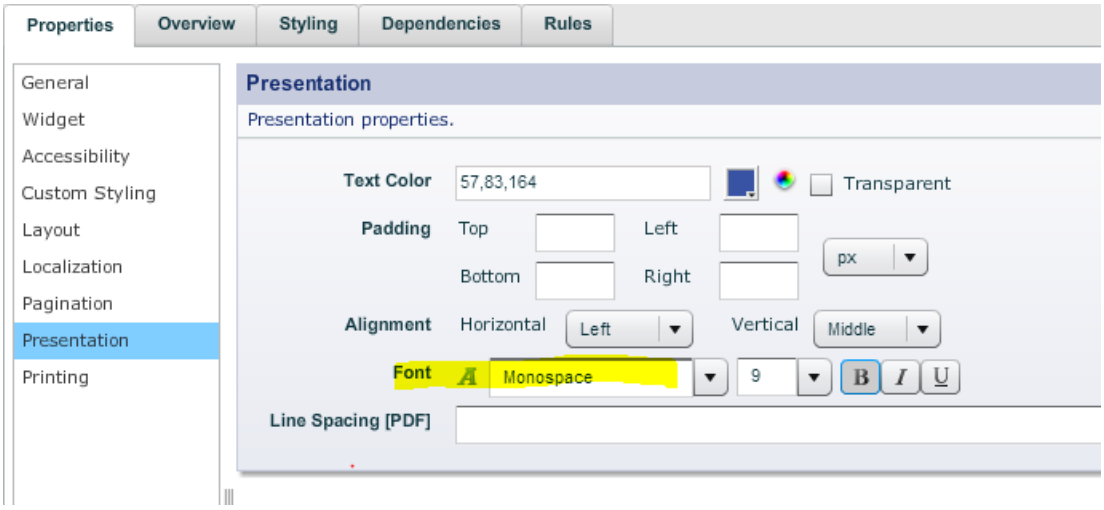


3. Ensure that the font has been loaded to the Operating system of the Composer instances that will be serving the forms.

For more information on loading fonts please refer to

http://windows.microsoft.com/en-au/windows-vista/install-or-uninstall-fonts

4. Once loaded you will see the Font appear in the Text Editor menu and the Field Edit Properties Windows



N.B. If you are using LiveCycle Output you will also need to Load the fonts into the LiveCycle > Fonts folder.

The following steps are for Composer 3.6.1 and above

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

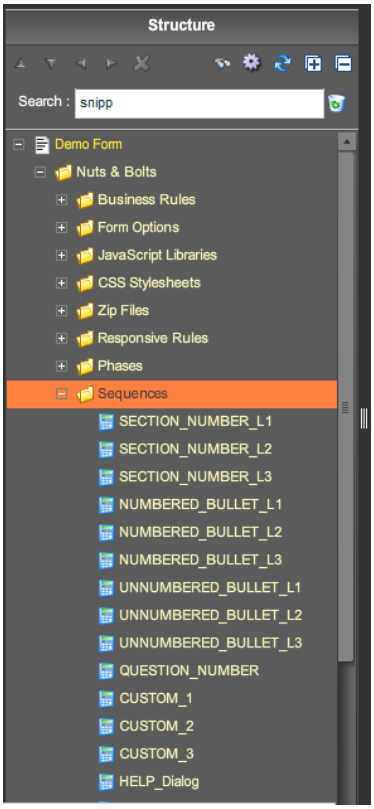
How to style bullet points in a Composer form

Unknown macro: 'redirect'

If you have certain requirements when adding text to forms for Terms and Conditions that requires Sub Numbering you can easily edit the Numbered Bullet points to match your styling requirements.

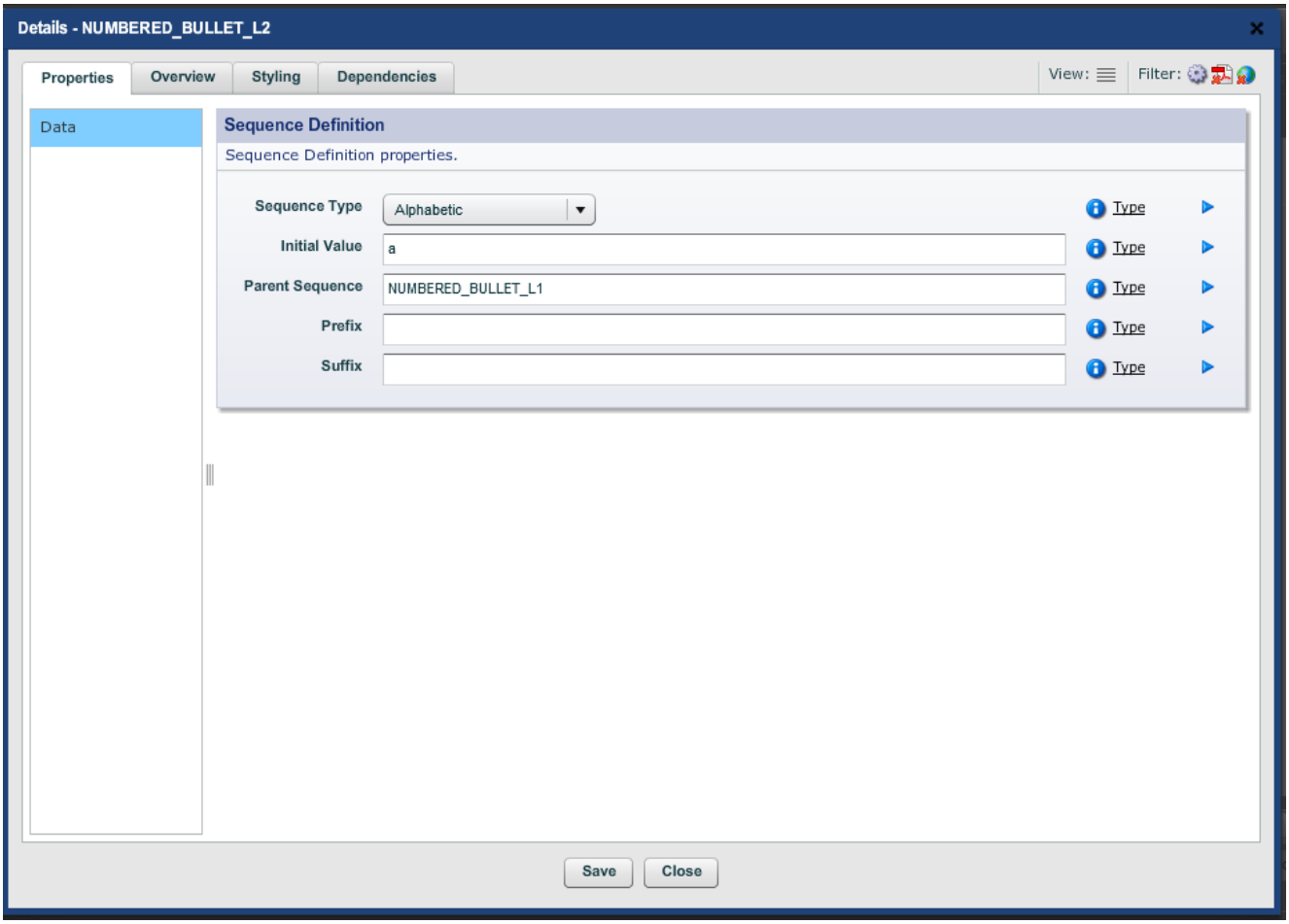
Steps to change your numbered bullets

1. Go to Nuts and Bolts > Sequences



2. Select the Bullet point you want to edit e.g. NUMBERED_BULLET_L2

Edit/change the properties and save



You can edit "Initial Value" and "Prefix" and "Suffix"

3. Click Save

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

Images in Composer forms

Unknown macro: 'redirect'

To add an image into a Composer form, drag and drop the 'Image [Resource] widget into the form, and select the image from the uploaded resources, or click the 'Upload Resource' button to upload a new image.

Composer supports images in PNG, GIF, and JPEG format.

The following forms demonstrate how different image formats behave in a rendered form:

HTML: https://tm.demo.avoka.com/finance/cookbook/images-html/
PDF: https://tm.demo.avoka.com/finance/cookbook/images-pdf/

Transparent Images

PNG alpha transparency (i.e. an extra colour channel for transparency per pixel) is not supported in PDF, however an indexed-mode PNG with a specific colour in its palette set to transparent **is** supported in PDF (and will probably look better than GIF).

PDF forms do not support transparent GIF files, and when the images are rendered, the transparent sections of the image will show black.

Setting an image logo using the standard template

The Standard Form template uses an image resource with a default 300x100 pixel image. It may be better to use 900x300 images so that when scaled down they remain sharp but still look good in other form factors.

1. Acquire image and load into graphic tool (gimp)
2. Set Image Mode to Indexed – 256 colours
3. Scale image width or height to nearest multiple of 900 or 300 respectively so that the other dimension doesn't exceed its limit.
4. Now size the other dimension without scaling to its max value and centre the image. Image should now be 900x300 with a preserved aspect ratio.
5. Add a transparency channel to the image layer
6. Use the magic lasso to cut out the background colour leaving transparency channel
7. Export as PNG and upload to Composer

The image should be as close as possible to 900x300. Too small and the scaling creates pixilation artefacts. Too big and you can get funny colour quantisation effects.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)
- Addresses

Including Web Fonts in Composer forms

Unknown macro: 'redirect'

While Composer is yet to officially support Web Fonts, it is possible to use Web Fonts right now, if you're willing to write a small amount of custom CSS.

Matthew has already written an excellent article on this topic, most of which remains applicable. The key distinction between the approach taken in his article versus this article is the location of the static font resources.

Matthew's approach requires creating a custom TM portal resource for each file involved (which can be time-consuming if you wish to provide the font in several formats to maximise cross-browser compatibility), this article will discuss how to include the resources as part of the form, which requires less manual configuration, and is less prone to errors when publishing forms, as all resources are contained within the generated publish archive.

For this example, we will be using the Font Awesome icon font.

Including font resources

1. Download the latest release of Font Awesome from the official site. At the time of writing, you will end up with font-awesome-4.2.0.zip
2. Drag a "Zip File" widget onto your form from the palette, placing it in the appropriate place (Nuts & Bolts -> Zip Files). The name you set for the widget is irrelevant to its functioning, but for clarity's sake, we will name it "FontAwesome".
3. In the dialogue that opens upon adding the aforementioned widget, clear the "Unzip path" field, and select the archive you downloaded in step 1 for "Zip File".
4. Click Save

Importing Font Awesome's CSS

Font Awesome provides its own CSS file which defines the custom font and includes numerous helper classes to quickly apply icons to elements. A full listing of these classes can be found here.

1. Drag a "CSS Stylesheet" (**Note:** We cannot use "CSS Snippet", as CSS3 @import declarations must occur before any other types of declaration in their containing stylesheet - this constraint will not be satisfied if you use a "CSS Snippet" widget) onto the form, adding it under Nuts & Bolts -> Css Stylesheets.
2. Name the added CSS Stylesheet widget whatever you wish. Upload a file containing the CSS detailed in appendix A.
3. Click save

Using FA icons

1. For each widget (usually buttons) you wish to use a FA icon on, browse to "Custom Styling" in the widget's properties and add "fa" (without quotes) to the "Custom Classes [HTML]" property.
2. Copy the desired icon from the FA cheatsheet and paste it into the "Caption" field for the target widget. Please note that nothing will appear (as Composer's flex UI cannot render these characters). You can verify that this worked by looking at the form definition XML in a unicode-aware text editor.
   (**Note:** Ideally, we could simply add the appropriate fa-XXXX class to our field, rather than this rather hacky approach, unfortunately, this will not have the expected result due to the structure of the markup generated by Composer). This approach can be used if you are willing to use additional custom CSS, but this is beyond the scope of this article.
3. Assuming a standard "Basic Push Button" widget is being used, you may wish to change the button type to "Link Button", disable underline on the caption, and amend the "Label Width" so that the resulting button is square.
4. Preview the form.

Appendix A - CSS to import Font Awesome's stylesheet (Only required because the "CSS Stylesheet" widget is unable to reference external resources, like the "Javascript Library" widget can)

```
/* Import's FA's stylesheet which is extracted to this path from the zip file.
 * Note that this stylesheet will in turn load the appropriate font resource
 * for the browser being used */
@import url("../font-awesome-4.2.0/css/font-awesome.min.css");

/* Restores the default font inheritance behaviour for any child elements of an
 * element with the fa class applied */
#sfc-container .fa * {
    font: inherit;
}
```

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Collaborative form completion with anonymous users (form sharing)

- Addresses

How to publish forms in multiple brands

Unknown macro: 'redirect'

If you want to publish the same form in multiple brands then please follow the steps below

Compatibility

Since	
Deprecated	

Step-by-step guide

1. Export the organisation and keep as a back up in case you make a mistake.
2. Create the brands in composer and verify that your form looks how you want using the preview .. for this example they will be called BRAND1 and BRAND2
3. Select the organisation in composer and click on the templates tab
4. if you have a template for you organisation then open it up .. if you do not have a template then create a new Template based on Template-Maguire
5. In the template editor go into the XML source and you will see a node called variants, the default is

```
<variants>
 <variant name="html-receipt" label="HTML form with Default PDF
receipt" publishsuffix="" isdefault="true" technologies="html-
desktop,html-receipt"/>
 <variant name="html-pdf-receipt" label="HTML form with LiveCycle
Output receipt" publishsuffix="LiveCycle Receipt" isdefault="false"
technologies="html-desktop,pdf-receipt"/>
</variants>
```

6. Update this section to add in the new variants below ensuring you change words BRAND1 & BRAND2 to your brands as they are labelled in the custom types tab of your organisation

```
        <variant name="BRAND1" label="BRAND1" technologies="html-
desktop,html-receipt">
            <stylesheet name="Brand-BRAND1" category="Brand"/>
        </variant>
        <variant name="BRAND2" label="BRAND2" technologies="html-
desktop,html-receipt">
            <stylesheet name="Brand-BRAND2" category="Brand"/>
        </variant>
```

7. Save the template and now update your form to use the new template.
8. When you publish you will now be able to select the brands you wish to publish on

Related articles

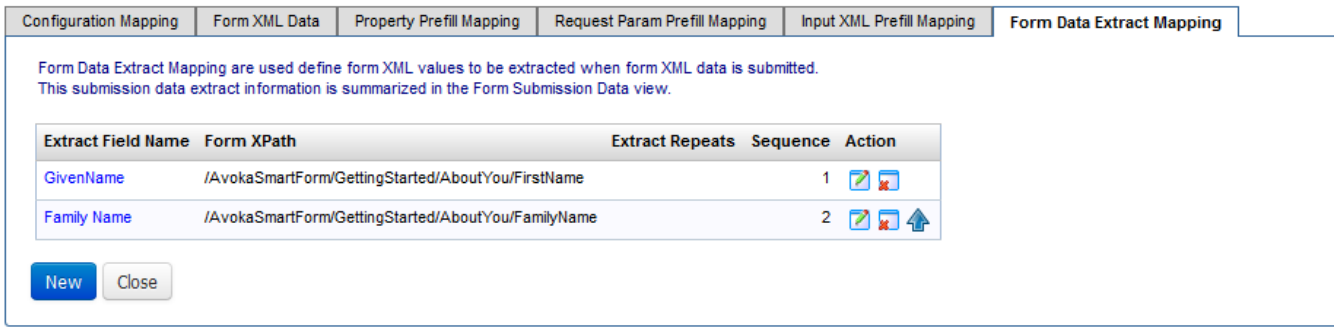
For more information on creating brands please look at the article Composer Style sheets

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Collaborative form completion with anonymous users (form sharing)
- Addresses
- Address Block Samples

Submission Data Extracts

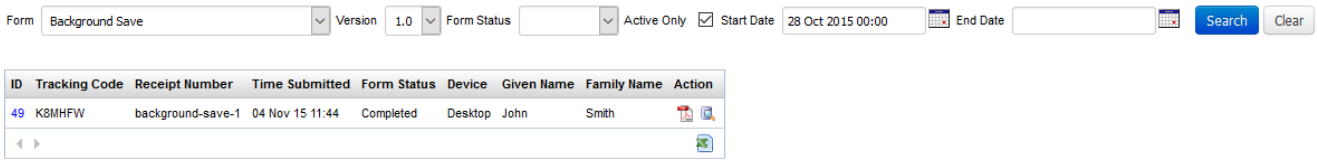
Unknown macro: 'redirect'

In the Form version's Data Config, Submission Data Extract mappings can be created which specify values that should be parsed out of Form Submissions and stored separately.



Submission Data Extract mappings

Submission Data can be viewed under Operations > Form Submission Data. The extracted values are displayed in a table against the Submission Id of the selected Form.



Form Submission Data

Email Delivery

Submission Data Extracts can be used in the message or subject of Email Deliveries configured in an Organisation's Form Delivery Details.
It can be used by using the Velocity variable *${submission.dataExtractMap.extractname}*, replacing "extractname" with the submission data extract name.

Displaying Submission Data Extracts in Space (Portal)

Submission Data Extracts can be used in the Space filters by default (exact string matches only) but you can also display them using something like:-

```
#if (${item.submission.dataExtractMap.EntName})
 <div class="span8">
   Entity Name: <b>${item.submission.dataExtractMap.EntName}</b>
 </div>
#else
 <div class="span8">
   Entity Name: <b>Unknown</b>
 </div>
#end
```

Related articles

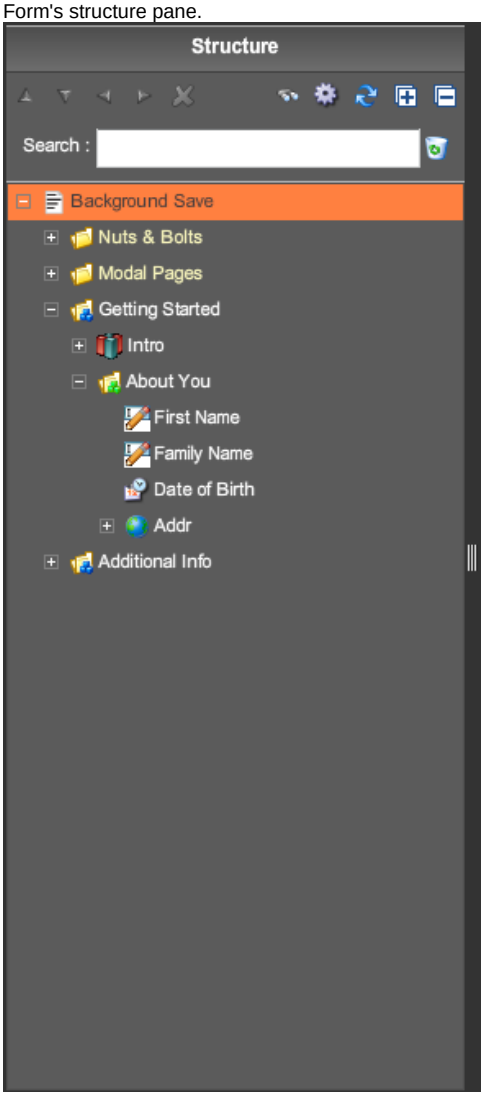
- How to pre-fill static map images into TransactField Tasks
- Determining the IP address that a form is requested from
- Repeating submission data extracts V4.1
- How to automatically extract the email address from a form and update the confirmation page
- How to define Form Submission Data extract fields in Composer V4

How to define Form Submission Data extract fields in Composer V4

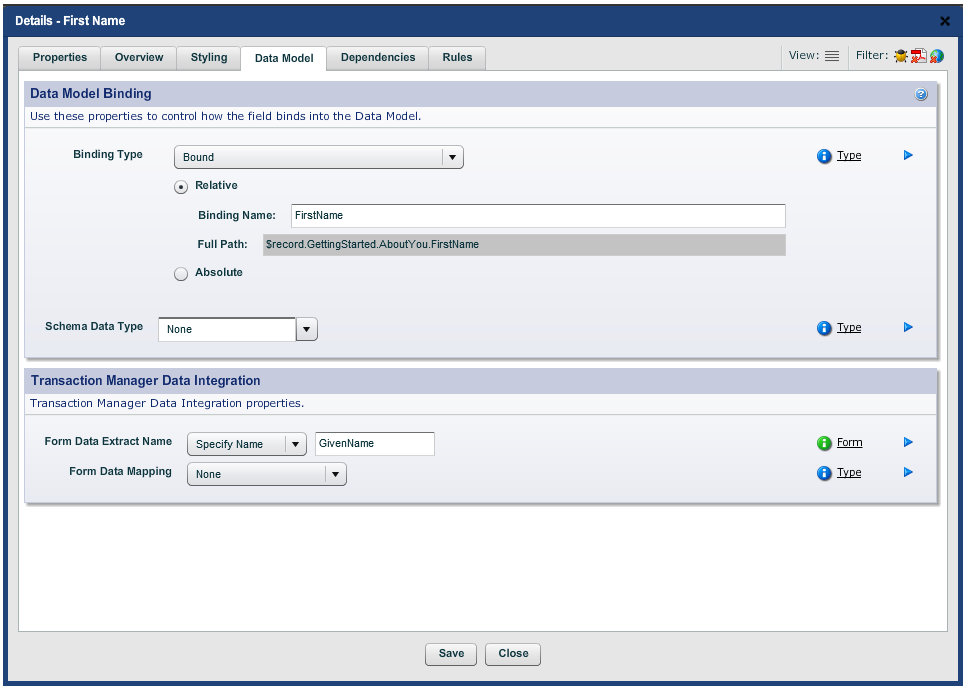
Unknown macro: 'redirect'

Composer

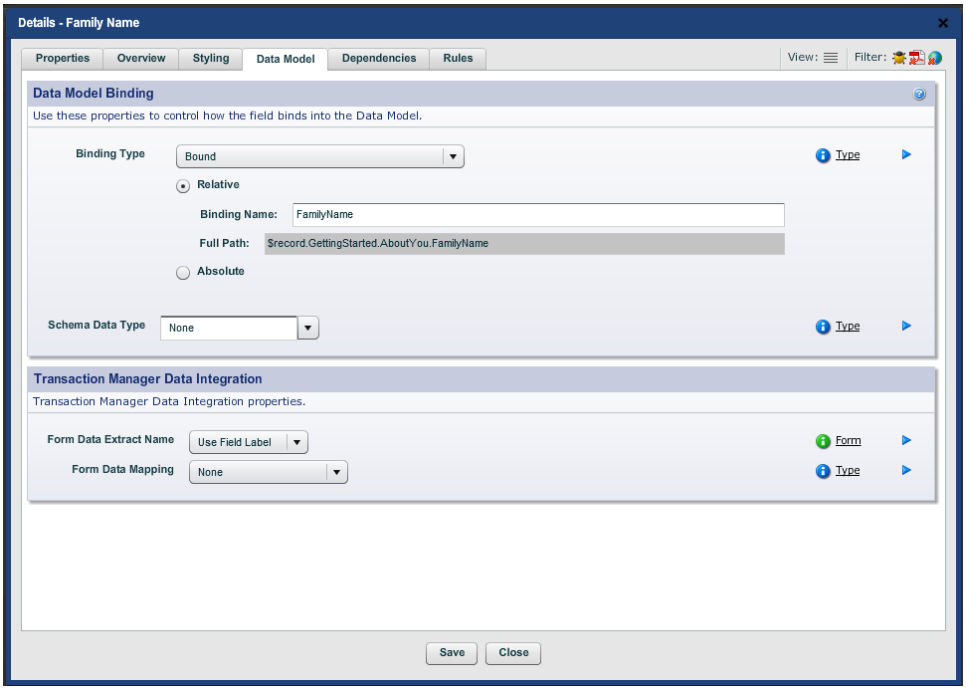
- Form's structure pane.



- Open the 'Property editor > Data Model > Transaction Manager Data Integration' section for the field you want to extract data from and save in Transaction Manager.



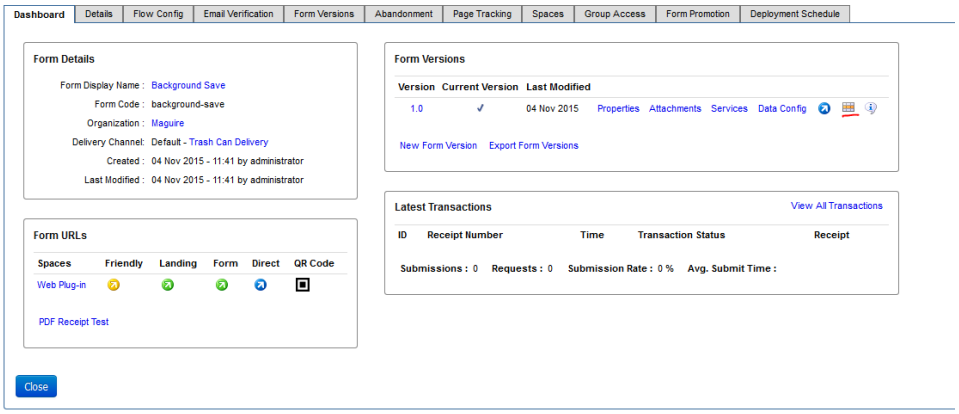
In the screenshot above, I have chosen a TM Data Extract Name of 'GivenName' for the 'First Name' field. It is safer to specify the TM Data Extract name than to use the field label (see screenshot below where I have used the field's label i.e. Family Name), because if you modify the label at a later stage, any post submission processes that use the data extract will fail.



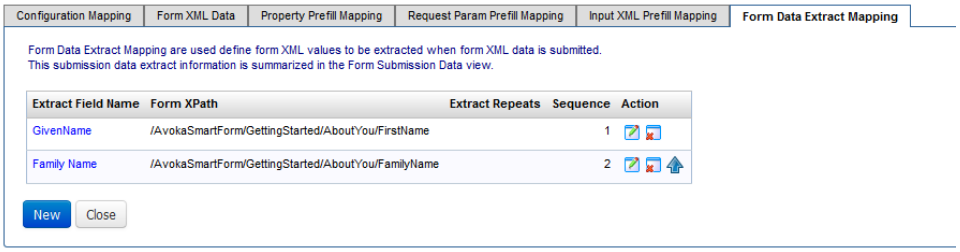
- Publish the form to Transaction Manager.

Transaction Manager

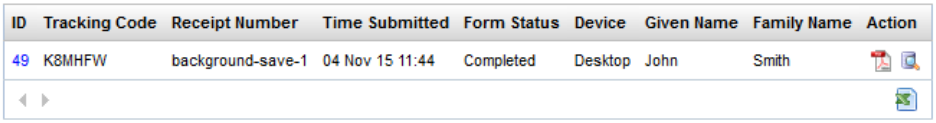
- The 'Form Submission Data' icon is visible on the form dashboard (initially blank).



- Select the 'Data Config' button (under 'Form Versions') > Form Data Extract Mapping' to see the XPath.



- Submit the form.
- Select the 'Form Submission Data' icon to see the data extracted.



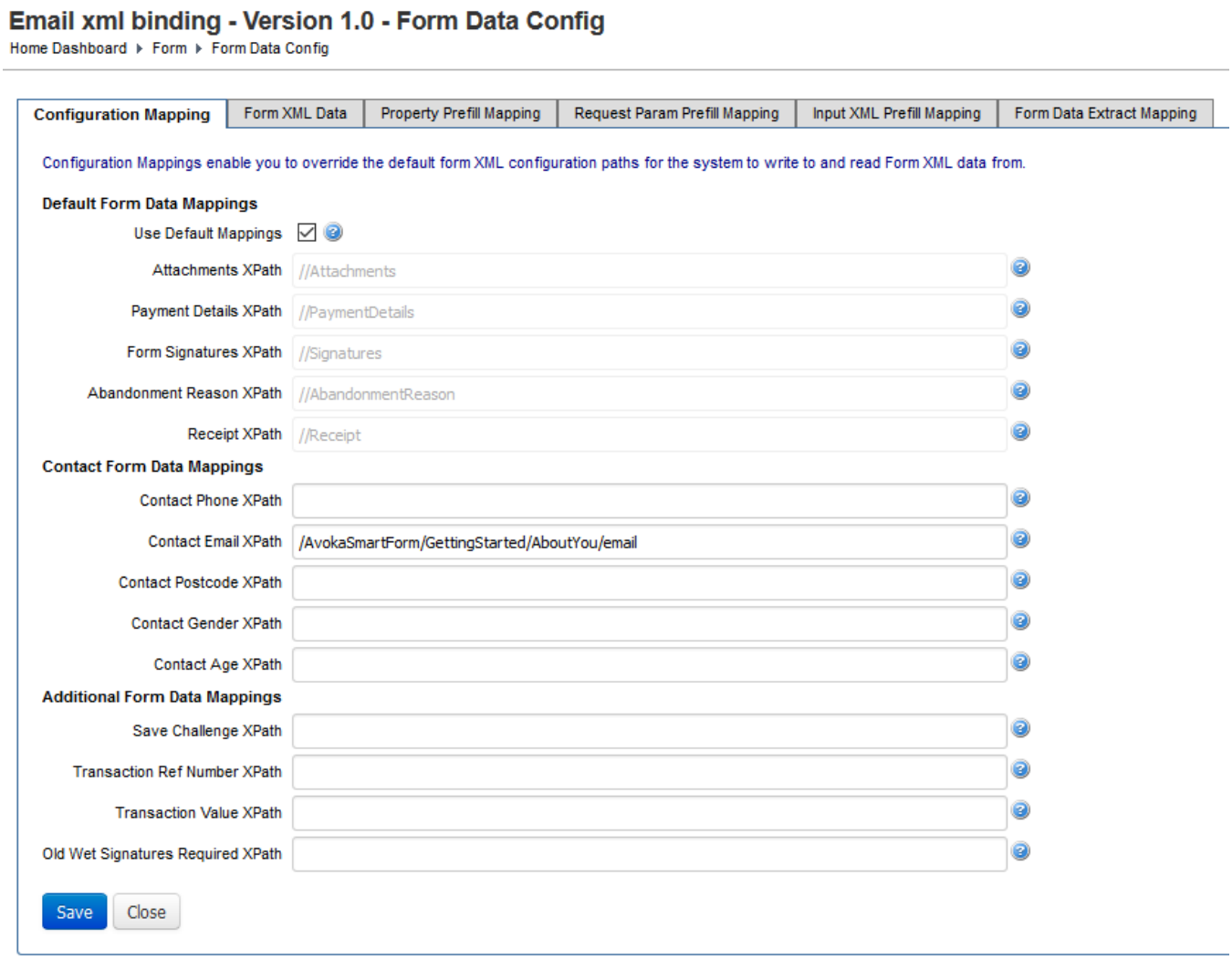
Related articles

- How to pre-fill static map images into TransactField Tasks
- Determining the IP address that a form is requested from
- Repeating submission data extracts V4.1
- How to automatically extract the email address from a form and update the confirmation page
- How to define Form Submission Data extract fields in Composer V4

How to automatically extract the email address from a form and update the confirmation page

Unknown macro: 'redirect'

To set up automated email extraction for a form, edit the form data configuration for the current form version and enter the XPath where the email address will be stored into the "Contact Email XPath" field.



Whenever a form is submitted or saved, the content of the specified element is read from the submitted data and stored against the submission. This email address is used by Transaction Manager during submission email verification and can also be used by custom services (e.g. Groovy delivery services).

To update the email field on the Confirmation page to automatically include the Contact email address

1. Open the form version where you want to implement the change.
2. Select the Data Config > Configuration Mapping tab.
3. Update the 'Contact Email Xpath' property with the email field element. Note: if you want to verify the XPath, use the 'Form XML Data' tab.

To reference the contact email address on the confirmation page for a form version

1. Include the HTML Confirmation Page form property because I don't want to change the confirmation page for all forms in the portal.
2. Find the code "<label class="" for="">Your Email Address</label>" and update the code as follows:

Code

```
<label class="" for="">Your Email Address</label>
                $emailForm.fields.emailField
                    <script> $( document ).ready(function() {
                      $('#emailForm_emailField').val("${submission.contactEmailAddress}");
                      });
                  </script>
```

3. **NOTE: the code above will only work if the email field is not blank.**
4. Save your change.
5. Test your form.

Related articles

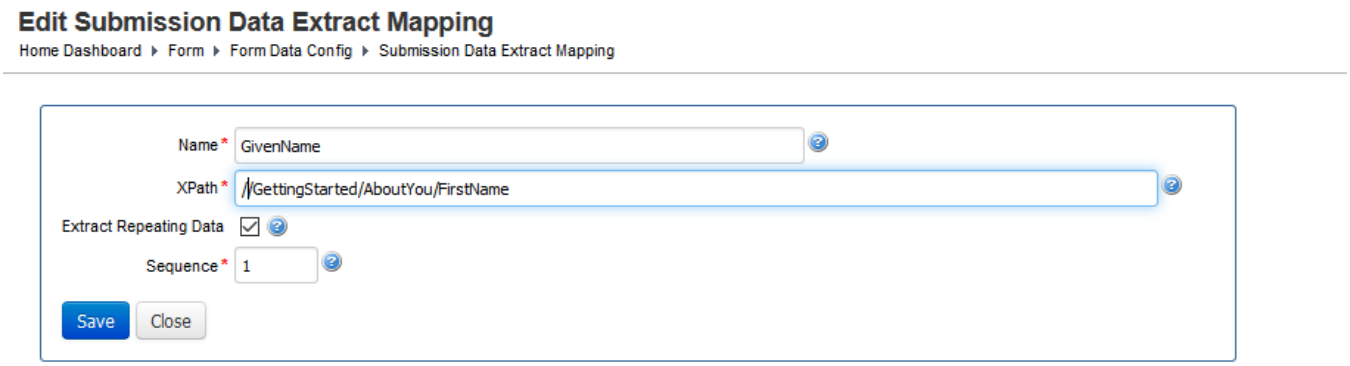
- How to pre-fill static map images into TransactField Tasks
- Determining the IP address that a form is requested from
- Repeating submission data extracts V4.1
- How to automatically extract the email address from a form and update the confirmation page
- How to define Form Submission Data extract fields in Composer V4

Repeating submission data extracts V4.1

Unknown macro: 'redirect'

Transaction Manager v4.1 introduces repeating functionality for submission data extracts.

To use this functionality, configure the submission data extract mapping in 'Form Data Config' -> 'Submission Data Extracts'



Note that the XPath provided must include a static node: i.e. if in the above example the 'Fields' node is the repeating node, providing the XPath //Fields /Firstname will only find the first instance of fields. To find all instances you must include at a minimum the static 'Section' node in the XPath, ie //Section /Fields/Firstname.

To include all instances, you must also ensure the 'Extract Repeating Data' option is checked.

Accessing the extracted data

Whenever the 'Extract Repeating Data' option is checked, the extracted data name will have an index appended to the extract name. For the above example, the first instance of 'firstName' will be stored under the submission data extract name 'firstName_1', the second will be stored under 'firstName_2' etc.

This enables the extracted data to be accessed in the normal way in groovy services:

```
def dataExtractMap = submission.getFormDataMap()

def firstName1 = dataExtractMap.firstName_1
def firstName2 = dataExtractMap.firstName_2
```

It is also possible to loop through the extracted repeating data using something like:

```
for (def i = 1; dataExtractMap.get("firstName_"+i); i ++) {
    def suffix = "_"+i
    def name = dataExtractMap.get("firstName"+suffix)
    //do something with name
}
```

Related articles

- How to pre-fill static map images into TransactField Tasks
- Determining the IP address that a form is requested from
- Repeating submission data extracts V4.1
- How to automatically extract the email address from a form and update the confirmation page
- How to define Form Submission Data extract fields in Composer V4

Widgets

Unknown macro: 'redirect'

- Address Block Samples
- Addresses
- Cascading dropdown list
- Hidden fields
- How to create a calculated field
- How to create and maintain a Table
- Populating Dropdown Lists
- Same as Above...Pattern
- Selections (One option from many)
- Static Dropdown list
- Tabular list from dynamic data

Address Block Samples

Unknown macro: 'redirect'

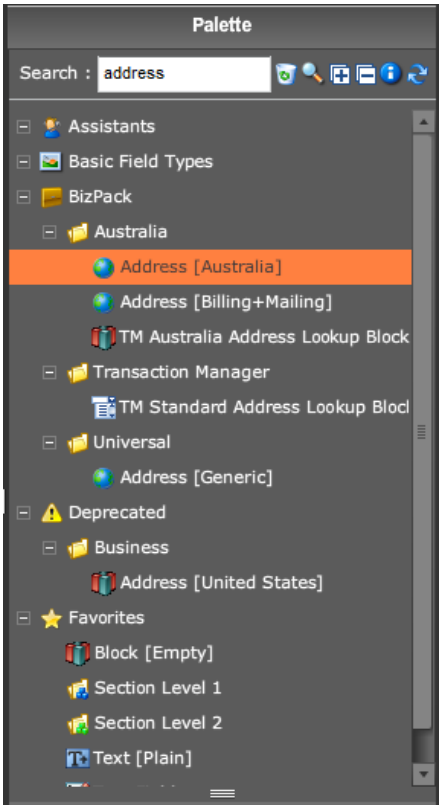
These samples will demonstrate how to create a simple address block with all the common functionality found in forms.

In Composer there are several pre-built widgets that will assist you in creating your forms. Here are a few that are located in the palette for creating an address block.

Address [Australia]

Address [billing + Mailing]

Address [United States]

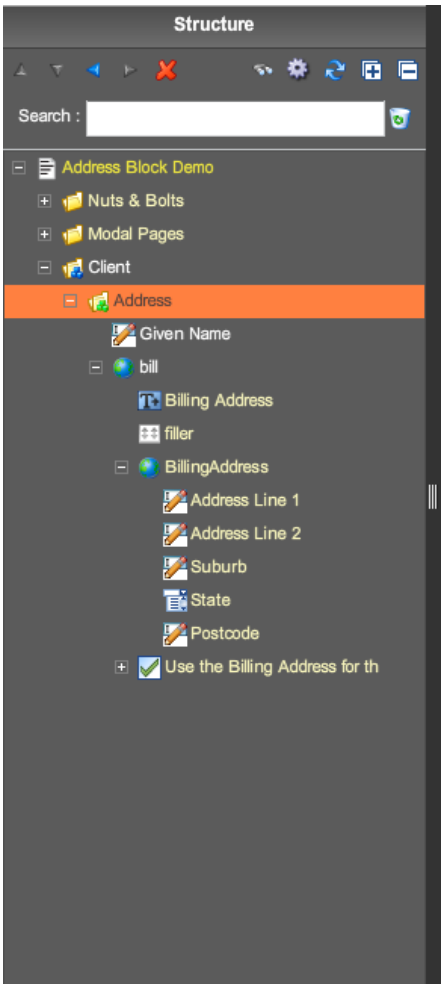


The widget: Address [Billing + Mailing], allows you to have your billing address and mailing address with a copy to check box.

This widget is very helpful. It is also important to note that the functionality can be modified but the layout cannot be modified.

Since this is a pre-existing widget you will notice the fields cannot be deleted. (yellow text in the hierarchy confirms locked fields)

Locked fields with in a widget:

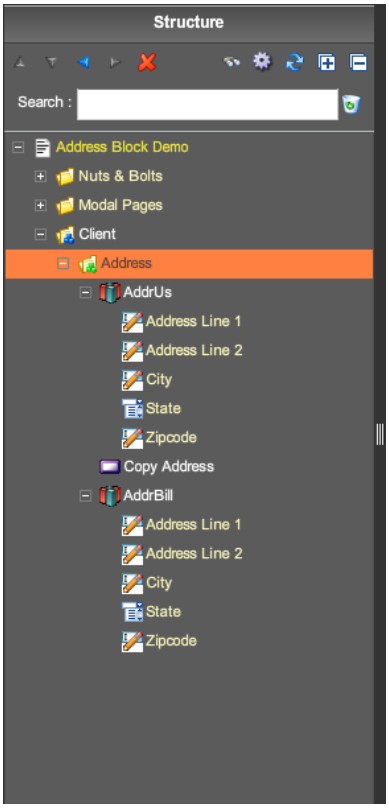


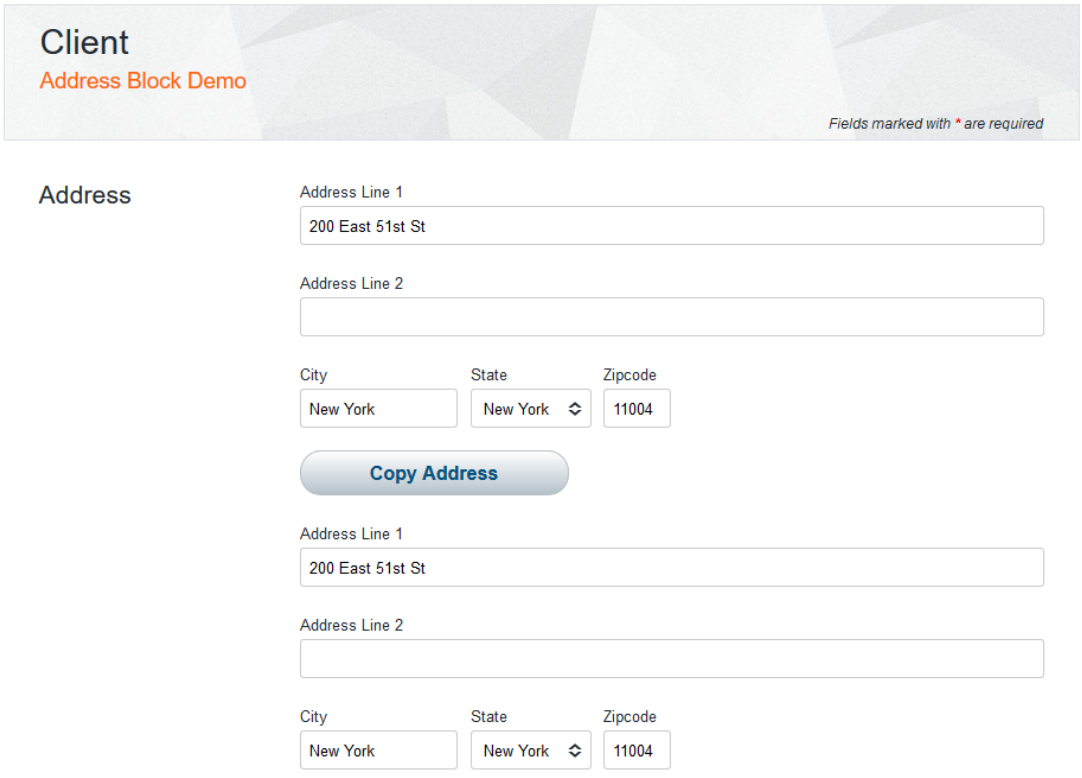
The script for this widget is written using a general business rule that only copies the fields over when checked.

```
if({_cbInternal}=="true"){
sfc.copyValues({BillingAddress}, {addressBlock});
}
```

One thing to notice about this widget is that if the user makes changes to any of the fields after checking the check box, it does not update the fields below.

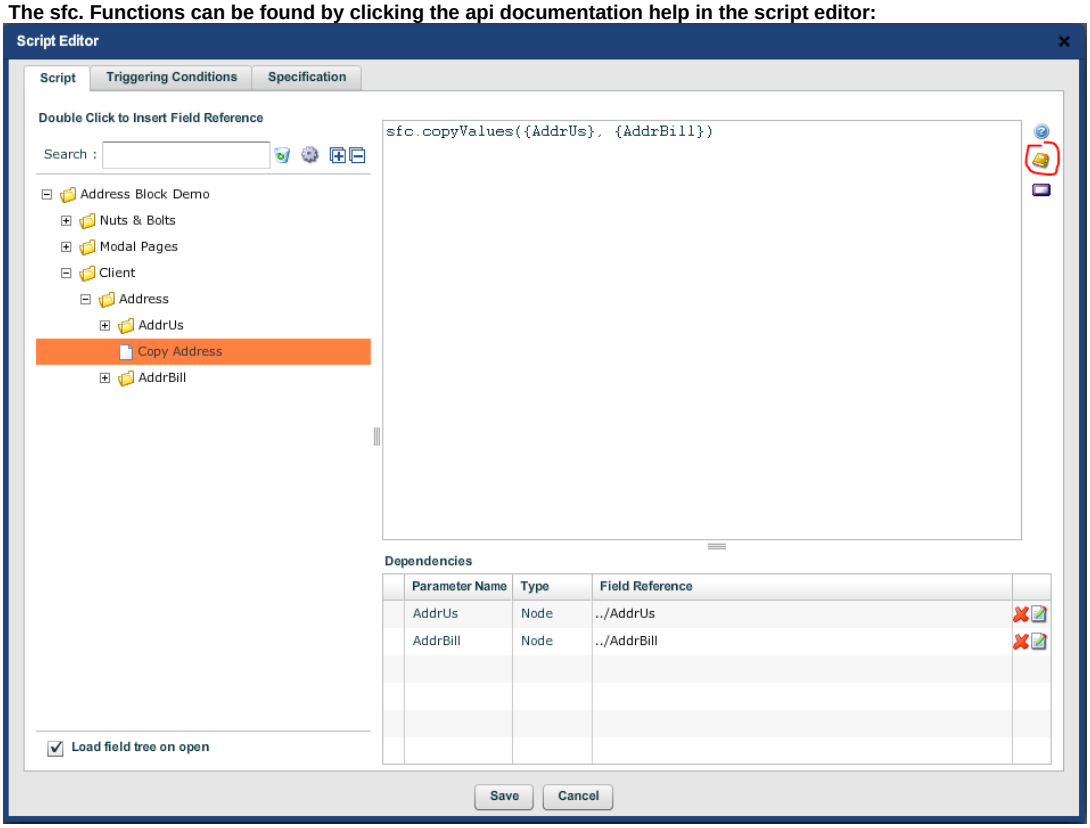
The next sample uses the Address [United States] and a button to copy the fields.





The concept here allows the user to change this slightly by using a button the same functionality applies.

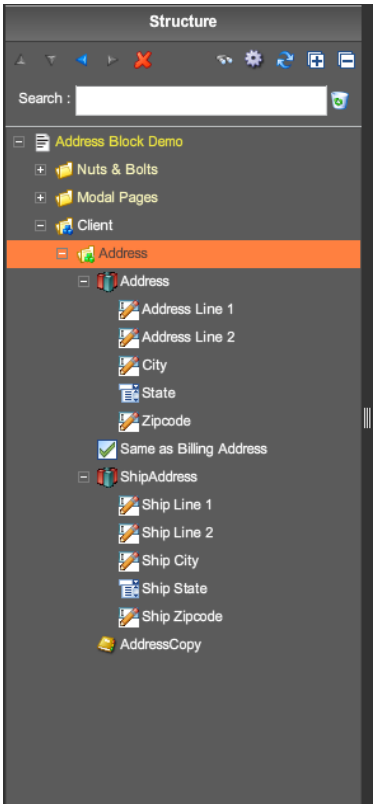
The sfc. Functions can be found by clicking the api documentation help in the script editor:



> *sfc.copyValues({Address1}, {Address2})*
>
> *This is using the sfc function: sfc.copyValues*
>
> ## *Method Summary*
> *Method AttributesMethod Name and Description*
>
> | | |
> |---|---|
> | *<static>* | *sfc.**copyValues**(srcBlockNodeRef, targetBlockNodeRef)*
Copies the values from the children of the source block to the same named children of the target block. |

Going a step further to create the same functionality but adding accessibility and accuracy by the users input, we can add script that will update the fields as they change.

The first example I will show you updates each field manually by inserting each field as a dependency in the script in general business rule. By adding each field as a dependency, any time the user makes a change in the field and then tabs or clicks out of the field the script will re-fire. For each field to be independent the fields are inserted as individual fields.



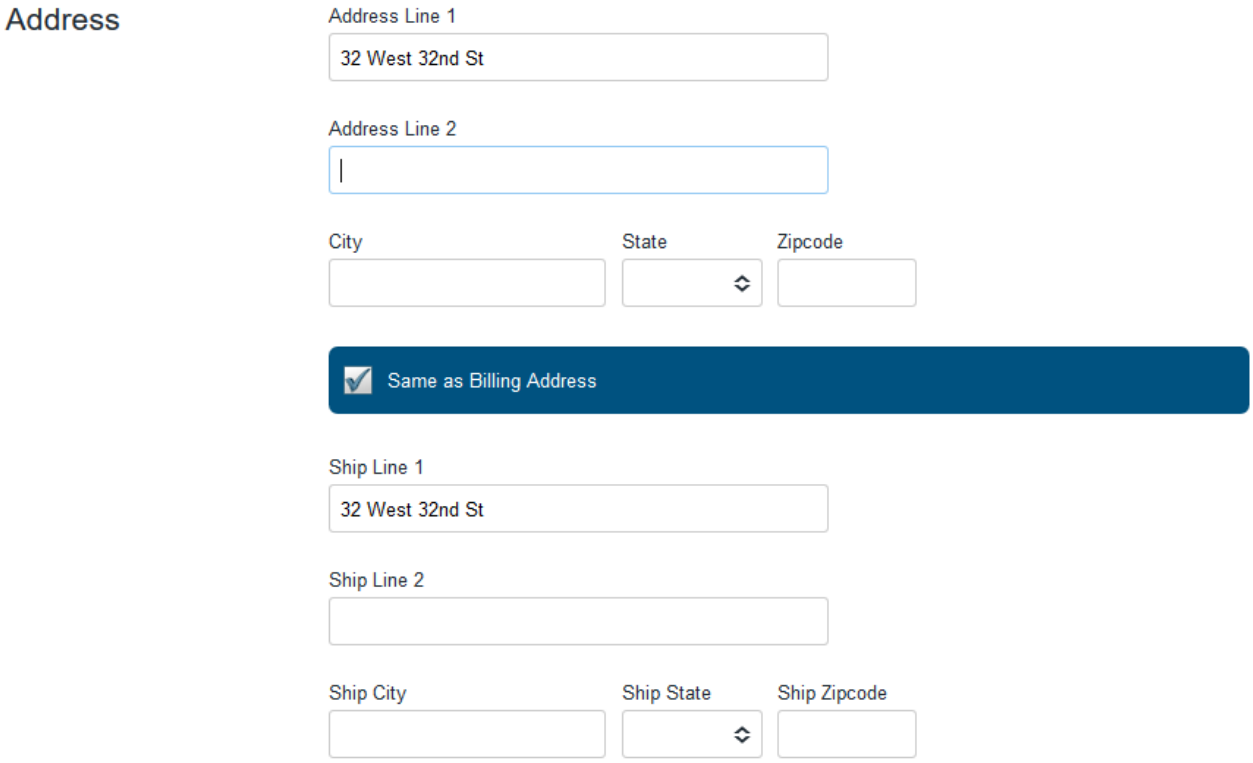
The script here is a little more dense because we are capturing each fields data in the first address block and setting each fields data in the second block manually using the functions; sfc.setRawValue and sfc.getRawValue.

This code manually gets and sets each field independently. This would be necessary if the blocks of fields did not have children with the same names.

```
if({SameasBillingAddress}=="true"){
sfc.setRawValue({ShipLine1}, sfc.getRawValue({MailingLine1}));
sfc.setRawValue({shipLine2}, sfc.getRawValue({MailingLine2}));
sfc.setRawValue({shipcity}, sfc.getRawValue({Mailingcity}));
sfc.setRawValue({shipstate}, sfc.getRawValue({Mailingstate}));
sfc.setRawValue({shipzipcode}, sfc.getRawValue({Mailingzipcode}));
}
else {
sfc.setRawValue({ShipLine1}, null);
sfc.setRawValue({shipLine2}, null);
sfc.setRawValue({shipcity}, null);
sfc.setRawValue({shipstate}, null);
sfc.setRawValue({shipzipcode}, null);
}
```

The last example emulates the same functionality but instead of combining our rules in a general business rule we set the rule individually on each field in the address block that we want to copy the first address blocks fields to by calculating the field values.

```
if({CopyAddress}=="true"){
return {addressLine11};
}
else {
return null;
}
```



Each of these examples is useful for different reasons and can be used to achieve the desired functionality.

Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

Addresses

Unknown macro: 'redirect'

There are many different ways to specify an address on a form. The appropriate pattern to use depends on your requirements.

Some of the factors to consider are:

- Do your addresses need to populate an existing database schema?
- Do your addresses need to populate an existing paper form for receipts?
- Billing / Mailing Addresses
- International Addresses

Billing / Mailing Addresses

A user of a form may have different addresses; one for delivery of goods, and another for billing. As such, it is often necessary to capture more than one address for a user however as it's possible the user will want to reuse the same address across these scenarios, allowing the user to reuse an already provided address within a form can help the user enter data quicker, and more accurately.
Composer comes with an *Address [Billing + Mailing]* Widget that caters for this common scenario. This provides a checkbox to the user, which then makes the second set of Address Fields disabled/read only. It is also common practice for the user to be able to see the address they previously entered in these read only fields.

Sample form: https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=addresses3

International Addresses

- Do you need to cater for international addresses, or can you assume a single country?
- If you do support international addresses, are they a minority or a common case?

Considerations also need to be made to the validations that you may be using for postal/zip codes depending on the user's country, and if there is use of dynamic drop downs such as states.

Select Country via Dropdown

A drop down with a list of countries can be provided to the user for them to select the country of their address. If it's expected that most user's will have their address in a particular country, this can be the defaulted value.
If a common country/s is known, it can be helpful to the user and reduce errors in data captured by validating the values provided by the user (such as zip /postal code format depending on country), or restricting the available States in a drop down list depending on their Country.

Pros

- Allows a common case country to be the defaulted value
- Can create additional field rules/validations based on the user's selected country
- Ensures the user selects a country that is valid i.e. prevents user entering Australia

Outside of Country option

If it's generally expected the user will have an address in a particular country, another option for corner cases is to provide a checkbox where the user can select they are outside of the particular country. This can then be used to prompt the user for the Country, and update any other validations/drop downs accordingly.

Pros

- Only prompts the user for a country when necessary
- Can set up fields, their rules and validations to suit the common case as best as possible

Cons

- If they are outside of the expected country, it becomes a two step process; Select the checkbox, THEN select/enter the country.

Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

Cascading dropdown list

Unknown macro: 'redirect'

Sometimes it may be necessary to change the values in a dropdown list based on a user's data entry. One way to solve this is to use a hidden text field which contains a calculation and a populating dropdown list which gets its value from the text field.

Sample form: https://composer.prod.avoka.com/composer/secure/composer.htm > Selections > Cascading Dropdown

https://tm.demo.avoka.com/finance/servlet/SmartForm.html?formCode=cascading-dropdown3

Using the 'Cascading Dropdown Manager' widget

Composer provides built-in Widgets which can be used automatically populate a series of Dropdown Lists based on a user's selections using prepopulated XML Data as a source.

- Cascading Dropdown Manager
- Cascading Dropdown List

Cascading Dropdown Manager

The Cascading Dropdown Manager manages the source XML Data as well as the Dropdown states.

Example XML

```
<root>
    <os name="Android" value="Android">
        <manufacturer name="HTC" value="HTC">
            <model name="One" value="One"/>
            <model name="One X" value="One X"/>
        </manufacturer>
        <manufacturer name="Samsung" value="Samsung">
            <model name="Galaxy S3" value="Galaxy S3"/>
            <model name="Galaxy S4" value="Galaxy S4"/>
        </manufacturer>
    </os>
    <os name="iOS" value="iOS">
        <manufacturer name="Apple" value="Apple">
            <model name="iPhone 5" value="iPhone 5"/>
        </manufacturer>
    </os>
</root>
```

In this example, there are three different tags which would three different cascading dropdown lists could reference (os, manufacturer, model).
The name attribute is used as the Dropdown label, while the value will become the Field's value if that particular option is selected. (name/value = label /value).
The XML can be provided to the Dropdown Manager directly or it can refer to XML data from a resource.

Cascading Dropdown List

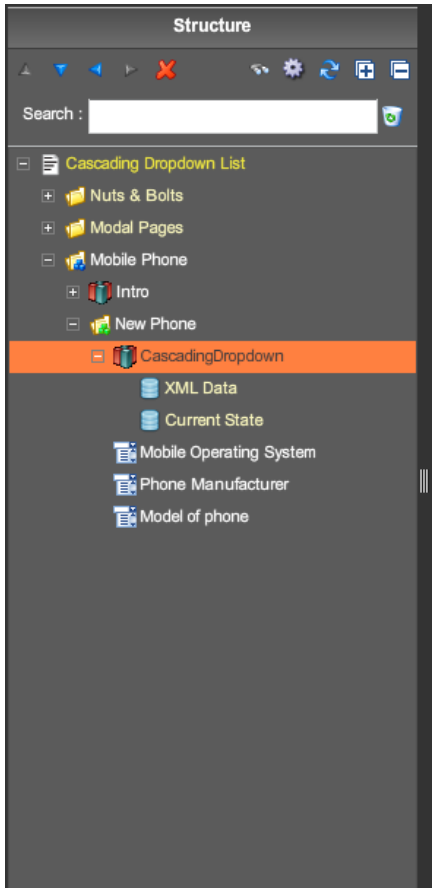
The Cascading Dropdown List widget provides the standard Dropdown List functionality, with extension to tie it to a Cascading Dropdown Manager via the Cascading Dropdown Controller reference.

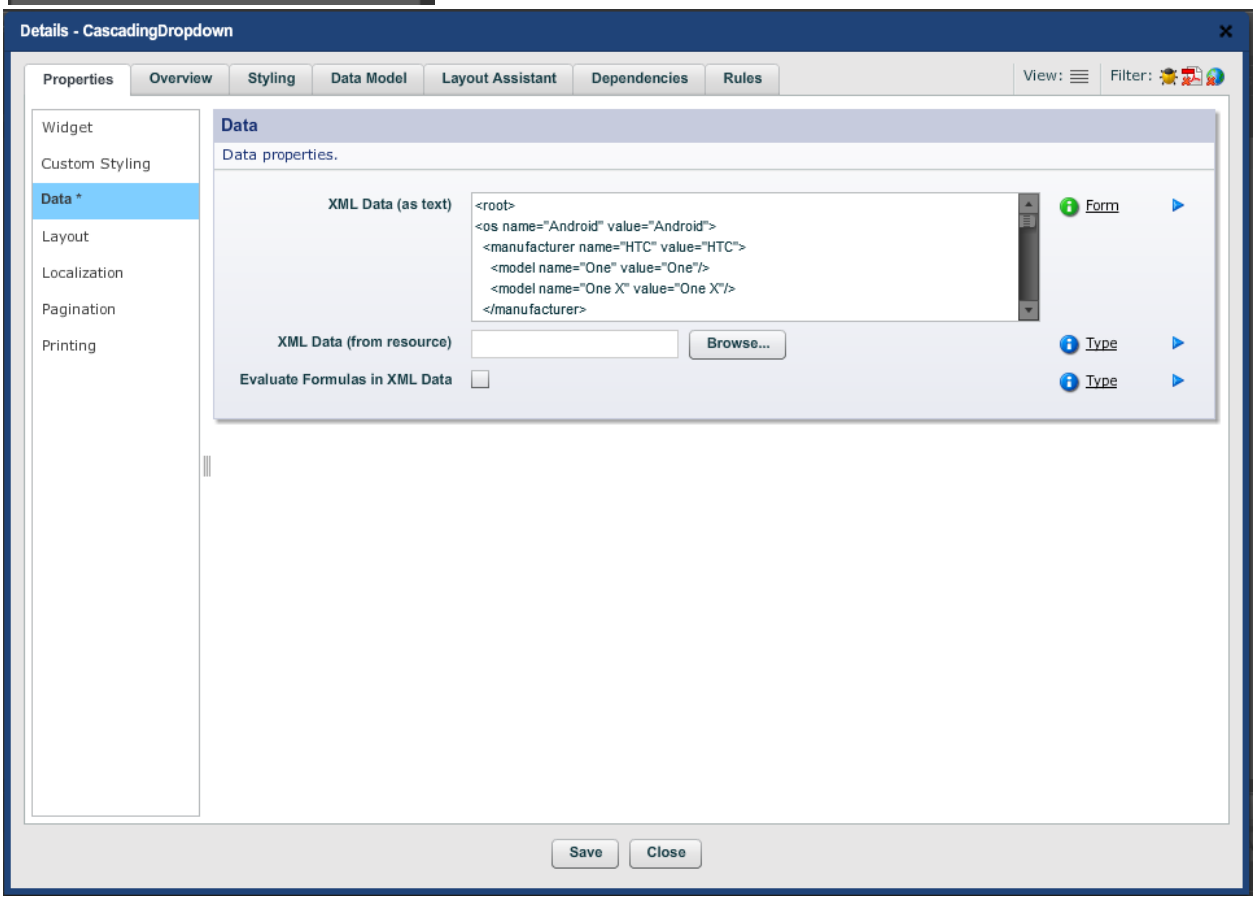
Each Cascading Dropdown List then has a data path to a tag within the Dropdown Manager's XML. To suit the example above, the data path for a model Dropdown List would be /os/manufacturer/model

Considerations

- Please consider usability if a large data set is used i.e. Dropdowns with very large lists.

Example Visually





Using API functions

The ‘sfc.clearListItems()’ and ‘sfc.addListItem()’ API functions can be used to populate dropdown lists manually, using javascript. The sample form contains an example using the following code:

```
var currentValue = sfc.getRawValue({Model});
sfc.clearListItems({Model});
if ({Make} == "Nissan") {
    sfc.addListItem({Model}, "Skyline", "Skyline");
    if (currentValue == "Maxima" || currentValue == "Skyline") {
        sfc.setRawValue({Model}, currentValue);
    }
} else if ({Make} == "Toyota") {
    sfc.addListItem({Model}, "Camery", "Camery");
    sfc.addListItem({Model}, "Corolla", "Corolla");
    if (currentValue == "Corolla" || currentValue == "Camery") {
        sfc.setRawValue({Model}, currentValue);
    }
}
```

The script is run from a ‘Business Rule (general purpose)’ widget. When using this method, it is important to check the selected value of ‘model’ before clearing the values, so that if a ‘make’ is selected of which the current value is valid, it can be re-set as the selected value. This is necessary because business rules will run when the value of any field that it references is changed, therefore this script will run when the ‘Model’ field is changed, and can produce erratic behavior.

https://composer.prod.avoka.com/composer/doc/jsdoc/symbols/sfc.html#.addListItem
https://composer.prod.avoka.com/composer/doc/jsdoc/symbols/sfc.html#.clearListItems

Using the ‘Populating Dropdown List’ widget

The ‘Populating Dropdown List’ links to a field, which tells it what values to display. The sample form contains an example of this, where the linked field has a calculate rule applied to it, which changes the available values dependent on the value of the first dropdown. The calculate script on the linked field is:

```
if ({Continent} == "Africa") {
    return "southAfrica|South Africa,zimbabwe|Zimbabwe";
} else if ({Continent} == "Europe") {
    return "england|England,france|France";
} else {
    return "";
}
```

Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

Hidden fields

Unknown macro: 'redirect'

The widgets in Composer can have their visibility set to hidden. Hidden fields continue to exist in the underlying data model of the form but aren't visible or editable by form respondents.

In addition to their visibility widgets also have a 'Visibility Data Clearing Policy' which dictates what should happen to any data contained within that field when it's hidden. If this policy is set to 'When Hiding – Don't Clear Data' the field can continue to be used for data storage and calculations.

Submit buttons also have a 'Clear Hidden Data on Submit' option which can be turned on or off. By default is it on, but if it's turned off hidden fields will have their data included in the form submission.

Data Field

A data field is a standard Composer widget, in essence it's a regular Text Field which is invisible by default and has its Data Clearing Policy set to 'Preserve'. It is intended to store (and submit) internal data rather than user-visible data. This can be a quick and easy way of storing the result of things like calculations without needing them to be directly visible to the form respondent.

Using hidden fields to simplify form logic

It's often the case that certain questions or groups of questions will trigger the hiding or displaying of subsequence questions or sections. For example you may display fields to collect someone's billing address only after they indicate that it's different from their shipping address.

If the visibility of several questions is dependent on the values of multiple preceding questions, writing and maintaining visibility rules on each of the fields can be a challenge. You may need to copy the visibility rule from one field to the next. When an update is required you would need to make sure it was applied to each field using the same rule.

Hidden fields can be used to simplify this situation. For example a hidden checkbox could be added to the form, this checkbox will have a calculation rule that switches its value from 'false' to 'true' when the preceding question contain the required values. The later questions can have their visibility rules all point to the checkbox, making them visible when it is 'true' and hidden when it's 'false'. If in future the visibility rule for these fields changes, you can simply change the calculation rule on the checkbox. (for those familiar with database design this serves a similar purpose to 'junction tables', resolving many-to-many relationships).

Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

How to create a calculated field

Unknown macro: 'redirect'

Fields can automatically be filled with a calculated value by adding a calculation script.

To create a calculated field:

Open the target field's property editor > Rules option -> Calculation Rule, select 'Script Based' and enter a script to calculate the value.

There is a form in Composer Demo > Cookbook for Clients > FormPatterns > that has an example of a 'Monthly Salary' field, which is calculated by dividing the user entered 'Yearly Salary' field by 12. The simplest way to do this would be:

return ({MonthySalary_1}/12);

However, depending on the number entered into the Yearly salary field, it can return numbers with too many decimal places. It can be rounded to 2 decimal places in the script:

return (Math.round({MonthySalary_1}/12)*100)/100);

This multiplies the number by 100, rounds it to the nearest integer, then divides by 100 to get 2 decimal places.

The example also has a checkbox providing the option to override the calculated value, and allow the user to enter their own value. For this to work, the script must check the value of the checkbox, and if it is true, retain the value entered into the field:

```
if ({OverrideMonthlySalar}) {
    return {MonthySalary_1};
} else {
    return (Math.round(({AnnualSalary}/12) * 100) / 100);
}
```

Sample form: https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=calculations2

Related articles

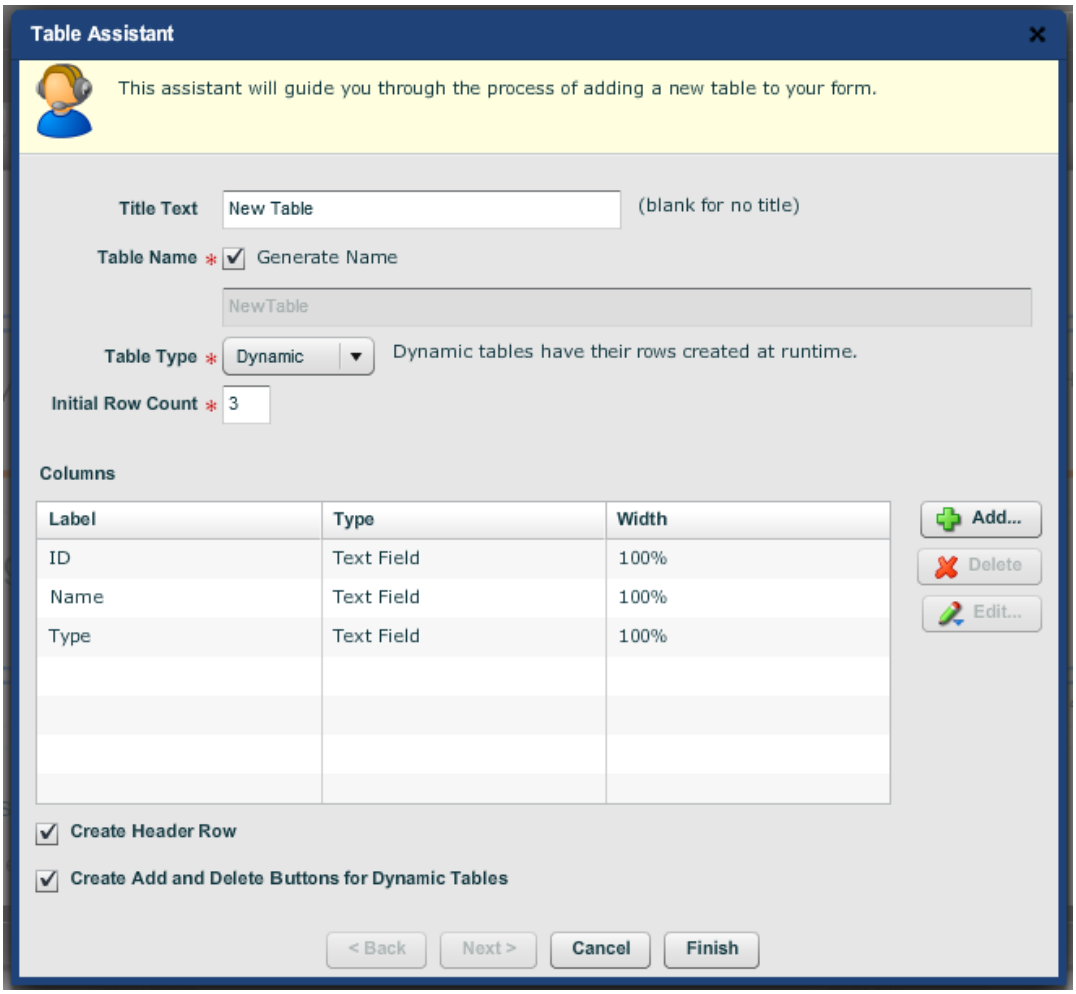
- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

How to create and maintain a Table

Unknown macro: 'redirect'

Creation

Creating a Repeat is easiest for Tables, and is discussed here. And the easiest way to setup basic Table structure is to use the Table Assistant from the Composer Palette->Assistants section.



The Assistant provides a simple way to create a Table object, complete with a containing Block, a Title, a Header showing the column labels and Row whose data type may be specified.

Note that the columns in the screenshot are all set to the same width – 100%. This is a simple way to create columns of equal width. The Layout Manager for the form render will assign space to each column proportionally based on the available width of the screen or containing blocks. The resulting table is shown below.

New Table

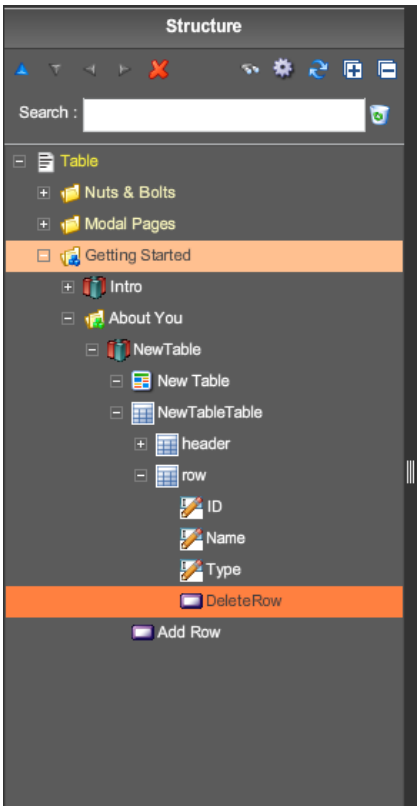
ID	Name	Type	
34	John Smith	User	

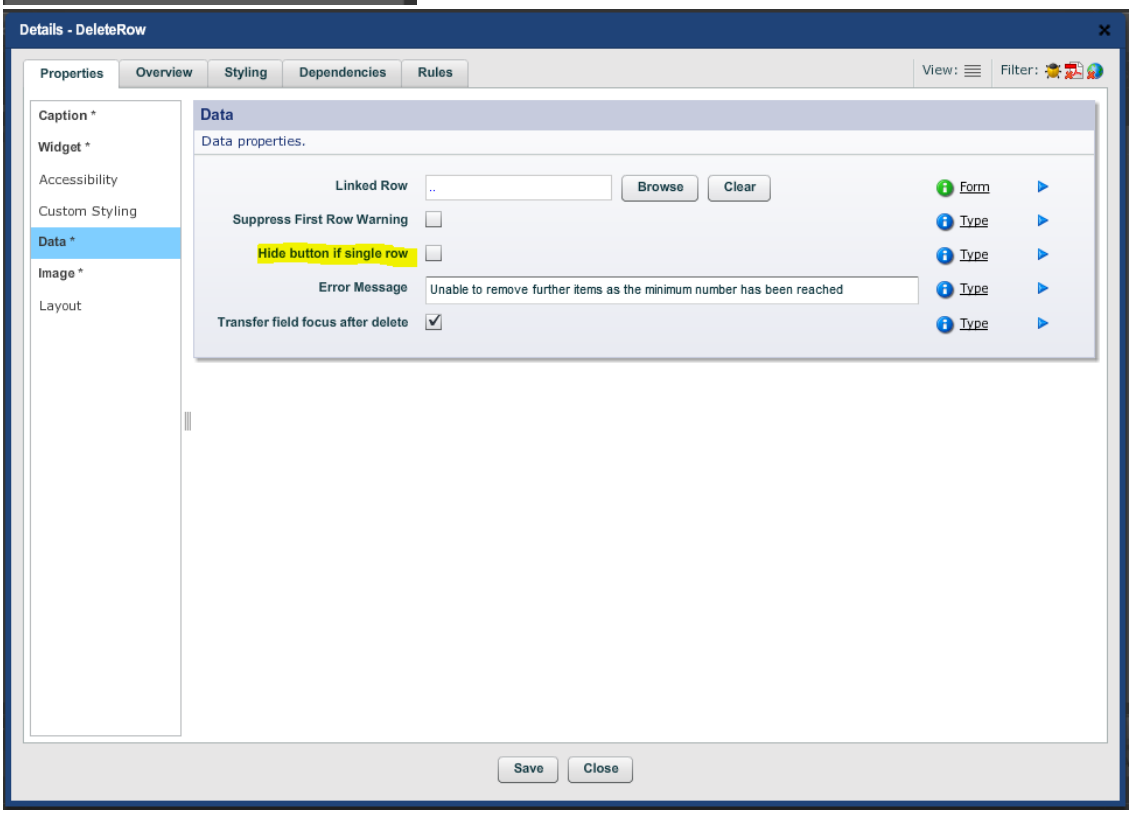
Add Row

Remove Delete Buttons for Single Instance

This is easily done, but it is hidden in a bit of an unexpected place.

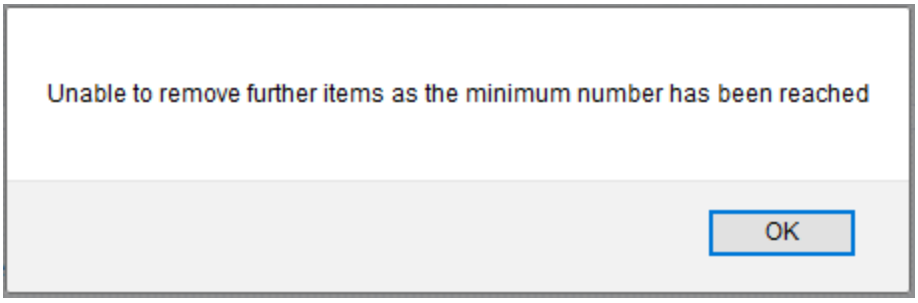
On the properties of the Delete Row button, on the Data panel, there is a checkbox for “Hide button if single row” – just make sure this is checked. The button will then be invisible when there is only one row in the repeat.





You do need to be sure this is the desired use case, and that the Delete Row button setting is in agreement with the Initial and Minimum count settings. Meaning, if the repeat should be able to have zero rows, the Delete Row button should display itself when there is only one row. Conversely, if the repeat must have at least one row, the Delete Row button should hide when there is only one row left.

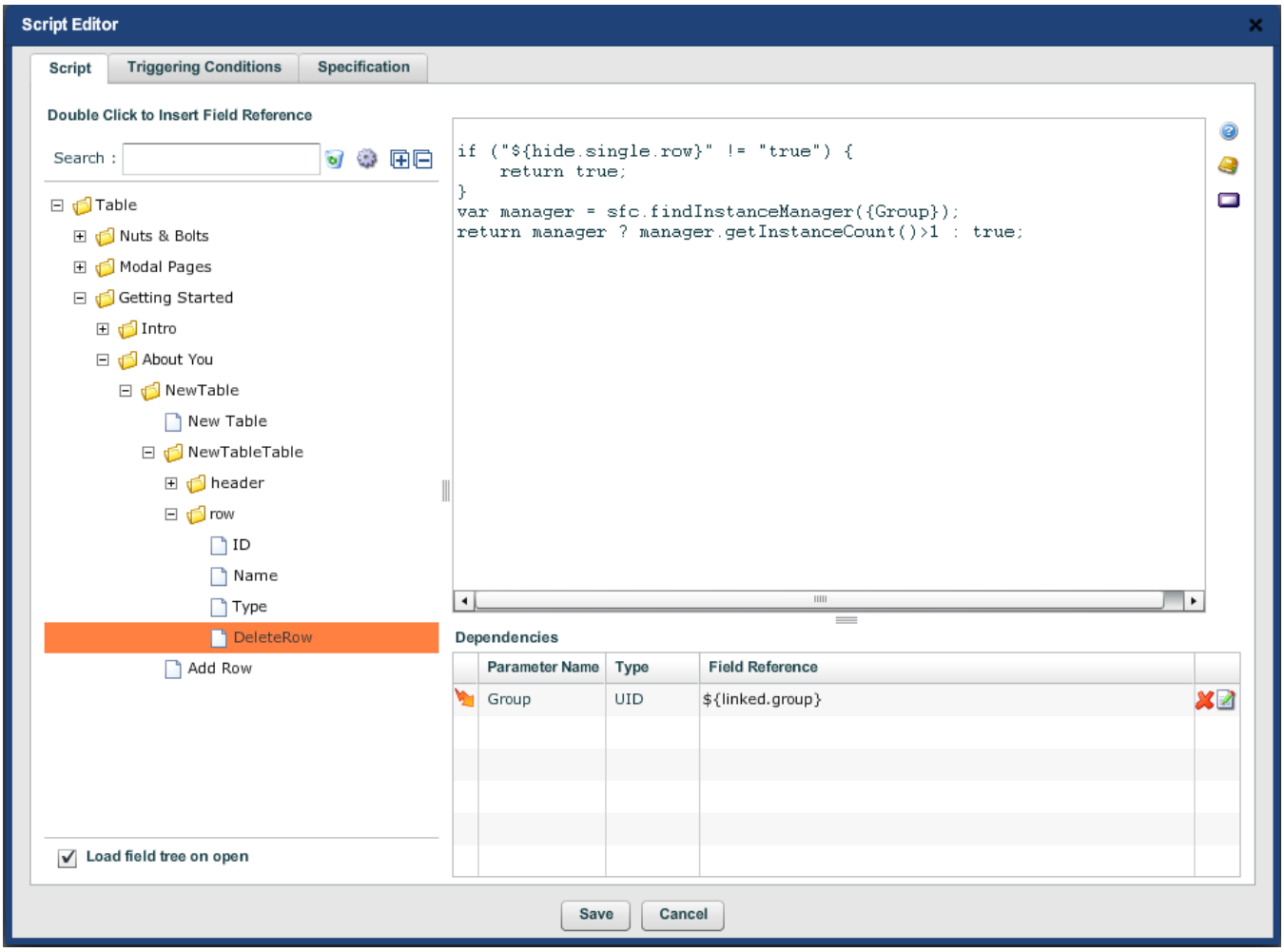
Should the Delete Row button be visible when there is only one row of data, and the Repeat properties indicate that there is a minimum of at least one row required, an error message will be displayed if the Delete Row button is clicked and the last row will not be removed. It is however Best Practice to not allow the error message to be displayed – Hide the button if it should not be clicked.



If the requirements indicate that the minimum rows in the repeat is greater than one, the visibility of the Delete Row button can be controlled to meet the requirement.

First, leave the “Hide button if single row” checkbox unchecked.

Second, change the Visibility Rule on the Rules panel by clicking the Edit button.



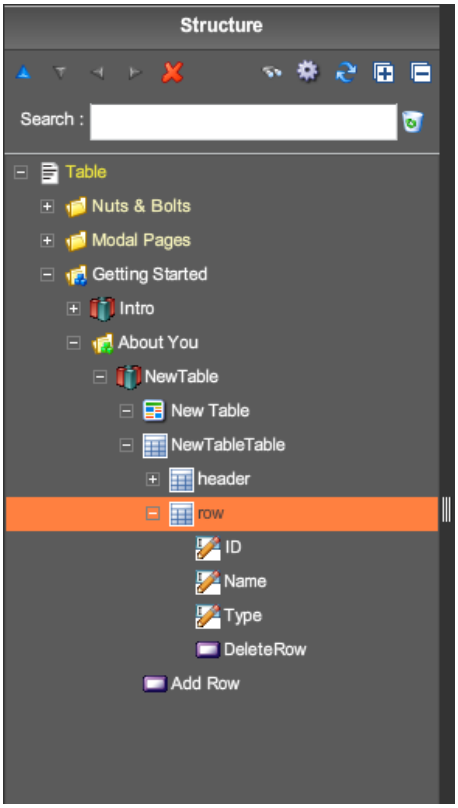
The Script as provided controls visibility for a minimum of one row scenario. Remove or comment out the “if” block – the value of the checkbox will not be used. Instead change the number “1” on the last line of script to match your minimum row requirement. For example:

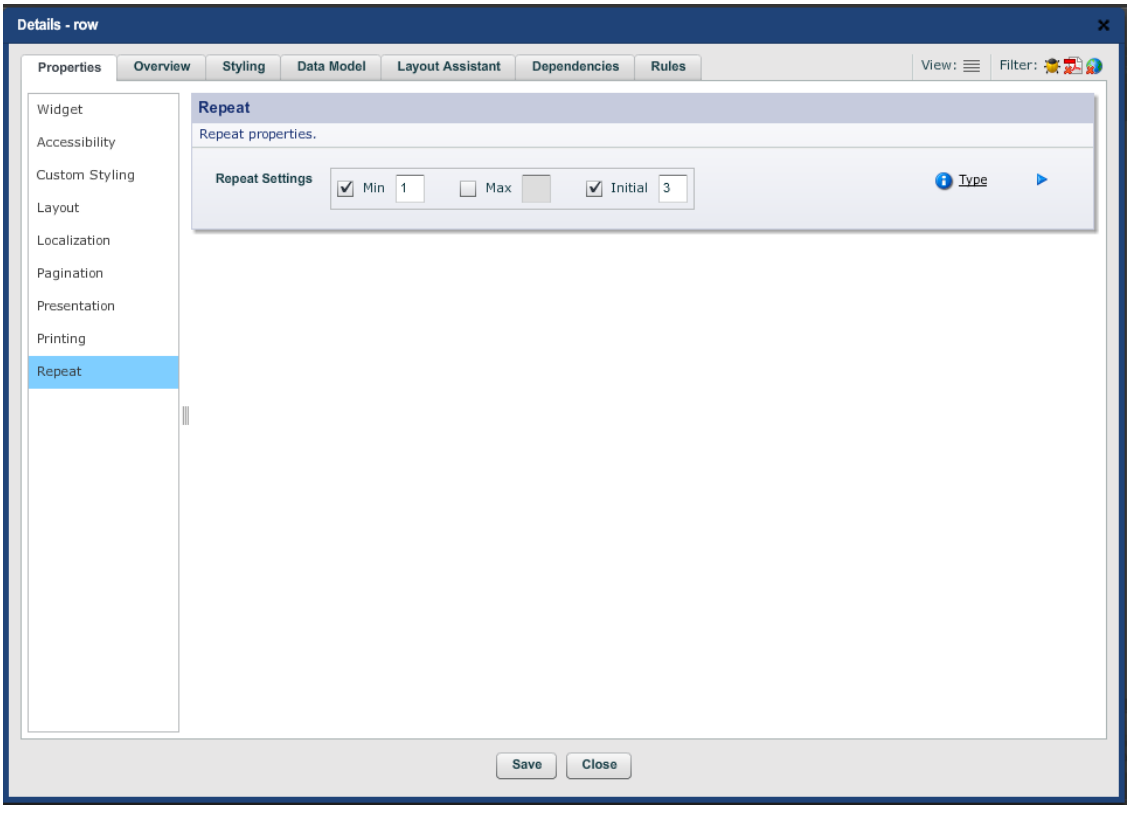
return manager ? manager.getInstanceCount()>3 : true;

would only allow the button to display if at least three rows exist.

Initial Count, Min, Max

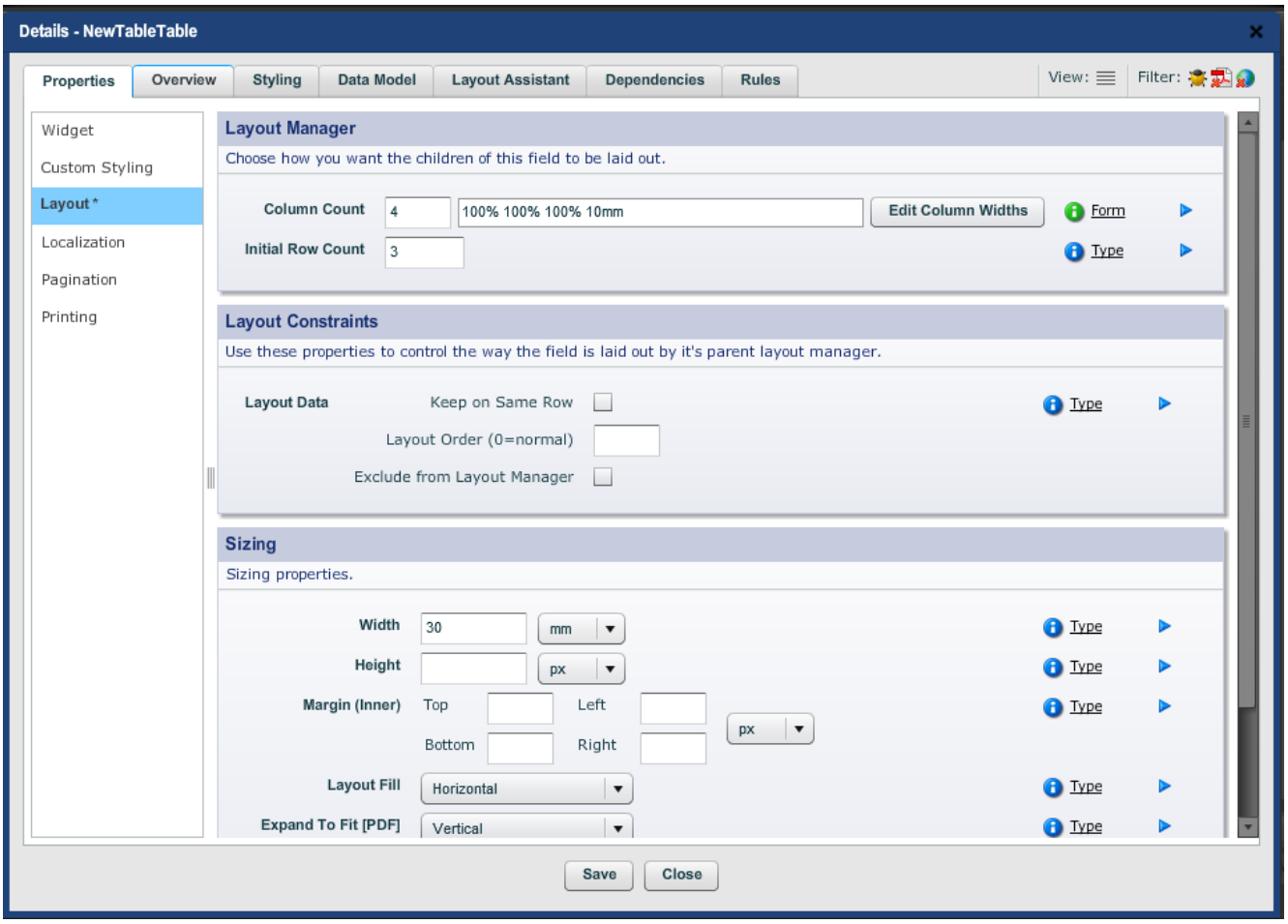
A repeating section can be limited and automatically kept within a certain minimum and maximum size, and the initial number of rows can be set as well. The Composer Table Assistant will let you set the Initial value, but Min and Max must be set in the Repeat Panel of the Row Properties. See below.





If a Max value is not supplied, the form will not impose a maximum limit on number of Rows. If a Min or Initial value is not supplied a zero is used.

The Initial number of rows MUST be set in two places. The first place is as shown above in the Table Row's Properties. The second is in the Properties of the containing Table object, as shown below.



Updating a table to include a new column

The example illustrates how to include a new column at the beginning of the table that will automatically show the row number.

1. In the property editor for the table widget (an example can be seen in the screenshot above), update the property 'Column Count'. The new column is always added to the end of the table so select the 'Edit Column Widths' button and move the Delete button i.e. 7mm to the end.
2. Add a Text [Plain] widget to the 'header' component to populate the column heading and move it to the top of the 'header'.
3. Add an 'Integer' widget and a 'Repeatable Block Index [Text Field]' widget as the first 2 elements in the 'row' component of the table. Note: the 'R epeatable Block Index [Text Field]' widget is never visible and is used to get the row number. However the index starts at 0.
4. So to number your table rows from one, add the following calculation script to the Integer widget:

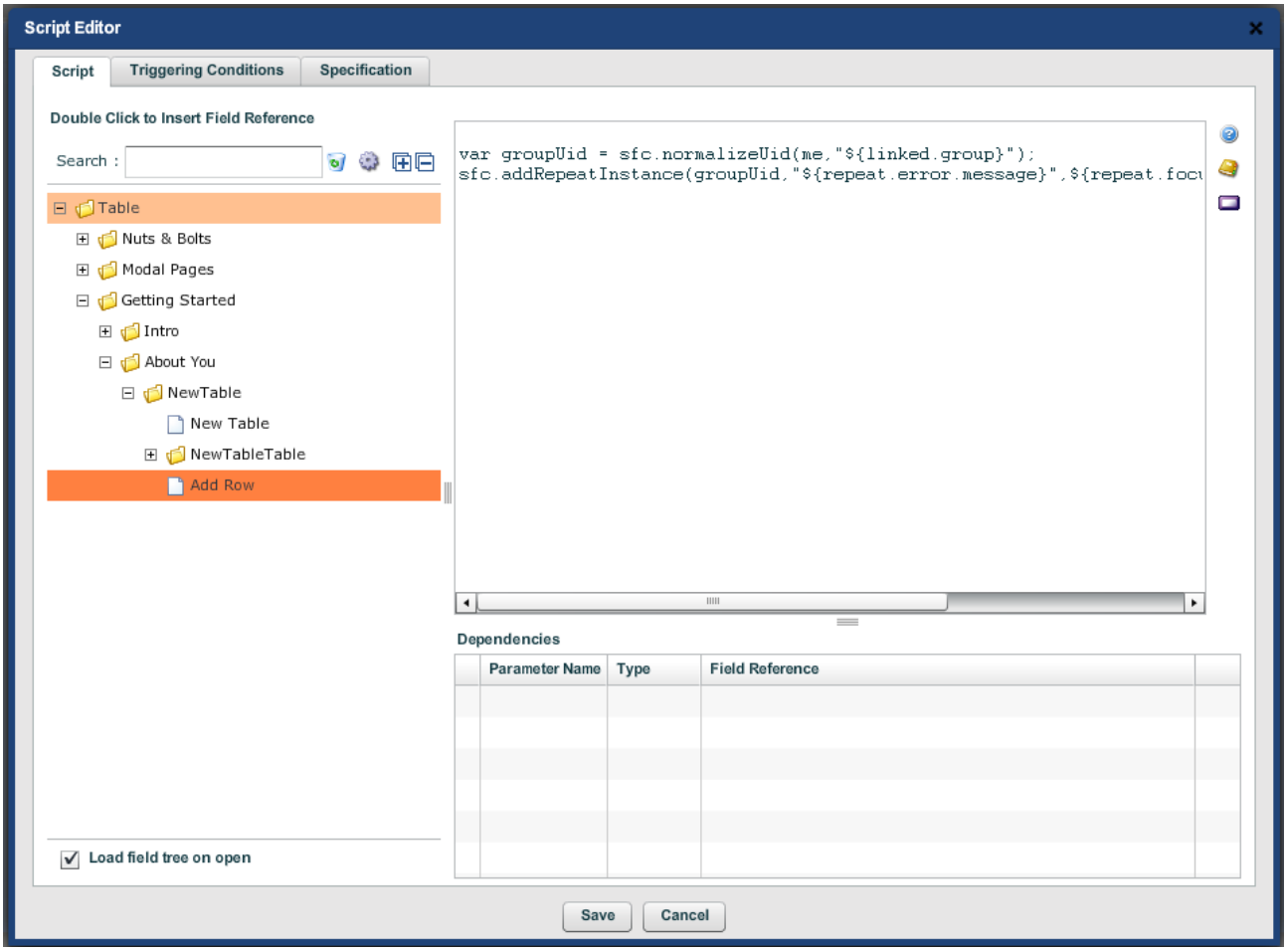
return {*parameter name of integer widget*} + 1;

Add a Footer to the table

1. Drag a 'Table Footer' widget into the table widget. It will be positioned below the 'row' component.
2. Populate it with appropriate widgets. Note: if you are calculating values for a column e.g. a total, a 'spring' widget may be useful when formatting the footer as it is able to span multiple columns.

Repeat Index – How to Use and Adding and Removing Instances

Scripting against Table Row repeating objects makes use of an InstanceManager object. The InstanceManager for a repeating object tracks how many of the objects exist, what are the Min, Max, and Initial settings, and provides methods for creating and deleting Rows.



The above shows the Script Editor for a Button object. The scripting uses the Instance Manager of a Table's Rows to create new rows. The full script is:

```
var manager = {row};
var startCount = manager.getInstanceCount();
var index = 0;


for( index = 0; index < startCount; index++ ) {
  var oldUid = sfc.convertToUid( manager.getInstance( index ) );
  if( sfc.getRawValue( sfc.convertToNode( oldUid + "/LicenseType" ) ) == "Inbound" ) {
    continue;
  }


  var newInstance = manager.addInstance();
  var newUid = sfc.convertToUid( newInstance );


  sfc.setRawValue( sfc.convertToNode( newUid + "/LicenseType" ),
        "Inbound" );
  sfc.setRawValue( sfc.convertToNode( newUid + "/ProductCategory" ),
        sfc.getRawValue( sfc.convertToNode( oldUid + "/ProductCategory" ) ) );
  sfc.setRawValue( sfc.convertToNode( newUid + "/ProductType" ),
        sfc.getRawValue( sfc.convertToNode( oldUid + "/ProductType" ) ) );
  sfc.setRawValue( sfc.convertToNode( newUid + "/ProductSubType" ),
        sfc.getRawValue( sfc.convertToNode( oldUid + "/ProductSubType" ) ) );
  sfc.setRawValue( sfc.convertToNode( newUid + "/Technology" ),
        sfc.getRawValue( sfc.convertToNode( oldUid + "/Technology" ) ) );
}
```

The screenshot shows the Dependency that was created to access the Instance Manager for the Rows to be accessed. Using the Instance Manager object, a count of the current repeating Rows is stored as “startCount.” The for-loop can use “startCount” to scan the repeating Rows, obtaining a noderef to each Row with manager.getInstance().

If the if-statement’s condition is met, a new instance is added to the repeating Rows with a call to manager.addInstance() and the new Row is populated.

This method of accessing individual instances of repeating form elements does not actually use an index or counter to select a Row. Rather, it takes advantage of the return value of manager.addInstance(). The code sample below parse through JSON data and adds it to a Repeat.

An alternate approach to creating new instances of repeating Rows as well as an alternate way to access a repeating Row’s Instance Manager are shown below.

```
var onSuccessNotify = function(evt,me) {
  var jsonData = evt.evtData.resultData;

  // Determine number of "Data Blocks"
  var jsonDataBlocks = jsonData.DataBlock.DataRepeat;
  var cntDataBlocks = jsonDataBlocks.length;

  // If nothing found, say so.
  if(cntDataBlocks < 1) {
    sfc.showAlert("No data found.");
    return;
  }

  // Prep access to the repeating data blocks on the form
  var formDataRepeat = {DataRepeat};
  var imDatas = sfc.findInstanceManager(formDataRepeat);

  // Create a form repeat for each data block, and get the array
  imDatas.setInstanceCount(cntDataBlocks);
  var formDataBlockNodes = sfc.convertToNodeArray(
              formDataRepeat.concat("/DataBlock"));

  for( var i=0; i < cntDataBlocks; i++) {
    // Deal with each data block
    var jsonDataBlock = jsonDataBlocks[i].DataBlock;
    var jsonNameField = jsonDataBlock.NameField;
    sfc.setRawValue(sfc.convertToNode(
          sfc.convertToUid(formDataBlockNodes[i]).concat("/NameField")),
          jsonNameField);
  }
};
```

The code uses a Dependency on the repeating Row and a call to sfc.findInstanceManager() to access the Instance Manager for the repeat. The Instance Manager is used to create a number of instances to match the input JSON data, and then the sfc.convertToNodeArray() is used to obtain an array of node references to the new form objects (the repeated Rows). A for-loop then updates the Name Field with values from the JSON data using the variable “i” as an index.

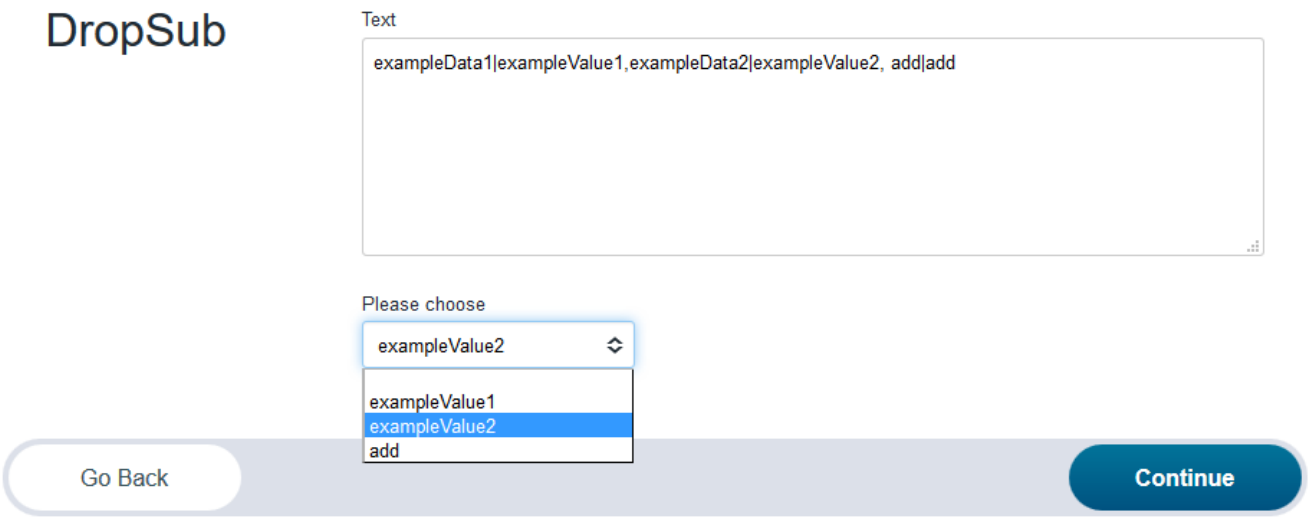
Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

Populating Dropdown Lists

Unknown macro: 'redirect'

A ‘Populating Dropdown List’ is a standard Composer widget used to create dropdown lists that dynamically update their available values based on the content from another field.



For example, a populating dropdown list can retrieve values in a comma separated format from a text field and display them as its available options.

It is also possible to assign a data value to the dropdown selection, separate from the display value, using the following format:

```
dataValue1|displayValue1,dataValue2|displayValue2,...
```

Populating a Dropdown list using API functions

It is possible to populate a dropdown list using the sfc.clearListItems() (to clear the dropdown list before adding items), and the sfc.addItem() functions. The 'Cascading Dropdown' demo form in 'Selections' in the cookbook has an example of this, which uses the following code:

```
var currentValue = sfc.getRawValue({Model});
sfc.clearListItems({Model});
if ({Make} == "Nissan") {
    sfc.addListItem({Model}, "Maxima", "Maxima");
    sfc.addListItem({Model}, "Skyline", "Skyline");
    if (currentValue == "Maxima" || currentValue == "Skyline") {
        sfc.setRawValue({Model}, currentValue);
    }
}
```

When using this method, keep in mind that you can use the sfc.setRawValue() function to set the selected value of the dropdown, if the selected value before the list update is in the updated list.

The APIs for the functions listed above are here:
https://composer.prod.avoka.com/composer/doc/jsdoc/symbols/sfc.html#.addListItem
https://composer.prod.avoka.com/composer/doc/jsdoc/symbols/sfc.html#.clearListItems
https://composer.prod.avoka.com/composer/doc/jsdoc/symbols/sfc.html#.setRawValue

Dropdown Values from Pre-population Data

Setting dropdown list values from pre-population data is fairly straight forward using populating dropdown lists. Map the pre-population data into a hidden text field or data field, then link the populating dropdown list to that field.

For more information on these techniques see articles on 'Pre-population' and 'Hidden Fields'.

Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

Same as Above...Pattern

Unknown macro: 'redirect'

Many forms exhibit a pattern where the user has the option of specifying the one address (often the Residential Address) is the same as the Postal Address (or vice versa). It can be used for other types of data, but is most often seen with addresses because addresses are a very common multi-field block.

There are several different patterns that can be used to implement this seemingly simple solution.

In general, there is no right or wrong way of doing this.

Some things to consider are:

- Is the data feeding a backend system? In other words, is the "same as above" checkbox purely a convenience for the end-user, or does it flow through to the backend system itself, and what does this backend system expect the data to be when the checkbox is checked. This can affect the approach taken.

Sample form: https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=same-as-above-patte3

Built-in Billing/Mailing Address Widget

Composer provides a built in Business Field Type Widget - *Address [Billing + Mailing]* for the common scenario of needing to capture a Billing and Mailing Address. This widget allows the Billing Address to be reused as the Mailing Address via a checkbox.

Cons

- Only works with Australian Addresses "Out of the box"

Copying Blocks of Fields

```
sfc.copyValues (srcBlockNodeRef, targetBlockNodeRef);
```

This will copy the values of all fields from the source block to the target block, based on the names of the fields.

Pros

- This also works for copying data from one row of a table or repeat to another row.

Cons

- This method does not descend recursively into sub-blocks - if you have inserted additional sub-blocks within your block, you will need to copy each sub-block independently.

See Sections ‘Sample A’ (uses a checkbox as a trigger) and ‘Sample B’ (uses a button as a trigger) in the sample form above to see how this can be achieved.

Copying Individual Fields

```
sfc.setRawValue()
```

This method is used to copy the value from one field into another field. This is necessary if the field names in the blocks are not the same.

See Section ‘Sample C’ in the sample form above for an example on how to achieve this.

Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

Selections (One option from many)

Unknown macro: 'redirect'

There are a number of ways to allow end users to select a single option from multiple choices.

There are two dimensions to this problem.

1. The widget type used to present the option to the user.
    a. Dropdown lists
    b. Cascading dropdown lists (eg Country, State, City)
    c. Tables
    d. Dynamic select lists aka type-ahead boxes (similar to google maps interface)
    e. There are potentially other types of list, such as lists of radio buttons or checkboxes - these are currently not recommended or supported.
2. The way in which the selection list is populated
    a. Statically in the form
    b. Injected into the form (pre-fill) at generation time
    c. Dynamically at runtime using a dynamic data call to the server

There are a number of questions that should be considered when deciding which technique to use for any particular lookup in a form (you may mix and match on a particular form):

- How much data is there? This informs the decision of whether the data can be injected statically into the form.
- How often does the data change? This informs the decision of whether the data can be statically bound into the form, or needs to be more dynamic (either prefill or dynamic data).
- The structure of the data. If it's tabular data, a table might be best. If it's hierarchical data, cascading dropdown lists may be best. If it's large amounts of data, a dynamic type-ahead box might be best.
- Does the data need to be available offline in the Mobile App?
- Do older versions of the form need to reflect older versions of the lookup data, or can they always refer to the latest data.

Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

Static Dropdown list

Unknown macro: 'redirect'

A Dropdown list allows a user to select a value from a list of options. Composer has widgets that allow static Dropdown lists to be added and configured within your form.

Data Values

Dropdown Lists have two sets of configurable data values:

- Values. This is a list of values that are the underlying raw values that the field will contain when a user selects a value.
- Display Values. **(Advanced Property)** This is a list of values that are displayed in the user interface. By default, the value is the formula *${values}*. This maps the set of Display Values to match the Values. This can be removed and overridden to set your own list of Display Values, however the number of Display Values and Values must be identical.

Usability

- Consider that long lists of values may be difficult for a user to use.
- Consider a sorting/logical pattern for your list of values to help users scan through

Built-in Widgets

Sample form: https://tm.demo.avoka.com/finance/servlet/SmartForm.html?formCode=static-dropdown-lis3

In addition to the standard common Dropdown List Widget, additional Dropdown List Widgets are provided to suit specific data constraints and commonly used scenarios. These can also be customised.

Advanced Widgets

- Dropdown List [Range] - Customizable Minimum/Maximum Integer range.
- Dropdown List [Years] - Customizable starting year which leads to the current year.

Business Type Widgets

Composer provides built-in Widgets for the following commonly used scenarios

- Dropdown List [Countries]
- Dropdown List [Genders]
- Dropdown List [States, Australia]
- Dropdown List [Titles]
- Dropdown List [Yes/No]

The following have matching sets of Display Values, however the latter scenario uses abbreviated State Codes for the internal value.

- Dropdown List [States, United States]
- Dropdown List [States+Codes, United States]

Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

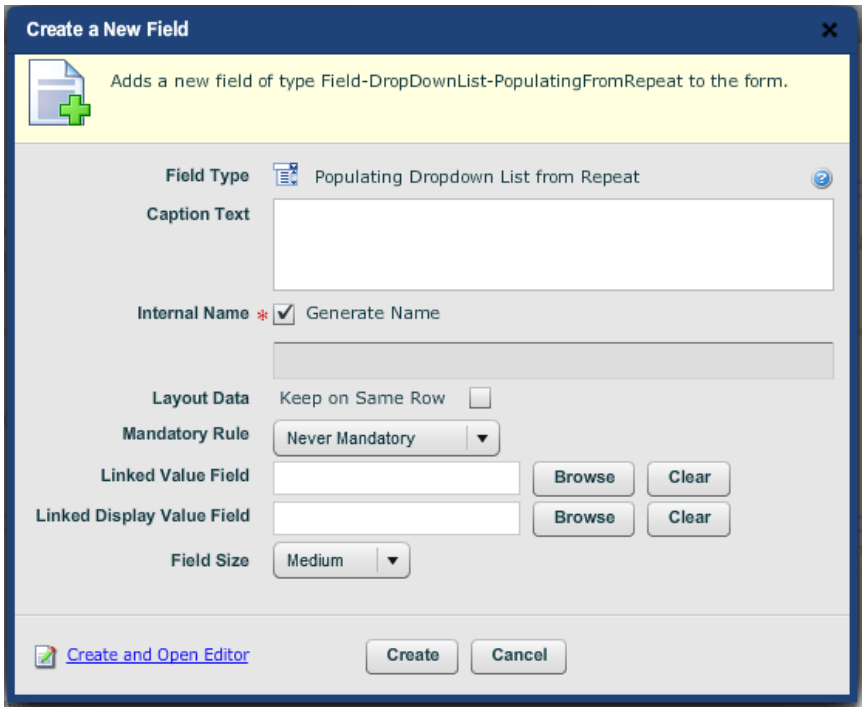
Tabular list from dynamic data

Unknown macro: 'redirect'

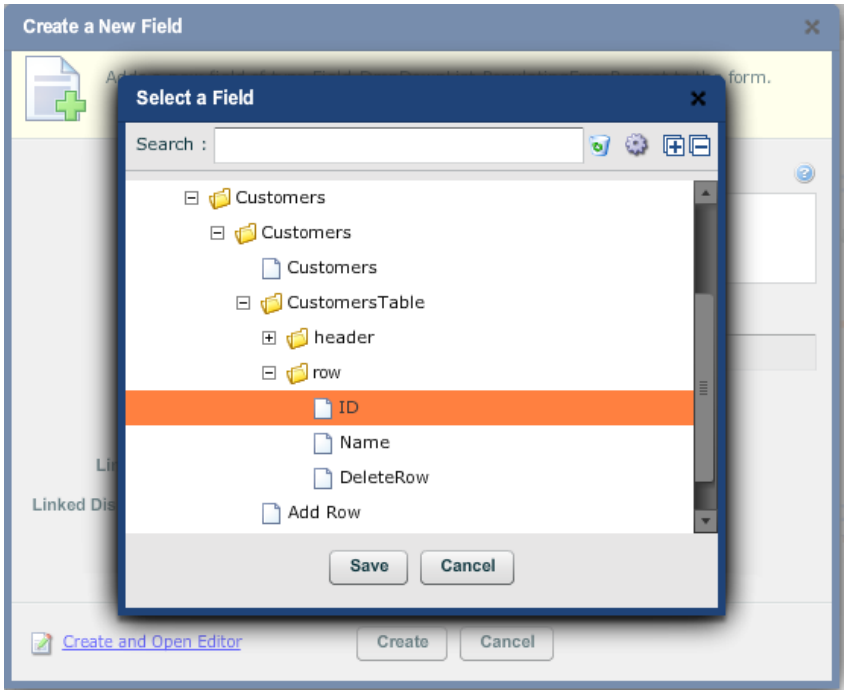
Using a Populating Dropdown List within a repeat widget allows a Dropdown List to be driven by repeating elements of Labels and Values. In the example below, we are using the rows of a table as input.

Data Configuration

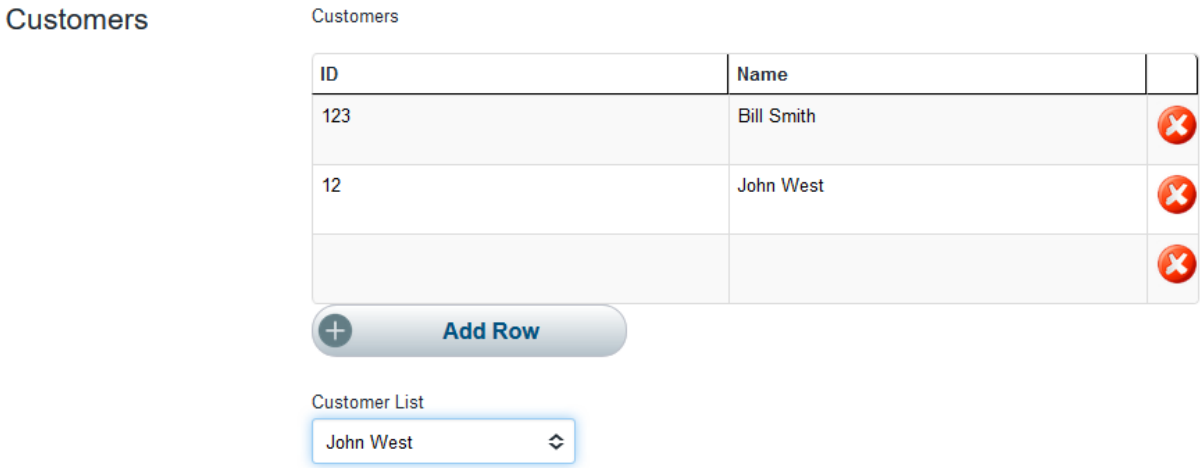
The Link Value Field links to the value repeating element (Such as a value column in a Table Widget's Row).
The Linked Display Value Field links to a label repeating element (Such as the label column in a Table Widget's Row).



A Populating Dropdown List from Repeat configured to use specific columns from a Table Widget's Row



Selecting a column titled Label in a Table Widget's Row for use as the Linked Display Value Field



An example of a Dropdown List being populated by Table Data. The Label column data forms the Display Values visible in the Dropdown List while the Value column forms the value saved in the Form for the selected Dropdown Item

Notes

- Data in the repeating source does not have to be unique.
- For rows that are blank in the table, there will be a blank Dropdown List option
- Each time a row/block is added or deleted from the repeating section, the Dropdown List will be updated accordingly.
- Either links and data for both the Value and Display Value are required, or the Value Link can be used alone.

Cons

- Initial data cannot be set on the Dropdown List unless the initial values in the Table/Repeating section are known.

The Dropdown List in the sample form is driven by data entered into the Table. As data is entered, the options available in the Dropdown List are updated

Sample form: https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=tabular-dropdown-li3

Related articles

- Tabular list from dynamic data
- Static Dropdown list
- Selections (One option from many)
- Same as Above...Pattern
- Populating Dropdown Lists

Collaboration Jobs and Bundling

Unknown macro: 'redirect'

- Collaboration Jobs (Workflow): One step approval guide
- Collaboration Jobs - Separate out emails from Task Assigns using multiple actions
- Collaborative form completion with anonymous users (form sharing)
- How to launch one form from another, with prepop
- Task Assignment
- Transact Collaboration Jobs - Task Sharing, Actings Proxies

Collaboration Jobs (Workflow): One step approval guide

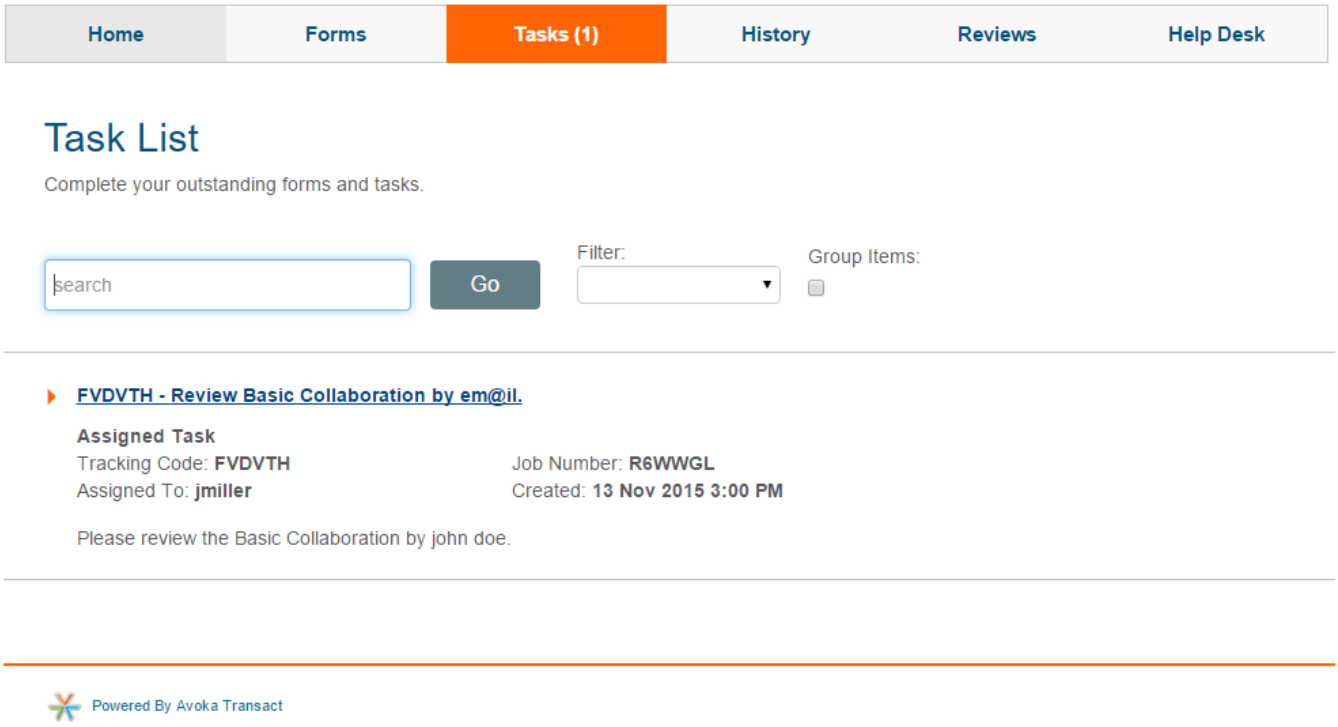
> **Note**
>
> This is an old document ported from Zendesk. It is out of date.
>
> Please refer to the official product documentation Job Basics, Form Design.
>
> This example is documented here 1 Step Review Job.

Old Documentation Below

Basic configuration for a one step workflow.

Introduction:

Collaboration Jobs provide a way to break a job into steps, a step may have one or more tasks associated with it, these tasks may require actions from a different person. Collaboration Jobs are used to create tasks for individual users or user groups, which appear in the user's task list upon logging into the portal.



A common use-case is a form that requires approval from a certain person before being processed. In this case, a Collaboration Jobcan be configured that takes a form submission and creates a task, which is assigned to a user. The user is notified by email that they have a task assigned to them. They then access the task by logging into the portal, and can choose to either accept or reject the form. If the form is accepted, it is then submitted via the normal delivery channel.

The Collaboration Job feature is designed to simplify existing Transact functionality, however it is limited to simple single-thread workflows. To create workflows that spawn multiple tasks in a single step, or require multiple tasks to be completed before initiating a step, you will need to make use of groovy delivery services instead.

Example form:

The example form contains a field labelled 'Assign task to'. By entering your Transaction Manager username into this field, you can assign the resulting task to yourself. To access the task, log into the Finance Corporation portal.

Components of a Collaboration Job:

Collaboration Jobs are driven by 'Job Controller' services. A job controller is a JSON file which defines the steps of the workflow. The Job Controller assigns tasks to users or user groups. Tasks are based on Composer forms which make use of the 'Collaboration' widgets to display content based on the current step in the collaboration process, and to allow the user to choose the next step in the process.

Job Controller breakdown:

The job controller is defined in JSON format. It contains an array of Steps, each of which has a name, type, an array of actions, and an array of routes.

The following step is taken from the "Cookbook Review Demo" Collaboration Job in tm.demo:

```
{
   "name": "Manager Review",
   "type": "",
   "actions": [
      {
         "name": "Assign Review",
         "type": "Job Task Assign",
         "properties": [

            { "name": "Task Assign User", "value": "${formDataMap.assignToUser}" },
            { "name": "Task Form Code", "value": "approval-workflow-fo" },
            { "name": "Task Message", "value": "Please review the ${submission.formName} by ${formDataMap.firstName}
${formDataMap.lastName}." },
            { "name": "Task Review Previous Step", "value": "true" },
            { "name": "Task Send Email", "value": "true" },
            { "name": "Task Subject", "value": "Review ${submission.formName} by ${formDataMap.assignToUser}." },
            { "name": "Task User Deletable", "value": "true" }

         ]
      },
      {
         "name": "Review Wait",
         "type": "Job Task Wait"
      }
   ],
   "routes": [
      { "name": "Approve", "nextStep": "Application Delivery" },
      { "name": "Reject", "nextStep": "Application Rejected" }
   ]
}
```

Name

The name of the step, e.g., "Manager review"

Type

Only required in certain cases, such as "start" and "endpoint".

Actions

An array of actions defined in the step. An action consists of a Name, a type, and an array of properties. Actions defined in Java classes which implement the AbstractJobActionService class, such as JobTaskAssignService, which is configured using the action type "Job Task Assign".
The properties required depend on the type of action, and can be seen in the API documentation of the Java class.

The Java APIs can be accessed at:
<TM server instance URL>/manager/admin/javadoc/index.html
eg: https://tm.demo.avoka.com/manager/admin/javadoc/index.html

Routes

An array of possible steps that the process can progress to on completion of this step. When a task is assigned to a user, these steps can be made visible to them, allowing them to chose the next step, e.g., 'reject' or 'approve'. The steps defined in this array must match the name of another step defined in the job controller, but steps do not have to be defined in seqduential order.

If there is only one possible next step, it should be defied as "default".

Composer form components:

There are two things to consider on the Composer side of things; allowing the user to select the next step, and hiding/showing content based on the current step of the Collaboration Workflow.

Allowing the user to select the next step

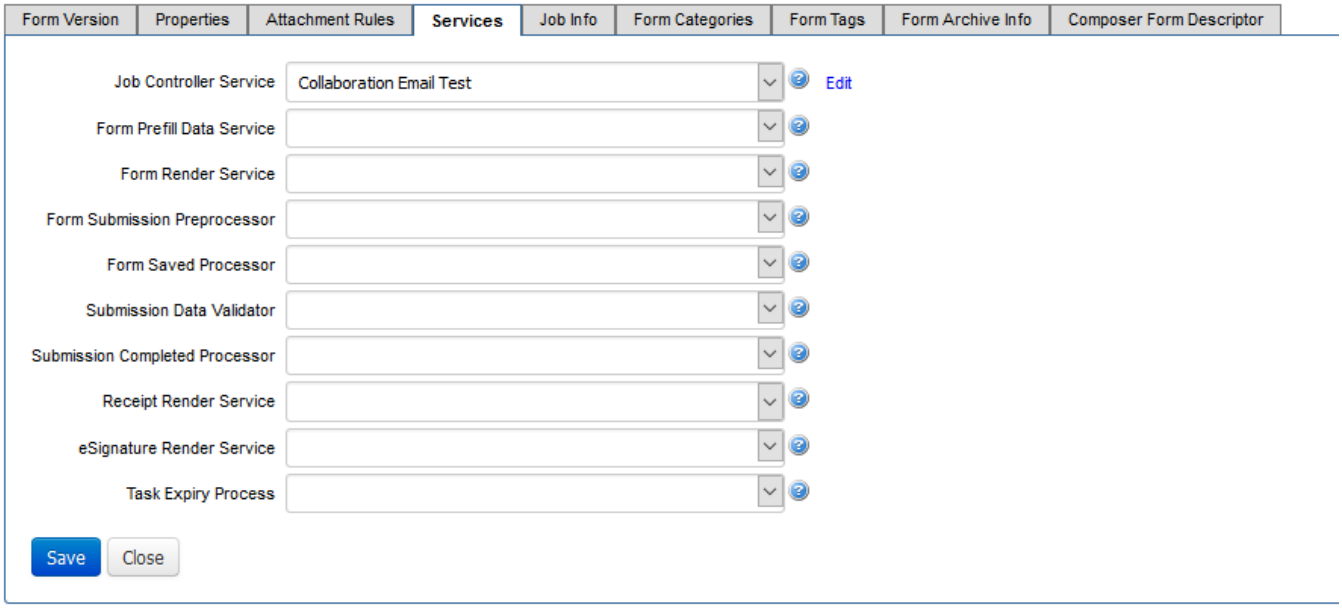
This is achieved by including the "Collaboration Routes Dropdown" widget. This will automatically populate when the task is rendered using the data defined in the "Routes" array for the current step in the workflow. Alternatively, it is possible to configure your own field to collect this data, such as a radio button, and set the data binding to $record.SFMData.SystemProfile.Job.RouteName.

Hiding/Showing content based on the current step in the Collaboration Workflow

This is done using a visibility script that references the $STEPNAME() function, e.g., $STEPNAME() == "Review". To make this easy to remember, there is a 'Collaboration Step' button in the 'Functions' list in the script editor that will insert the function reference into the code.

Form configuration:

To configure a form to trigger a Collaboration Workflow when submitted, publish the form to TM, and in the 'Services' tab, select the Job Controller you wish to use



Collaboration workflows will not be triggered while the form is in test mode, so ensure the 'Test mode' checkbox is unchecked.

To use data from the submitted form in the a task, such as creating dynamic task messages, you will need to configure a Form Data Extract Mapping on the Form -> Data Config -> Form Data Extract Mapping page. Alternatively, this can be done in Composer on the data model tab of the field you wish to map.

To trigger a delivery service from the Job Controller, the form will have to be configured to use the delivery service. It will not be triggered on submission, only when the collaboration process reaches the step in which it is defined as an action.

Related articles

- Transact Collaboration Jobs - Task Sharing, Actings Proxies
- Collaboration Jobs - Separate out emails from Task Assigns using multiple actions
- Collaborative form completion with anonymous users (form sharing)
- Collaboration Jobs and Bundling
- Collaboration Jobs (Workflow): One step approval guide

Collaboration Jobs - Separate out emails from Task Assigns using multiple actions

Unknown macro: 'redirect'

This article describes the steps to modify a Collaboration Job's task assigned email notifications. Normally when a Collaboration Job is submitted by a user the email notifications are sent to all members of the user group assigned as reviewers for that form. This guide will show you how to modify the Collaboration Job to send notifications to a different group of users rather than the task assigned group.

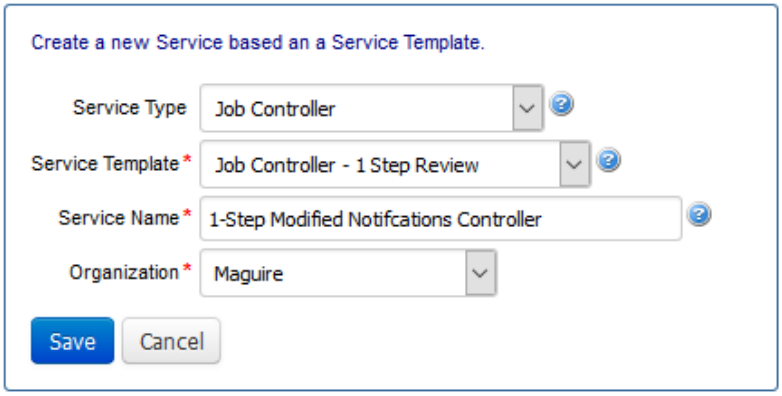
Step-by-step guide

This guide will use a basic 1-step review Collaboration Job.

Collaboration Jobs in TM

In order to adjust the functionality of your collaboration job you will need to make your own Job Controller service.

1. In TM navigate to Service Definitions, in the Systems menu and click New.
2. Create a new 1 Step Job Controller as below.



3. Once created the Job Controller will open showing you the Job Definition. The definition defines the actions the form takes as it is submitted and reviewed.

   TM comes with a number of prebuilt Job Controllers, such as the Review and Approval - 1 Step User Review below. This service assigns a task to a single user and emails them a notification of the task.

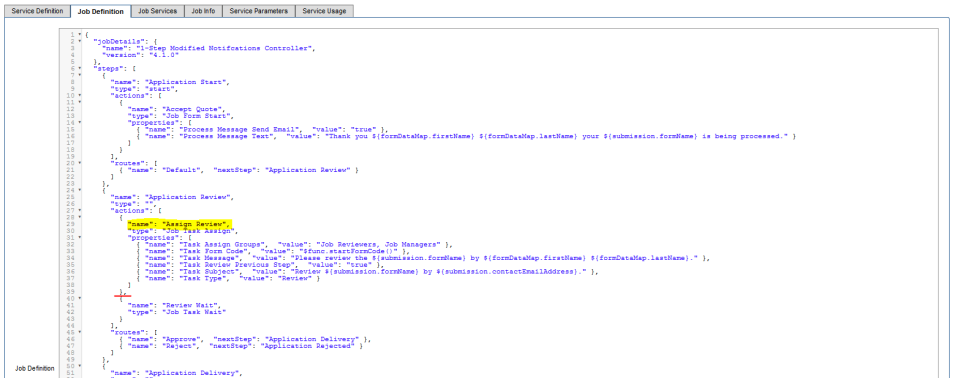
Example of a default Job controller that notifies by email the assigned User.
The following JSON is how the standard Job controller send email notifactions to assigned Users or groups.

Code

```
{ "name": "Task Send Email",  "value": "true" },
{ "name": "Task Email Message",  "value": "$func.invoke('Render Job Template', 'Email Task Notification Message')" },
{ "name": "Task Email Subject",  "value": "${submission.formName} for ${formDataMap.projectTitle} from ${formDataMap.applicantName} requires your sign off" }
```

These emails would be recieved by the assigned users or all member users of the groups assigned in the Assign Review step; depending on whether the task is assigned by groups or users. To add the none standard functionality for a seperate email group, emails notifications will need to be made outside the standard assign review task.

Below is the default template that is provided with your 1 step review job controller.



Notice that this default template doesn't contain the email functionality we will add this next.

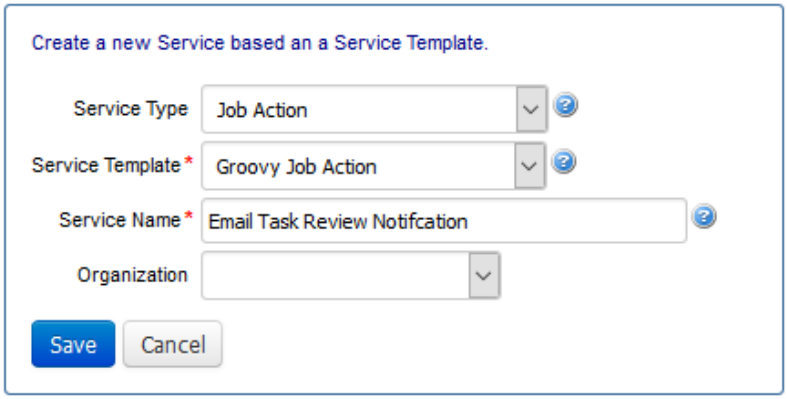
4. First add the following lines of code to the job definition, just after the closing curly bracket and comma of the Assign Review Task, red line.

Code

```
{
    "name": "Email Task Review Notifcation",
    "type": "Job Action"
},
```

This code tells the job controller to call a groovy job action, that we will write next. This groovy code is what we will use to send the emails to the other group of users.

5. Save the Job Controller and return to the Service Definitions Page.
6. Click New and create a Job Action service as below.



7. Click the Groovy Script Tab, here you add the code to send emails to users. Below is an example script that sends a basic email, through the TM email queue, to a group of users in Manager I created called Email Notice.

Groovy Code

```
/* Groovy Job Action service which is used to execute job step actions.

  Script parameters include:
     actionContext : com.avoka.fc.core.service.job.ActionContext
     actionStepProperties :  com.avoka.fc.core.service.job.ActionStepProperties
     serviceDefinition : com.avoka.fc.core.entity.ServiceDefinition
     serviceParameters : Map<String, String>
     job : com.avoka.fc.core.entity.Job
     submission : com.avoka.fc.core.entity.Submission
     formDataMap : Map<String, String>
```

```
   Script return:
      actionResult : com.avoka.fc.core.service.job.ActionResult
*/
import com.avoka.fc.core.service.job.ActionContext;
import com.avoka.fc.core.service.job.ActionResult;
import com.avoka.fc.core.service.job.ActionResult.Status;

import com.avoka.core.groovy.GroovyLogger as logger;
import com.avoka.fc.core.entity.EventLog;
import com.avoka.fc.core.service.EventLogService;
import com.avoka.fc.core.service.ServiceFactory;
import com.avoka.fc.core.dao.UserAccountDao;
import com.avoka.fc.core.dao.SubmissionDataDao;
import com.avoka.fc.core.dao.PortalDao;
import com.avoka.fc.core.util.PortalUtils;
import java.util.List;
import java.util.Map;
import com.avoka.core.util.StringTemplate;


// Get the list of submissions and the orginal form submission for the Collabaration Job
def submission = job.getSubmissions();
def firstSubmission = job.firstSubmission;

//Iterate through the submission list and find the task review submission
for(def subTemp in submission){
  if(subTemp.getTaskType() == "Review"){

    def submissionDataDao = new SubmissionDataDao();
    def formDataMap = submissionDataDao.getFormDataMap(subTemp);

    def EventLog = new EventLogService();
    def userDao = new UserAccountDao();
    def portalDao = new PortalDao();
    def emailService = ServiceFactory.emailService;

    //Retrieve a list of users in the notification group
    def userListGrp = userDao.getActiveUsersInGroup("Email Notice");
    def emailsQueued = 0;
    def adminPortal = portalDao.getAdminConsolePortal();
    def fromAddress = "donotreply@avoka.com";

    def refNum = subTemp.getTrackingNumber();
    def formCode = subTemp.getFormCode();
    def portal = firstSubmission.getPortal();

    //Iterate through the list of users
    for(def temp in userListGrp){
      //For each user add a subject and custom email message
      def subject = "New form submitted for review";

      def url = PortalUtils.getFormSavedUrl(portal, subTemp);
      // Map values for email template
      Map model = new HashMap();
        model.put("formName", subTemp.getFormName());
        model.put("url", url);


      def message = new StringTemplate(serviceDefinition.getServiceParameter("Email").getValue());
      def strMessage = message.merge(model);

      //Add the email to the email queue
      emailService.queueEmail(adminPortal, fromAddress, temp.getEmail(), subject, strMessage, "New Form for Review");
      ++emailsQueued;
    }

    //Add an event log entry listing the number of emails added to queue
    EventLog.logInfoEvent("There has been " + emailsQueued + " added to the email queue.");
  }
}

return new ActionResult(Status.COMPLETED)
```

This groovy code for creating an email and pushing it to the email queue

The above code takes the Job and finds the Submission relating to the Review Task. It then iterates through all the users associated with the review group and adds an email to the email queue for each of them. To create the email message a service parameter is used that defines the html template for the email body, this is shown next. In order to include values like the form review url, form name and other information that may be needed by the user, a string template is created to map and then merge these values with the html template.

The ideal url to pass to the notification email is structured as follows:

https://SERVER_CONTEXT_PATH/servlet/SmartForm.html?formCode=FORM_CODE&saveChallenge=true&referenceNumber=REF_NUMBER

On some systems the PortalUtils.getFormSavedUrl method may return a url like:

https://SERVER_CONTEXT_PATH/secure/form.htm?submitKey=SUBMIT_KEY

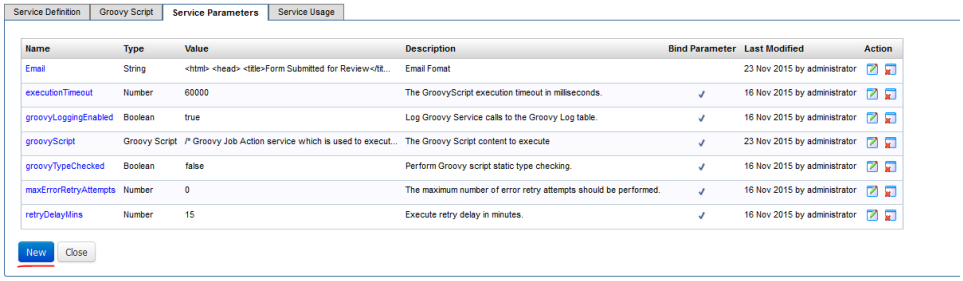
This can cause access/authorisation errors when tasks are assigned to different users from the orginal submitter. The work around for this is to try some of the other url build methods,

- getSaveChallengeUrl
- getFormSavedUrlWithReferenceNumber

You can also build the review url manual with the following code:

```
def context = portal.getContextPath()
def formCode = submission.getFormCode()
def trackingNumber = submission.getTrackingNumber()
def url = context + "servlet/SmartForm.html?formCode=" +  formCode +
"&saveChallenge=true&referenceNumber=" + trackingNumber
```

8. Click Save.
9. As stated above you will also need to add a service parameter that will be the email template for the notification message. Navigate to the service parameters tab and click new.



10. Call the parameter Email and choose String as the Type. In the Textbox that appears you can add the email template. Below is an example.

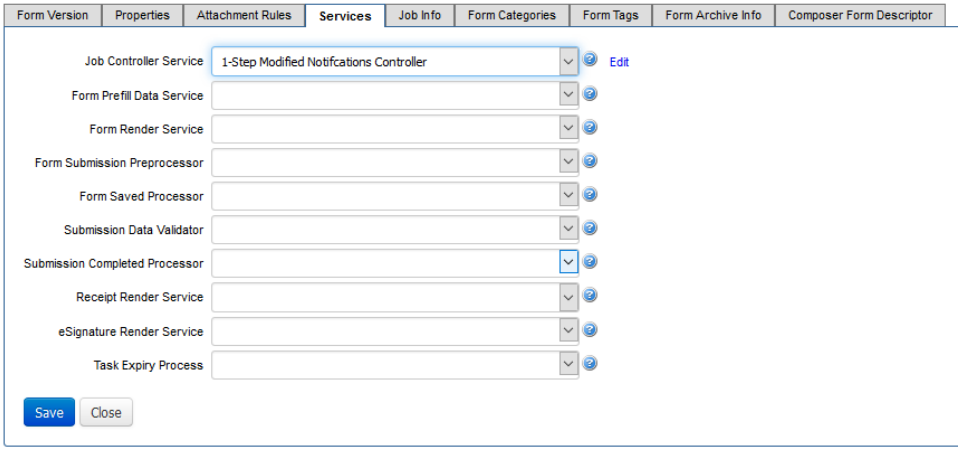
Email Template

```
<html>
<head> <title>Form Submitted for Review</title> </head>
<body style="background:#ececec;">
   <center style="padding:1em;">
   <table style="width:640px; border-collapse:collapse; font-family:Arial,Helvetica,sans-serif;">
   <tr>
      <th style="color:white; background:#005596; text-align:left; padding:1em;">
          Please review the new Submitted Form
      </th>
   </tr>
   <tr>
      <td style="color:black; background:white; text-align:left; padding:1em;">
      <p> A form, ${formName}, has been submitted for review.</p>
      <p> Please review the form by clicking below: </p>
      <p> <a href="${url}">Review Form</a> </p>
      </td>
   </tr>
   <tr>
      <td style="color:#d9fffd; background:#556c90; text-align:left; padding:1em;">
      </td>
   </tr>
   </table>
   </center>
</body>
</html>
```

This parameter is used by the groovy code to build the body of the message by mapping certain values like the form review url and form name to the email message.

Note: Review FormIs a link to the review form in the email. The ${url} value is mapped in the groovy code as is the ${formname}. These values are generated in the groovy code and mapped to the String Template used to build the unique email message for each review task.

11. Click save and navigate back to the Home Dashboard. Find your Collaboration Form and navigate to it.
12. Under Form Versions go to Properties and click the Services Tab.
13. Select your new Job Controller from the Dropdown next to the Job Controller Service entry.



14. Click save and test your form.

Related articles

- Transaction Delivery
- Transaction Reporting Feed
- TField and Locations
- Collaboration Jobs - Rolling Back To a Previous Step (State)
- test

Collaborative form completion with anonymous users (form sharing)

Unknown macro: 'redirect'

Compatibility

Since	4.0
Deprecated	

Transact has the capability that enables a form to be shared among anonymous users before form submission.

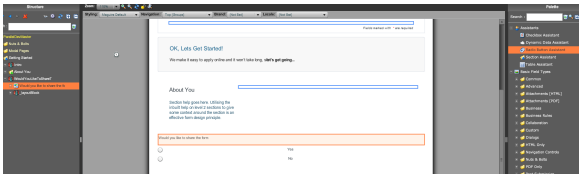
The sharing involves giving the name, email of the nominated other party, and a text area for notes to the other party; then the user, instead of submitting the form (as would be done if the form were complete), clicks on a dedicated button with some appropriate caption like "Activate Sharing". TM generates an email containing a link. The other party clicks on the link and is taken to the form (prefilled with the other user's data) which can then be edited, files attached and so forth.

Example

In Composer

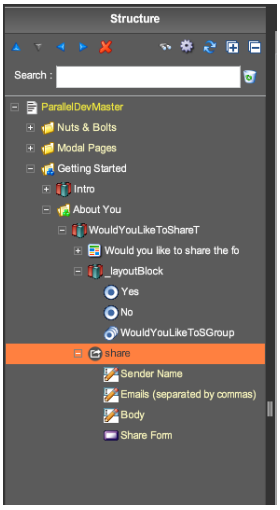
Build your form.

For the Share capability, create a set of 'Yes' and 'No' radio buttons with text of 'Would you like to share the form?'



Drag the 'TM Share Form Block' widget into the 'YesContent' block. There are mandatory fields in the block to capture the contact details of the original form user, otherwise there will be no connection to that person.

Resulting 'Structure pane':



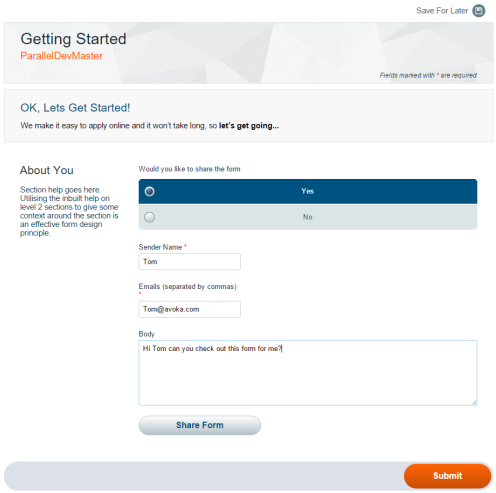
In Transaction Manager

Publish the form to Transaction Manager.

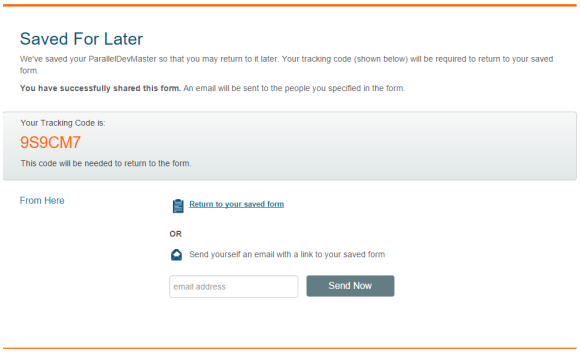
Change the form dashboard to ensure that the property 'Save Online' is 'Anonymous'. This ensures that o ther parties can also save the form when they activate the share button. The 'User Authentication' property must be set to blank (see below) or 'Anonymous'. If not, the prefabricated TM Sharing block has scripts that make it invisible to authenticated users.

Testing the form

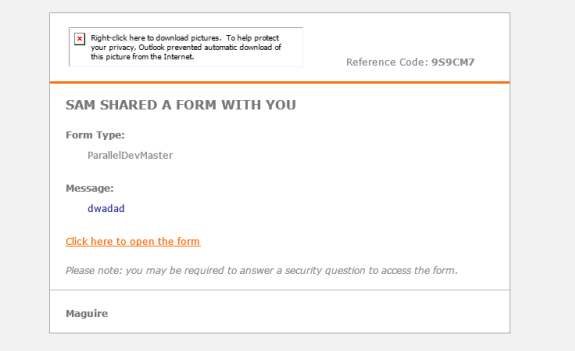
John Smith fills out the form and decides to share the form the a third party Tom



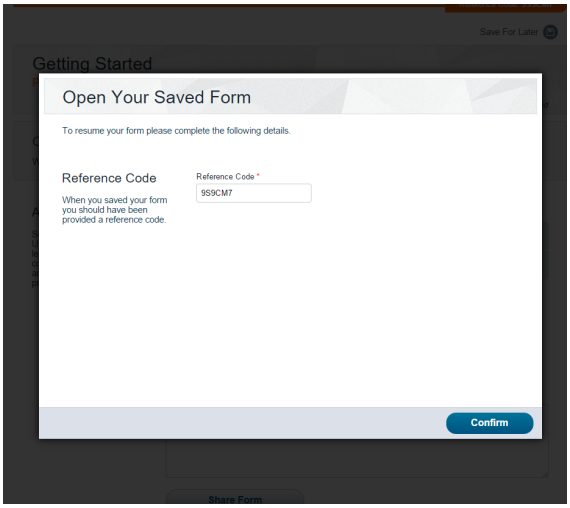
Result:



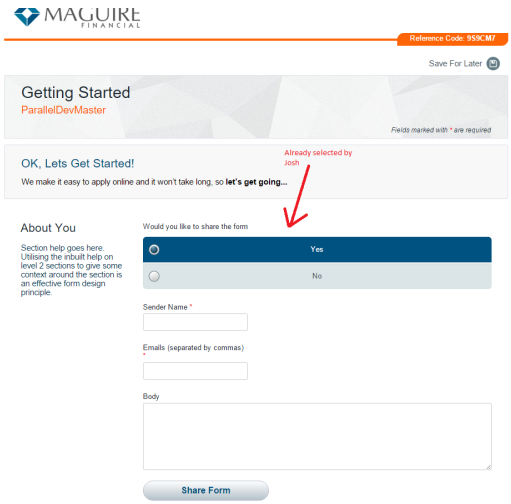
Tom will receive an email with the following message.



When she selects the form link, she will see the following security check page.



On opening the form, he will see the form prepopulated with information already completed by John.



He completes the Policy Holder Details section and submits the form.

In Transaction Manager, select Operations > Form Transactions > and review the transaction history with your forms tracking code .

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Accessing Form properties in Groovy Services
- Styling & Branding
- Transact Collaboration Jobs - Task Sharing, Actings Proxies
- How to launch one form from another, with prepop

How to launch one form from another, with prepop

Unknown macro: 'redirect'

Use case

Sometimes you may have an existing database of information that you want to use as a basis for pre-populating a form. This data is often known as Context Data.

For example, you may have an existing customer database (or list of properties or contracts or medical records or accounts or property locations or whatever), and you want to be able to:

- search for a customer by various criteria
- view summary information for all matching customers
- click a button to open a form on behalf of that customer (there could be multiple forms to choose from)
- have that form open in new window, and be pre-populated with the customer information.
- The form submits in the usual way.

Technique

1. Search form

Build a search form. This will use a Dynamic data lookup to populate a table or a repeat with information from the database. (This tech note does not cover how to build this search form using dynamic data, that is covered elsewhere.)

There are two alternatives.

1. If there is just a small amount of data, pass all the data in request parameters to the new form, and use pre-fill mappings to map the request parameters into form field data.
2. If there is a larger amount of data, pass a unique identifier in a request parameter, and use a form pre-fill service to populate the form.

This article will look at the second technique which is slightly more complicated - the first technique is very similar

In the result row, including a button with the following script:

```
var param = {MyUniqueIdField}; // This parameter is the unique id that comes from the repeating section
var formUrl = "https://tm.demo.avoka.com/finance/servlet/SmartForm.html?formCode=myformcode";
// Append a parameter to the end of the URL
window.open(formUrl + '&myparam=' + param,'_blank');
```

This script will launch the form named "myformcode" with an extra parameter.

Note that the Search form does not have a submit button - it is used only to retrieve data, and launch

2. Main form

The Main form will use a pre-pop script in order to fetch prefill data. There are several different ways to do the actual prepop, but the key issue is getting hold of the URL parameter that was passed from the Search form.

If you create a new service of type Groovy pre-fill script, it accepts several parameters, including:

```
formRequestParams : Map<String, String>
```

In order to retrieve our parameter, we would use something like this:

```
var myParam = formRequestParams['myparam'];
```

Thereafter, use this parameter to retrieve data from the backend system, and generate the XML pre-pop data.

Please see the following page for more information, and also refer to the Transaction Manager documentation.

https://tm.demo.avoka.com/manager/admin/help/topics/07-form-prefill-data.html

Alternatives

- Another way to do this is to directly add a similar button into the native application that you use for your customers. However, sometimes this is not possible to do, and building the Search form within Transact may be useful.

- This sample hard-codes the location of the server ('formUrl" in the JavaScript code above). You can build up the URL dynamically by using some of the values injected into the form by Transaction Manager - you will need to use the DynamicDataServiceURL and strip out some of it in order to find the base URL for forms. This has not been included in this sample to keep it simple.

Restrictions

This technique cannot be used in the Transact Field App.

Related articles

- How to launch one form from another, with prepop
- Collaborative form completion with anonymous users (form sharing)
- Setting up 'Leave Page Confirmation' in Composer using JavaScript
- Understanding transaction management in Maguire forms
- Building a Top navigation menu and 2-level navigation menus

Task Assignment

Unknown macro: 'redirect'

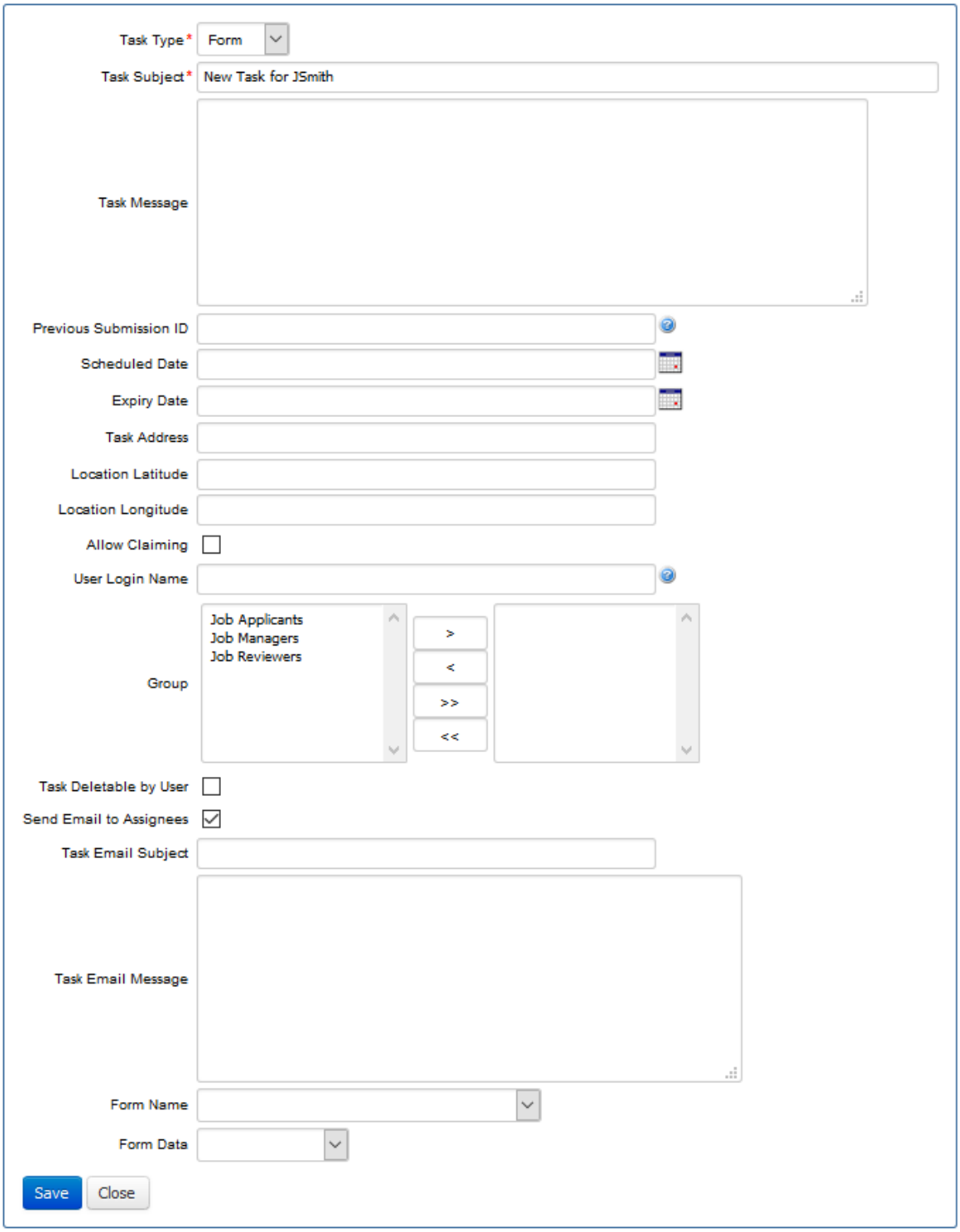
Transaction Manager provides several capabilities for assigning "tasks" to a user. There are several different patterns for doing this.
Note: In Transaction Manager terms, a "task" is a combination of:

- a form (or form-code)
- some XML data (which is used to pre-populate the form)
- the ID of a user or group to whom the task is allocated.

"Internal" tasks are also created when a user has not completed a transaction - for example, when they have saved but not submitted a form, submitted but not paid or added attachments, etc.

Notes

- When a task is allocated, you may configure an email containing the task details to be sent to the assignee.



- You can also set up scheduled jobs to monitor tasks, and send reminders if they haven't been completed after a period of time.

Related articles

- Task Assignment
- Configuring confirmation emails upon Save and Submit
- How to create and share a library V4
- Importing/Exporting Organizations from Composer

- How to create a Library in V4

Transact Collaboration Jobs - Task Sharing, Actings Proxies

Unknown macro: 'redirect'

How does Transact achieve "Task Sharing" versus "Actings" and "Proxies"

Acting's and Proxies are used with systems like Adobe LiveCycle process management and other BPM systems. What they achieve is the ability to complete workflow user tasks when a user is on leave holidays or away sick for a week.

This document briefly describe what Acting or Proxies are, how Transaction manager achieves similar functionality and a simple example.

So what are Acting / Proxies?

When an individual has been assigned to a workflow task. If they are going on holiday the assigned user can arbitrarily allocate any other workflow user to take care of their tasks while they are away. The new user becomes the original user's "proxy" or is said to be "acting" on behalf of the user.

Advantages

- Decentralised
    - Easy for an end user to delegate their job.

Disadvantages

- Decentralised
    - The proxy may not have the right training to do the job.
    - The assigned user may not have the right security clearance to perform the job.
    - Users that are off sick may not have set a proxy. Meaning the tasks don't get actioned.
    - Difficult to transfer relationships between environments (DEV, UAT and PROD)
    - Management cannot change the relationship without logging in as the user.

How Does Transaction Manager achieve the same functionality?

Transaction Manager via the Task API or Collaboration Jobs achieves this via the use of **form groups**. A group task can be assigned to one or more groups. Only people that are a member of one of the groups can complete the task. If a member of a group goes on leave then another group member can complete their tasks.

Transact 4.1 introduces claiming for group queues. Any group member can be assigned task claim permissions that allow them to claim an available task, but cannot open those claimed by others. Supervisors can be granted task re-assign permission which can allow them to claim a task from another user or reassign it to another group member.

Transaction Manager's approach is similar to how the military roles works. Let's say a Major in the Army is has a role (equivalent of a form group) of "Top Secret Projects". He and anyone assigned the role are the only one that is able to review the "Top Secret Projects". He may have a secretary but he can't assign the secretary access. The secretary needs to get the "Top Secret Projects" role before she can do the role. This would be done by someone higher up the chain of command. In a similar way if the Major goes on leave another officer must be given the role "Top Secret Projects".

Advantages

- Centralised
    - Coordinators have full visibility and control over who is doing what
    - Forces end users to follow process which ensures the user has the proper security and permissions
    - Can easily give other users access to the queue in event the user didn't before going on leave
    - Can easily move settings between environments (DEV, UAT and PROD)
    - Existing group members can do the work for other group members on leave

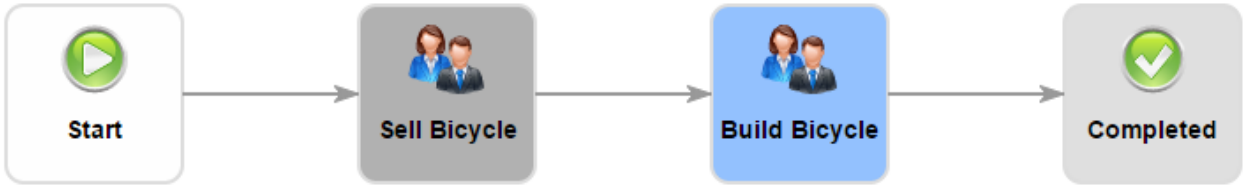
Disadvantage

- Centralised
    - It may be slightly quicker for the end user to arbitrary assign another user to do their job. Now they may have to contact the coordinator to arrange it.

Simple Example of Transact Collaboration Jobs – Task Sharing

This example job below uses a bicycle shop and concentrates on the bicycle sales process, refer to the Bicycle Sales Job below. As detailed In the table below, the shop has a couple of Sales people who responsibility is to sell the bicycle, 3 bicycle mechanics that are responsible for building the bicycles and a couple of managers to oversee both areas.

Bicycle Sales Job (Workflow) showing the Group Tasks



Bicycle Shop Groups, Members and Special Permissions

Group	Members	Special Permission
Sales	Dave Jenny	
Mechanics	Ken Kylie John	
Management	Matt Steve	Task Reassign

The **Sales** group are assign to the Sell Bicycle step. The Build Bicycle step would be assigned to the **Mechanics**. In addition the **Management** would be assigned to both groups and has Task Reassign permissions.

Sales works on a task pull method. As a customer first comes into the shop a sales person will approach them and get there details after that the customer is associated with the same sales person. Once claimed Jenny and Dave do not have access to work on or claim each other's sales.

The workshop uses a task push method. At build bicycle step a manager opens the task and assigns the one of the mechanics (push method). Ken, Kylie and John can only work on tasks that have been assigned to them.

What happens if a Dave in Sales goes on holidays who looks after his work. The manager can claim Dave's tasks and complete the sale. Alternatively the Manager can assign a Task to Jenny.

If John is half way on a bike but has to goes home ill. The manager can assign this task to Kylie. Let's say John is very sick and can't come back to work for a Month. They can get another temp Bike mechanic Colin to fill in, he gets added to the mechanics group. The manager can then assign him John's work.

Related articles

- Transact Collaboration Jobs - Task Sharing, Actings Proxies
- Collaboration Jobs - Separate out emails from Task Assigns using multiple actions
- Collaborative form completion with anonymous users (form sharing)
- Collaboration Jobs and Bundling
- Collaboration Jobs (Workflow): One step approval guide

Groovy Services

Unknown macro: 'redirect'

Please note that Transact 5 includes a new Fluent SDK that supports IDE development of Groovy Service and is much more productive. For more information please see the documentation: SDK51.

Groovy scripting in Transact

Unknown macro: 'redirect'

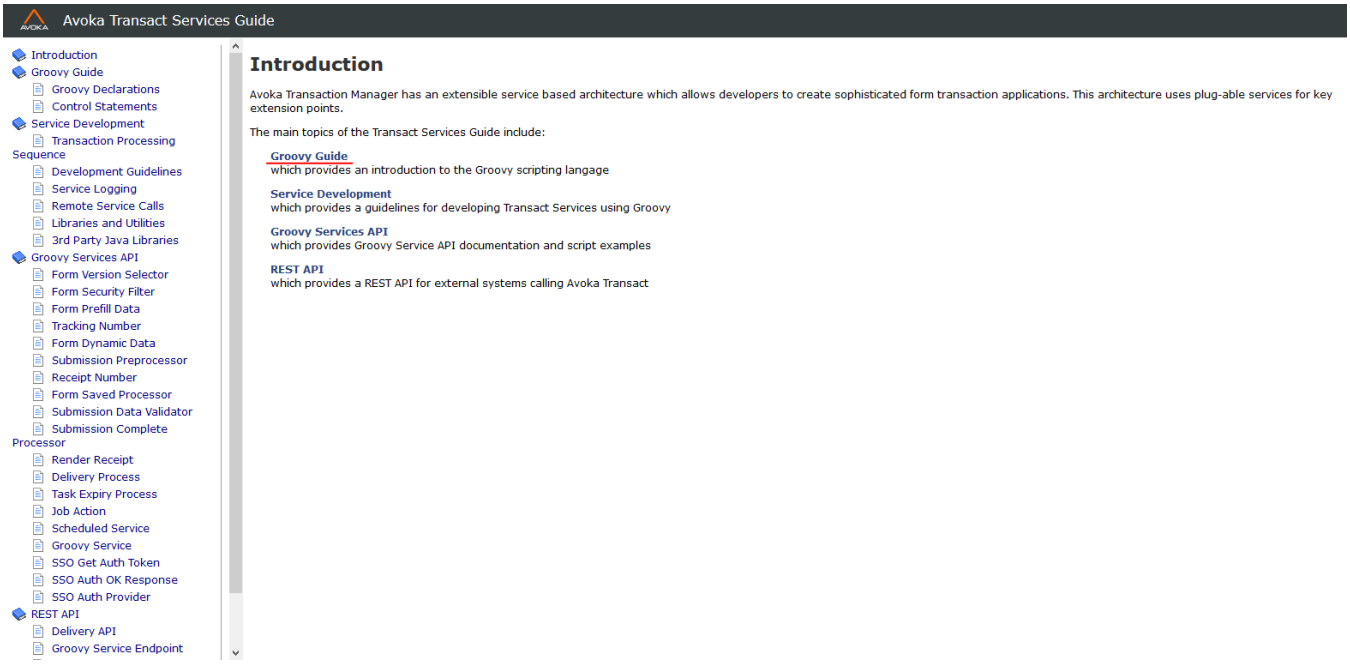
Transaction Manager provides a very flexible scripting language called Groovy. Groovy is quite a sophisticated language, and you can use it for all sorts of things in Transaction Manager, including doing form delivery processes, and linking dynamic data calls from a form to a database or web service, etc.

Groovy is an agile and dynamic language for the Java Virtual Machine. Groovy is used in Avoka Transact because it allows the development of fully customizable scripts, without re-compiling and re-deploying applications. For information about the Groovy scripting language, see http://groovy.codehaus.org/.

For a beginner's tutorial, see http://groovy.codehaus.org/Beginners+Tutorial

Groovy Services Guide

For information about Transact implementation of groovy scripts, including what they can be used for, and what data objects they have available to them, see the Groovy section of the **Transact Services Guide** under the **Services** menu in your Transaction Manager environment (can be found under the **System** menu in older installations).



Groovy is Glue

Please be aware that it can be unwise to try to do too much coding in Groovy. Our Groovy implementation is intended more as "glue" between Transaction Manager and external systems, rather than as an integration capability in its own right.

As a rough rule of thumb, you shouldn't have more than 50-100 lines of code in a Groovy script (*), and it should really only be used to link Transaction Manager to the back-end system, rather than trying to "be" the backend system.

One thing that Groovy can be very effectively used for is for relatively simple delivery processes - in the past, we used to use LiveCycle Process Management for this, but often you can achieve a similar outcome with a simpler implementation using Groovy. If you couple Groovy with delivery checkpoints, you can do some very powerful things very simply.

Related articles

- How to mix standard delivery functions with custom groovy processing
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating new form tasks in a Groovy delivery service

- Creating anonymous tasks via Groovy service

Groovy tips and tricks

Unknown macro: 'redirect'

Get the version of Groovy in Transaction Manager

```
print GroovySystem.getVersion()
```

Get Date widget into a W3C dateTime

The W3C has a specificationfor date time formats which has resulted in the sublime XMLGregorianCalendar java class. While Groovy provides some convenience mechanisms to work with Java types such as Date, an it's relatively easy to use format specifiers to produce compatible dateTime formats, I prefer the brute force code approach. Below is an example that takes a date from a COmposer widget and converts it into W3C compatibe dateTime via Calendar objects

```
javax.xml.datatype.DatatypeFactory.newInstance().newXMLGregorianCalendar(Date.parse("yyyyMMdd","19710517").toCalendar())
```

Here's a similar method that takes a date object and produces the compatible dateTime format

```
def StringBuffer timestamp = new StringBuffer(new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ").format(new Date()))
```

Transaction Manager included libraries

An easy way to check this is go to jboss module folder for groovy (e.g C:\avoka\transact\manager\server\modules\org\codehaus\groovy\main)
The file module.xml specifies like the following:

Transaction Manager 3.5

```
<module xmlns="urn:jboss:module:1.0" name="org.codehaus.groovy">
   <resources>
      <resource-root path="groovy-all-2.1.0.patch.jar"/>
      <resource-root path="groovy-wslite-0.7.1.jar"/>
      <resource-root path="http-builder-0.5.2.jar"/>
   </resources>

   <dependencies> <module name="com.avoka.core" />
      <module name="com.microsoft.sqlserver" />
      <module name="com.mysql" /> <module name="com.nuodb" />
      <module name="com.oracle" /> <module name="javax.api"/>
      <module name="javax.servlet.api"/>
      <module name="net.sourceforge.json" />
      <module name="net.sourceforge.saxon" />
      <module name="net.sourceforge.nekohtml"/>
      <module name="org.apache.camel.core" />
      <module name="org.apache.camel.spring" />
      <module name="org.apache.commons.codec" />
      <module name="org.apache.commons.collections" />
      <module name="org.apache.commons.lang" />
      <module name="org.apache.commons.logging" />
      <module name="org.apache.commons.io" />
      <module name="org.apache.httpcomponents"/>
      <module name="org.apache.log4j"/>
      <module name="org.apache.xml.xml-resolver"/>
      <module name="org.junit"/> <module name="org.opensaml" />
      <module name="org.slf4j" /> <module name="sun.jdk" />
      <module name="system" />
   </dependencies>
</module>
```

The dependency section and resources section are libs that your groovy scripts can use.

Groovy script body building

You can use the following code template to develop a SOAP message in the Groovy Script Console in Transaction Manager

```
import wslite.soap.*

def body = new SOAPMessageBuilder().build {
    version SOAPVersion.V1_2
    encoding('UTF-8')
    envelopeAttributes(
        'xmlns:_04':'http://cai.dataaction.com.au/Cai/Member/New/2011_04_15/',
        'xmlns:dat':'http://schemas.datacontract.org/2004/07/DataAction.Cai.Wcf',
        'xmlns:new':'http://cai.dataaction.com.au/Cai/Member/New/'
    )
    header{
        Action('soap-env:mustUnderstand':1, xmlns:'http://schemas.microsoft.com/ws/2005/05/addressing/none', 'http://cai.
dataaction.com.au/Cai/Member/New/2011_04_15/MemberContro...)
        To('soap-env:mustUnderstand':1, xmlns:'http://schemas.microsoft.com/ws/2005/05/addressing/none', 'https://dat
a-dfvcai3.cai.dataaction.com.au:8443/cai/member/new/2011...)
    }
    body{
        '_04:CreateNewPersonalMember'{
            '_04:newMember'{
                'dat:Organisation' ('Avoka')
                'dat:RequestId'(UUID.randomUUID())
            }
        }
    }
}
print body
```

Capturing Groovy Script Output

Capturing Groovy Script Output with SystemOuptutInterceptor

```
import groovy.ui.SystemOutputInterceptor
...
def outputString = ''
// a simple closure to add the iterator to the string; returning false diverts the output from system.out
def interceptor = new SystemOutputInterceptor({ outputString += it; false})


interceptor.start()
// ...do some Groovy code that generates output
interceptor.stop()


print outputString
```

Use a Builder and a Map to Conveniently Return Dynamic Data

Dynamic Data Services in Avoka Transact require data ina specific format that can be a little tricky to format. It uses an array around Json objects to parse and fill repeating groups of output fields in the form. That format looks like this:

```
[{"Name":"Value"},{"Name":"Value"}]
```

But that can be a real pain to get right if you're trying to do it using string concatenationand placeolders, error prone at best.
This example produces a Json object using a JsonBuilder with a Closure. Wraps it nicely inside an array and then returns as a string (to the Dynamic Data Service)

```
JsonBuilder + Map = Dynamic Data
def json = new JsonBuilder({
        Name(Value)
    })
def arr = [json.toString()]
return arr.toString()
```

Mocking up Web Service SOAP return data

Developing services on Transaction Manager is not the friendliest or easiest of experiences simply because it's intent is to provide a lean administration system, not a development enviornment. Yet there are times when the only way to get at the values you need are on Transaction manager, especially when connected to remote, thrid party systems. This is a simple example top mock up a web service response which you can then use to develop the response action like extracting data.

Mocking up a SOAP Response

```
def soapMessageText = '''
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a="http://www.w3.org/2005/08/addressing">
   <s:Header>
      <a:Actio s:mustUnderstand="1">http://cai.dataaction.com.au/Cai/Member/New/2011_04_15/MemberContro...
         CreateNewPersonalMemberResponse</a:Action>
   </s:Header>
   <s:Body>
      <CreateNewPersonalMemberResponse xmlns="http://cai.dataaction.com.au/Cai/Member/New/2011_04_15/">
         <CreateNewPersonalMemberResult xmlns:b="http://schemas.datacontract.org/2004/07/DataAction.Cai.Wcf"
            xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
            <b:ReceiptNumber>8071296</b:ReceiptNumber>
            <b:RequestId>d9da9951-1a6f-4f49-bfc0-9d00fc44f491</b:RequestId>
            <b:Body xmlns:c="http://cai.dataaction.com.au/Cai/Member/New/">
               <c:MemberNumber>22562764</c:MemberNumber>
            </b:Body>
         </CreateNewPersonalMemberResult>
      </CreateNewPersonalMemberResponse>
   </s:Body>
</s:Envelope>

'''
import wslite.soap.*
import wslite.http.*
def httpRequest = new HTTPRequest()
def httpResponse = new HTTPResponse()
def soapEnvelope = new XmlSlurper().parseText(soapMessageText)

response = new SOAPResponse(httpRequest:httpRequest,
                            httpResponse:httpResponse,
                            envelope:soapEnvelope,
                            text:soapMessageText)

print response.envelope.Body.CreateNewPersonalMemberResponse.CreateNewPersonalMemberResult.Body.MemberNumber
```

Related articles

- How to mix standard delivery functions with custom groovy processing
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating new form tasks in a Groovy delivery service
- Creating anonymous tasks via Groovy service

Managing multiple service endpoints and credentials for external service calls

Unknown macro: 'redirect'

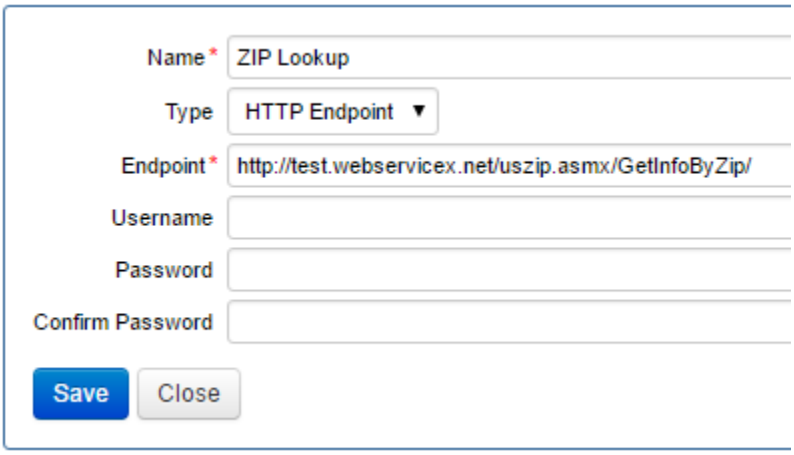
As we create services in TM that integrate with external sources via Web Service or REST we typically find that the external service endpoint and credentials change between TM environments (dev, test, prod). Or alternatively, you may have created a mock service endpoint for use during early development phases and want the ability to easily switch between the mock service and the genuine service for debugging and to verify expected results. This article describes the best practice approach to managing these different endpoint configurations using Service Connections.

Creating Service Connections

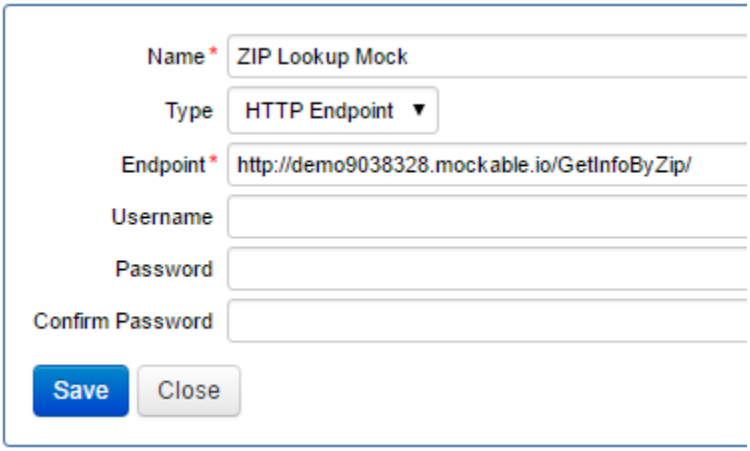
The best practice approach to managing multiple endpoint configurations is to utilize Service Connections. Lets look at an example.

Say we have a requirement to build a Dynamic Data service for use by forms that takes a US Zip Code and returns the City, State, Area Code and Time Zone. This service will call out to an external API to retrieve this information. Assume the external service provider has exposed a test service for use in dev /test environments which we will use until we deploy to the production server.

1. When creating the Service Connection for this mock endpoint I use the generic HTTP Endpoint type, then enter the name and endpoint URI. While this service does not require authentication credentials, I could just as easily add them if they were required.

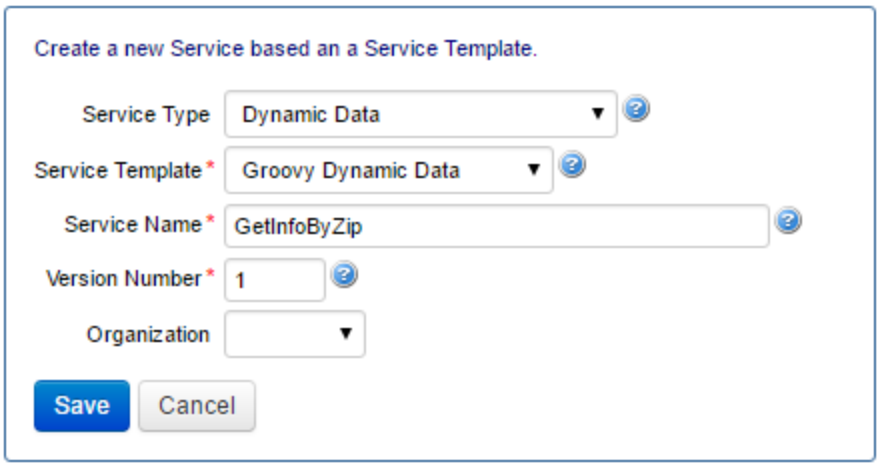


2. As we are developing this service we want to access a mock endpoint to control the response format and verify the behavior of my service, and so after building a mock service using on a mock platform (e.g. mockable.io) I can create a new service connection that points to it.

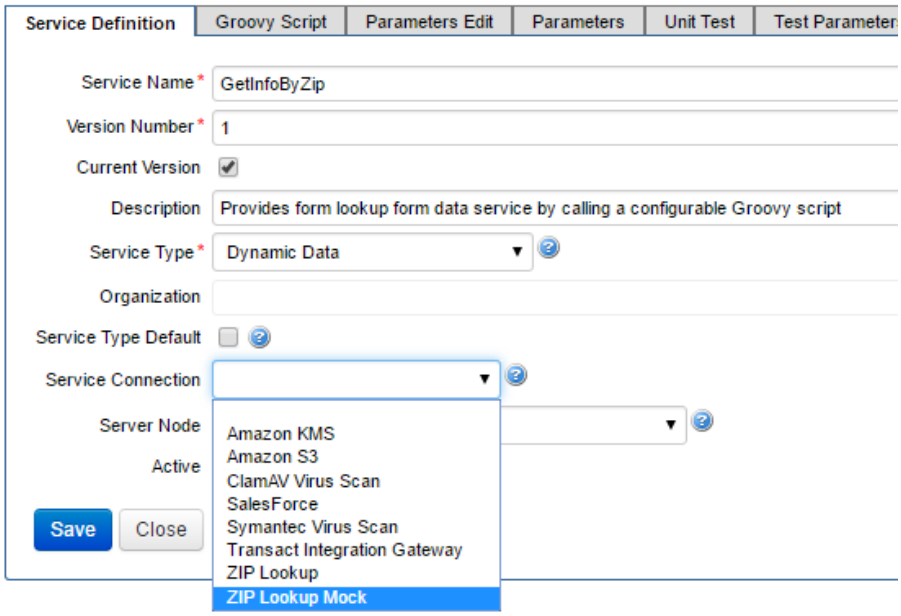


Using Service Connections in Groovy Services

1. To facilitate the dynamic data calls from the form we need to create a new service of type 'Dynamic Data', using the Groovy Dynamic Data template.



2. When configuring the service we can select the Service Connection we wish to use. We will start with the ZIP Lookup Mock endpoint.



3. Now in our Groovy Script we can retrieve the service connection from the service definition object and validate that it is configured correctly.

```
// Ensure that a Service Connection is selected
def serviceConnection = serviceDefinition.connection
if(!serviceConnection){ throw new IllegalArgumentException("No Service Connection defined for this
service.") }

// Retrieve the base endpoint URL from the service connection.
def endpoint = serviceConnection.endpointValue
if(!endpoint){ throw new IllegalArgumentException("No endpoint defined for the service connection.") }
```

4. Having a valid endpoint to work with we can now setup the remote request, apply basic authentication credentials if present, execute the call before marshaling the response.

```
// Create the remote request
def remoteRequest = new GetRequest(endpoint)
// If auth details provided, set them in the remote request
if(serviceConnection.username){
  remoteRequest.setBasicAuth(serviceConnection.username, serviceConnection.password)
}

// Now execute the remote call
remoteRequest.setParams(["USZip": zipInput])
def response = remoteRequest.execute()
```

5. At any time we can switch between the mock service and the genuine service by selecting the appropriate Service Connection in the Service Definition.

Full Groovy Script

```
/* Provides form lookup form data service by calling a configurable Groovy script.

   Script parameters include:
       form : com.avoka.fc.core.entity.Form
       request : javax.servlet.http.HttpServletRequest
       submission : com.avoka.fc.core.entity.Submission
       serviceDefinition : com.avoka.fc.core.entity.ServiceDefinition
       serviceParameters : Map<String, String>

   Script return:
       JSON string data value to be bound into the form data model
*/
import com.avoka.core.groovy.GroovyLogger as logger
import com.avoka.component.http.GetRequest
import groovy.json.JsonBuilder
import groovy.util.XmlSlurper

def zipInput = "" + request.getParameter('zip')
logger.info "Zip: " + zipInput
if(!zipInput){ throw new IllegalArgumentException("Missing input parameter - zip") }

// Ensure that a Service Connection is selected
def serviceConnection = serviceDefinition.connection
if(!serviceConnection){ throw new IllegalArgumentException("No Service Connection defined for this service.") }

// Retrieve the base endpoint URL from the service connection.
def endpoint = serviceConnection.endpointValue
if(!endpoint){ throw new IllegalArgumentException("No endpoint defined for the service connection.") }

// Create the remote request
def remoteRequest = new GetRequest(endpoint)
// If auth details provided, set them in the remote request
if(serviceConnection.username){
  remoteRequest.setBasicAuth(serviceConnection.username, serviceConnection.password)
}

// Now execute the remote call
remoteRequest.setParams(["USZip": zipInput])
def response = remoteRequest.execute()

// check HttpResponse status code
if (response.status == 404) {
  // object not found
  throw new RuntimeException('Could not find the requested service: ' + response.statusLine)
} else if (response.status != 200) {
  throw new RuntimeException(response.statusLine)
}

logger.info response.textContent

// Service returns XML so lets parse it for a JSON response
def parsed = new XmlSlurper().parseText(response.textContent)
def childNodes = parsed.Table.children().iterator().toList()
def responseValues = childNodes.collectEntries( [:] ) {
  [(it.name().toLowerCase()): (it.text())]
}

logger.info responseValues

// Build the JSON response
def serviceResponse = new JsonBuilder(responseValues).toPrettyString()
logger.info "Response: " + serviceResponse
```

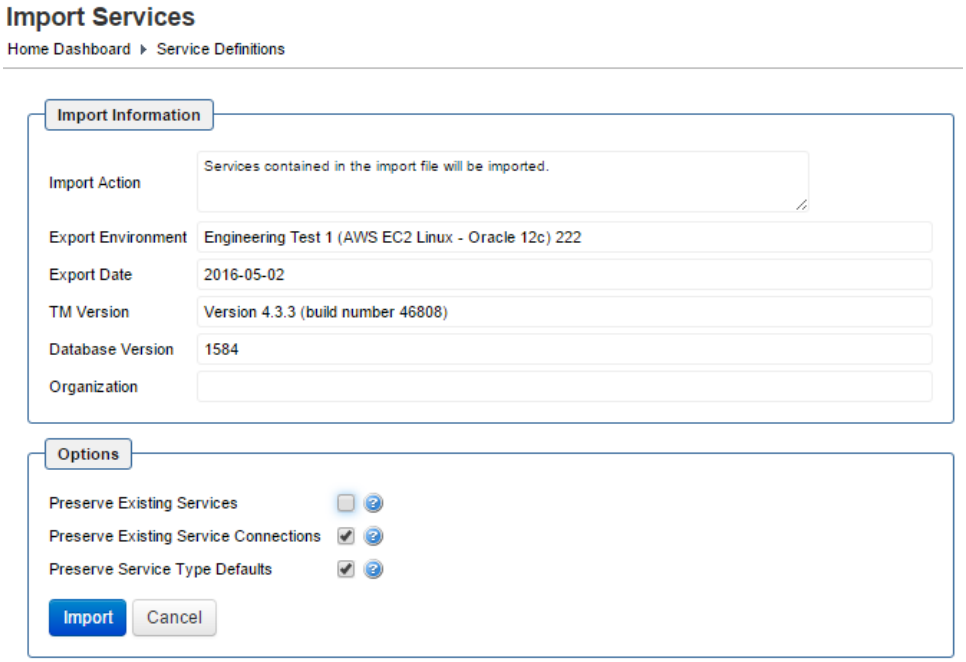
```
return serviceResponse
```

Deployment Practices

So what happens when we want to push this service into another environment with potentially different endpoint configurations? The following practices will help make this a very seamless process.

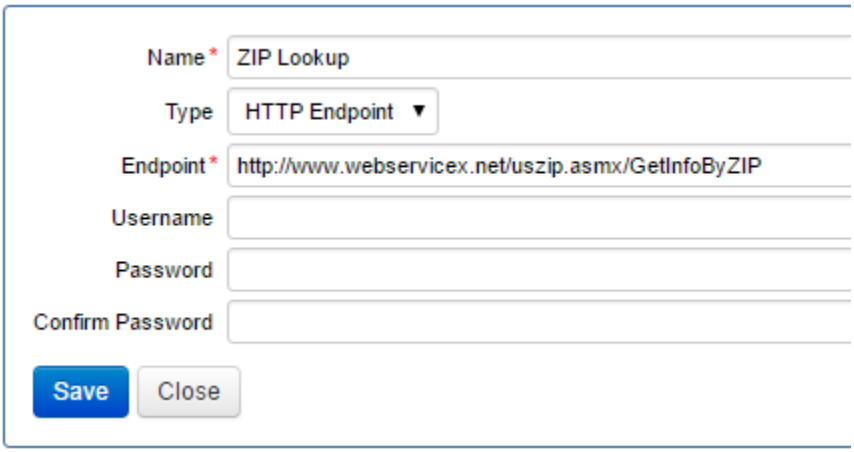
1. **Ensure the genuine service connection is selected before export.**
   When exporting a service from one environment to another, if a Service Connection is selected in the Service Definition it will also be exported into the archive file and available when you go to import the archive to another environment. So if you are using multiple endpoints in the development environment, ensure that you have the appropriate service connection selected when you go to export, i.e. the one you want to be deployed into the archive. It is unlikely that you will want to deploy the mock service to other environments, so usually you will want to select the genuine service.

2. **Import with 'Preserve Existing Service Connections' selected.**
   When importing a service archive into the target environment you are offered several import options. Check the help against each of these options and be aware how they impact the import behavior. Of most relevance to this article is the 'Preserve Existing Service Connections' option. If the import archive contains a Service Connection with the same name (and version) as an existing Service Connection in the target environment, it will not get imported if this option is selected. If no Service Connection exists with the same name then it will always be created. Typically, you will want this option enabled (which it is by default) so that changes to the endpoint configurations do not get overwritten each time you import.



3. **Use consistent naming of Service Connections between environments.**
   Ensuring that you use consistent naming of Service Connections between environments will avoid any manual reconfiguration each time you import a service from one environment to another. It is entirely up to you whether you per-configure the Service Connections in each environment or wait until the Service Connection is imported into each environment and modify the configurations at that point. In production environments you

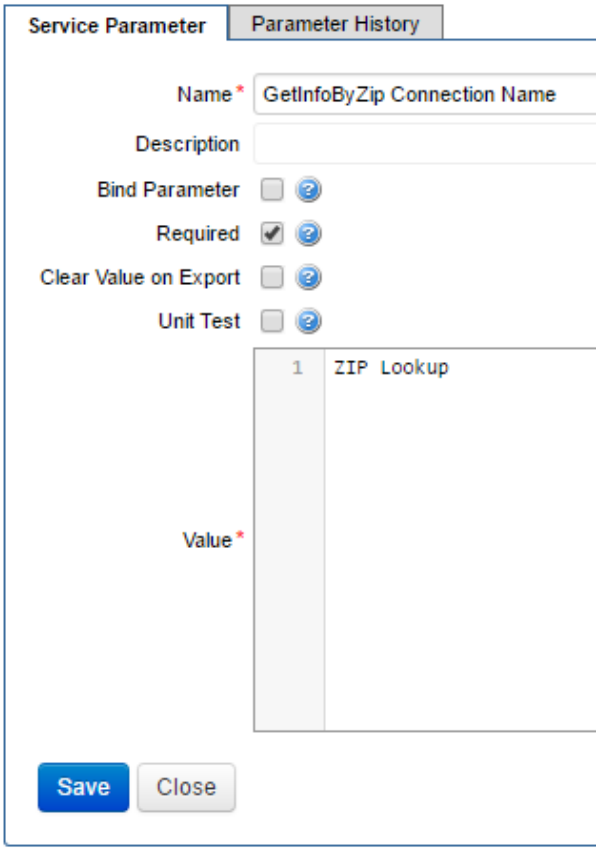
can configure the endpoint to point to the live endpoint with production credentials (if required) and if using the practices above, you can have confidence that these endpoint configurations will be maintained between import events.



Calling Multiple Endpoints in a Single Service

The practices detailed above assume that your groovy service is only required to call a single remote service, but what if it is required to call several remote services in a single execution?

1. To handle this type of scenario you will need to create a service parameter for each of the remote services, that stores the name of the Service Connection to use.



2. Within your groovy script you can retrieve each service connection by name using the ServiceConnectionDao class and then execute your remote call as normal.

```
// Check the endpoint parameter is defined
def zipConnectionName = serviceParameters.get('GetInfoByZip Connection Name')
if(!zipConnectionName){ throw new IllegalArgumentException("Missing parameter value for: GetInfoByZip
Connection Name")}
```

```
// Ensure that a Service Connection is selected
def serviceConnection = new com.avoka.fc.core.dao.ServiceConnectionDao().getServiceConnectionForName
(zipConnectionName)
if(!serviceConnection){ throw new IllegalArgumentException("No Service Connection defined for this
service.") }
```

Note that where Service Connections are referenced by parameter values in this manner they will not be exported into the service archive and must be manually created in other environments.

Related articles

- Managing multiple service endpoints and credentials for external service calls
- Groovy Services
- Understanding and integrating with Docusign
- Importing a JavaScript library into a Composer form
- XML Binding in Composer Forms

Developing a Dynamic Data Lookup

Unknown macro: 'redirect'

Introduction to Dynamic Data

Dynamic Data Services was introduced as a mechanism to provide back-end querying capability to Composer generated forms. The Dynamic Data service is a capability that is initiated in a Composer generated form, and supplied by the Transaction Manager web application. This extends the reach and reuse of data and functionality to users and customers, saving time and money, while improving the end user experience in a secure and efficient way.

The Composer widget maintains its focus on end-user ease-of-use, while providing amazing power and flexibility to the Form filling experience. The widget can appear as a Button or a Link and includes a Test facility so you can mock up the return data in Composer before publishing to Transaction Manager.

On the Transaction Manager side, a simple to create Service Definition includes all the parameters required to expose a Groovy script which defines the activity of the Dynamic Data Service. Groovy is a language with a syntax very familiar to ActionScript and Javascript programmers alike. You can find out more about Groovy here http://groovy.codehaus.org/, but this article assumes no prior knowledge of Groovy syntax or structure.

Handling Data

Dynamic Data can cater for single and multiple record return values, and as the Form designer, you need to know when to use the right output structures. Dynamic Data requires only the name of the Dynamic Data Service defined in Manager, but in reality needs an Input Block and an Output Block of widgets to be useful as a look up service. Data passed between the Composer generated SmartForm and Transaction Manager (and indeed the test data) is structured using the JSON format, a structure almost universally available on every platform known to mankind. While a complete description is beyond the scope of this article, JSON uses a simple key, value naming arrangement that can support deep nested structures and arrays.

Building the Composer Form

The example form referenced in this article uses an 'ABN Lookup Dynamic Data Service'.

Input Data

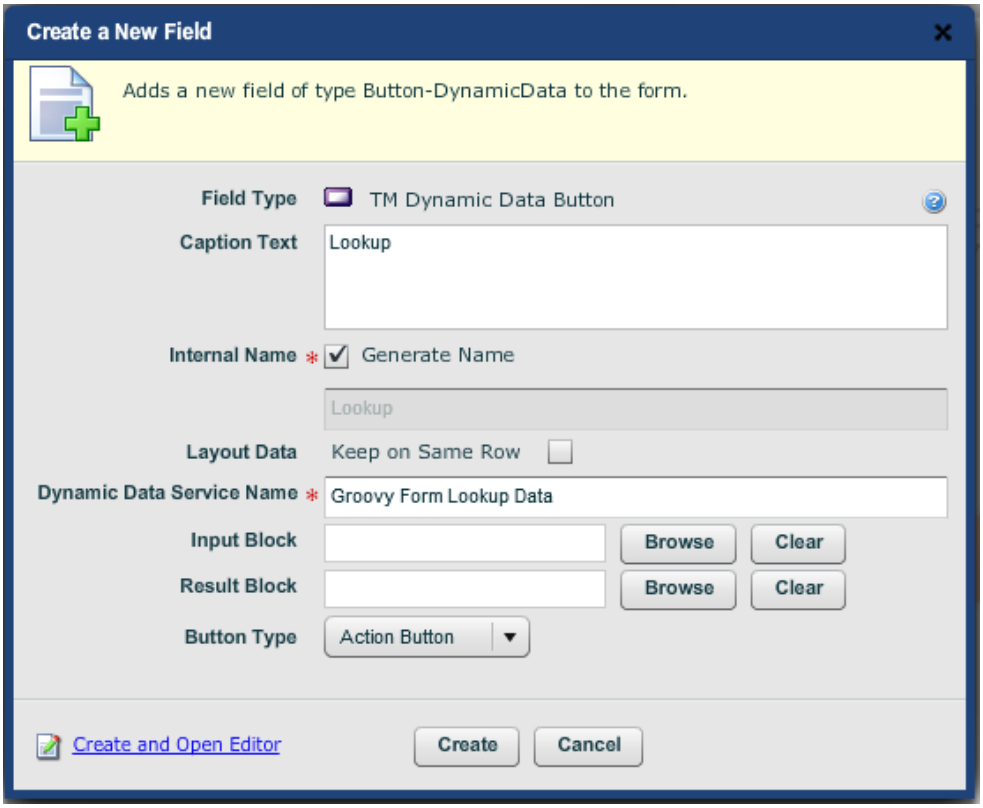
The example form takes an ABN number, collected using the ABN widget. The widget is placed inside a Block and configured with whatever other parameters are needed, like mandatory rules and visibility. The Block can be called anything you like and needs no other special configuration. Note that any fields in the Input Block *will* be automatically bound and transmitted to Transaction Manager as input data. This doesn't mean that your Dynamic Data Service has to use them, just be aware that they are there.

Output Data

This example returns one field called CompanyName to a standard block. Note that you don't have to have your output fields in a block of their own, they can be in a block with other fields. We'll see later how the binding of the output data occurs.

The Lookup Widget

In Composer, locate the TM Dynamic Data Button widget, typically found in the Transaction Manager section of the widget palette, and drop it on the form where you need it to appear. The following dialog will appear.



Click the Browse buttons next to the Input and Result block entry areas to locate the parent Blocks of the Input and Output fields you defined earlier. Finally change the Button type as desired.

You don't need to change the TM Dynamic Data Service Lookup Name just yet, we'll be using the test mode to prove out our scenario first and then migrating it to Transaction Manager a little later on.

Save the dialog.

Testing the Lookup

Our next step will be to provide some test data so we can be assured of the JSON data format required to prove our scenario. Bring up the property editor for the TM Dynamic Data widget. In the Data section, check the Test Mode flag and enter the JSON formatted return value into the Test Data text area.

The format to use when formulating JSON notation is as follows:

```
{"KeyA":"ValueA","KeyB":"ValueB"}
```

which equates to:

KeyA = ValueA
KeyB = ValueB

This might appear in Composer as:

```
{"FieldNameA":"Field Value","FieldNameB":"Field Value"}
```

There are a few gotchas to be aware of:

Remember to use Field Names as the key, not labels. Don't be fooled into thinking the hierarchy uses field names, you'll need to look up the field names in the property editor for the field (right click the field in the Structure panel on the left, and select "Edit Properties...").
Values need to be formatted as appropriate for the field.

Here is an example of the JSON test data that I formulated for my specific test case. RegistrationNumber, VehicleDescription and RegistrationRenewalD are the three output fields of type text, text and date respectively.

```
{"CompanyName":"Avoka technologies"}
```

When the Dynamic Data Service component receives the results data from either test or Transaction Manager, it binds the associated value to the appropriate field by means of its name. It's this name binding that makes it possible to specify an output block that has other fields that aren't part of the result set.

After saving the properties and previewing the form, you should see your test data appear in output fields. If your data does not appear, or you get errors you'll need to resolve these before migrating the test data to Transaction Manager.

Building the Transaction Manager Dynamic Data Service.

Transaction Manager exposes a single endpoint for Dynamic Data Services, but determines which service to use by a single hidden field included in the input. Manager also secures the use of the Dynamic Data Service by embedding a hash key into the form when published to thwart possible misuse of the service data by unauthorised clients.

Dynamic Data Service Definitions use a combination of Service Definition configuration values and parameters to configure the service.

Copying an existing Dynamic Service Definition

Log in to Transaction Manager instance and select the System > Service Definitions menu item. Select Dynamic Data from the list of Service Types and click the search button. This gives you a list of the existing Dynamic Data Services already developed and you should go through some to get a feel for how they work. For the purpose of this example we're going to go ahead and copy an existing service and then modify it to suit our needs.

On the Service Definitions page, click the Copy button. On the Copy Service Definition page, select Dynamic Data as the Service Type, "Groovy Dynamic Data" as the Existing Service Name and enter a new name for the service.

Modifying the Dynamic Service Definition

Now that the service exists, technically we could call it from a published SmartForm but the output data in the script isn't tailored to our needs, so we'll fix that up first. Edit the Service Definition for the new Dynamic Data service and click the Service Parameters tab. Click on the groovyScript parameter name, this will invoke the text area editor which has a handy code syntax highlighter to help you spot errors.

The script below is used by the 'ABN Lookup Dynamic Data Service' on Transaction Manager Demo. It uses the wslite library to call the Australian Government's ABN lookup service.

```
import wslite.soap.*

import net.sf.json.*

import net.sf.json.xml.*

def abn = request.getParameter('abn')

def client = new SOAPClient('http://abr.business.gov.au/abrxmlsearch/AbrXmlSearch.asmx')

def response = client.send(

    """<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:abr="http://abr.business.
gov.au/ABRXMLSearch/">

   <soapenv:Header/>

   <soapenv:Body>

      <abr:ABRSearchByABN>

         <!--Optional:-->
```

```
         <abr:searchString>""" + abn + """</abr:searchString>
         <!--Optional:-->
         <abr:includeHistoricalDetails>N</abr:includeHistoricalDetails>
         <!--Optional:-->
         <abr:authenticationGuid>avoka-7c4df64c-aa6c-41e5-abf1-0ffe200a8833</abr:authenticationGuid>
      </abr:ABRSearchByABN>
   </soapenv:Body>
</soapenv:Envelope>"""
)
String organisationName = response.ABRSearchByABNResponse.ABRPayloadSearchResults.response.businessEntity.
mainName.organisationName;
def jsonResponse = new JSONSerializer().toJSON(["CompanyName": organisationName]);
return jsonResponse;
```

The important parts of the code to take note of are:

```
def abn = request.getParameter('abn')
```

This line searches the input block in the composer form for a field called 'abn', and saves it's value to a variable in the script called abn.

```
def jsonResponse = new JSONSerializer().toJSON(["CompanyName": organisationName]);
```

This line takes the organisation name (saved in the variable 'organisationName', and inserts it into a JSON object, using "CompanyName" as a key. This will save the value returned by this script to a field named 'CompanyName' in the output block in the composer form.

Linking the Dynamic Data service in Transaction Manager to the Composer form.

The Composer form specifies the Dynamic Data service it will use by name, in the 'Dynamic Data Service Name' field. When publishing the form, remember to uncheck the 'Test Mode' button, or the service will not be called.

A Final Word - Testing Testing Testing

OK, so there's a good chance that when you got it all hooked up it didn't work. For debugging I highly recommend FireFox and Firebug or Chrome and it's Dev Tools for analysing page behavior. In the screenshot you'll notice there's a section at the bottom of the page where you can see the POST request to the Manager servlet that services the Dynamic Data Services requests. The response from that POST includes the data as formatted in my Test Mode data. Yours should too.

Bonus Points - Repeating blocks of results

Thankfully, the Composer developers catered for multiple results sets by allowing data to be returned in a JSON array. This is easily accomplished by wrapping the JSON response data in an array spec and separating records with commas, like so:

```
[{"record1":"value1"},{"record2":"value2"}]
```

On the Composer side, the output block must be Repeatable Content to get the Repeatable-Block field definition. Again, field binding is done by name with each record in the JSON array creating a new repeat section. Do be careful if switching between the two types (Repeating and Standard Blocks) since the Output Block specification differs depending on the type. In my example the Output Block specification for a single result is:

```
../VehicleDetails
```

If I were to change the block to Repeatable Content in response to some change in development, I must ensure my Output Block specification is updated and appears as follows:

```
../VehicleDetails[*]
```

Conclusion

Dynamic Data Services is a very important and strategic feature for Transact providing a mechanism to further enrich the user experience by including interactivity between Composer and Transaction Manager. The range of possible applications for this capability are untold with the flexibility and power in the current design.

References

1. Groovy - http://groovy.codehaus.org/
2. JSON - http://www.json.org/

Related articles

- Developing a Dynamic Data Lookup
- How to create a Dynamic Data lookup using test data
- Repeating and Nested data in Dynamic Data service
- Data Lookup

Invoking Dynamic Data Service from Groovy Service programmatically

Unknown macro: 'redirect'

Use Case

A Dynamic Data service is normally used in a Composer Form to perform a lookup or ajax calls to get information into the form from various integration endpoints. You may come across some situation where you want to leverage existing dynamic data service in your groovy script. For example, I have come across a situation where I have unformatted address string that I want to validate against Veda address service. Avoka Transact provides two dynamic data services for Veda address lookup integration. "Veda Address Search" service is used for type ahead address lookup to return possible matches and "Veda Address Detail" service is used to get more granular data for selected/provided address such as Unit No, street no, street name, street type, suburb, etc. I wanted to leverage existing "Veda Address Detail" service for two purpose. One to check if address string is a valid address in Veda address database (that is confirmed by if DPID is returned in the response) and second to get granular information from unformatted address string.

Solution

Firstly you will need to obtain a reference to the service with ServiceLocator class and cast it to IDynamicDataService to invoke required service by its name. As Dynamic Data services are intended to work with AJAX requests, it needs HTTPServletRequest object to invoke service. In our case, we cannot use originating request object as we need to pass our own request parameter (i.e. address string). As the request object is mutable and cannot add new request parameters to it, we need to create a mock request to fulfil our need. MockHttpServletRequest class can help us with that. See below for actual implementation.

```
import com.avoka.fc.core.service.ServiceLocator;
import com.avoka.fc.core.service.form.IDynamicDataService;
import org.springframework.mock.web.MockHttpServletRequest;
import groovy.json.JsonSlurper;

def concatedAddress = "300 George St Sydney NSW 2000";

def vedaAddressDetailsService = (IDynamicDataService) ServiceLocator.getServiceForName("Veda Address Detail");
def mockRequest = new MockHttpServletRequest();
mockRequest.addParameter("addressString" , concatedAddress);
def vedaResponse = vedaAddressDetailsService.getFormData(mockRequest, null, null);
def addressJson = new JsonSlurper().parseText(vedaResponse);
```

Related articles

- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services
- Validating Form Data Against a Schema in Groovy

Validating Form Data Against a Schema in Groovy

Unknown macro: 'redirect'

There are some scenarios where you need to validate submitted XML against an XML Schema.

This below code snippet is an example of this in Groovy

```
import javax.xml.XMLConstants
import javax.xml.transform.stream.StreamSource
import javax.xml.validation.SchemaFactory
import org.xml.sax.SAXException

// submissionString is a String variable containing the submitted XML

// the schema content is stored as a parameter on the Service Definition
def xsd = serviceDefinition.getServiceParameter('schema').getValue()


def factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI)
def schema = factory.newSchema(new StreamSource(new StringReader(xsd)))
def validator = schema.newValidator()
validator.validate(new StreamSource(new StringReader(submissionString)))
```

Related articles

- How to mix standard delivery functions with custom groovy processing
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating new form tasks in a Groovy delivery service
- Creating anonymous tasks via Groovy service

Accessing Form properties in Groovy Services

Unknown macro: 'redirect'

It can be the case that the operation of a Groovy service (prefill, delivery, dynamic data, etc.) must be tailored slightly for specific forms. One way of storing the data for these customisation is at the form level, in form properties. Once the data has been stored in these form properties, it must obviously be accessible from within the service that requires the data.

Accessing properties from within a Groovy script can be achieved by using the PropertyValueDao class. Using this class you can make calls to the 'getPropertyValue' method, providing the name of the property you're looking for and the form object it's associated with. Most Groovy services in TM (Prefill, Dynamic Data, Receipt Number, Form Saved, Submission Complete) have a predefined reference to the form object, in variable called 'form'. For these services properties for that form can be accessed with the following code:

```
import com.avoka.fc.core.dao.PropertyValueDao

PropertyValueDao propertyValueDao = new PropertyValueDao()

def theValue = propertyValueDao.getPropertyValue("The Property Name", form)
```

Delivery Processes on the other hand don't have this existing reference to the form object, but can acquire one easily from the 'submission' object as in the following example:

```
import com.avoka.fc.core.dao.PropertyValueDao

PropertyValueDao propertyValueDao = new PropertyValueDao()

//Notice 'form' has been substituted for 'submission.getForm()'
def theValue = propertyValueDao.getPropertyValue("The Property Name", submission.getForm())
```

Related articles

- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services
- Validating Form Data Against a Schema in Groovy

Creating anonymous tasks via Groovy service

Unknown macro: 'redirect'

A task is a form that has been prefilled with data and then assigned to a user. The typical use case for a task is: A user opens and completes a blank form. On submission, a task is created for another user, containing the original user's submitted data. The second user then reviews/modifies the data before submitting for processing.

Tasks can be assigned to TM users, in which case they are accessed when that user logs into a portal, or they can be assigned anonymously, in which case a saved form is generated, and the URL is sent via email. Anonymously assigned tasks can be created in a groovy script by using the SubmissionTaskService object.

The SubmissionTaskService API can be accessed at <TM server>/manager/admin/javadoc/com/avoka/fc/core/service/SubmissionTaskService.html

The following groovy script can be used to assign anonymous tasks:

```
import com.avoka.fc.core.entity.*
import com.avoka.fc.core.dao.*
import com.avoka.fc.core.service.SubmissionTaskService
import com.avoka.fc.core.service.SubmissionTaskService.SavedFormParam
import com.avoka.fc.core.util.PortalUtils
import com.avoka.core.util.StringTemplate
import com.avoka.fc.core.service.EmailService


def formUrl = ""

// Setup the saved form parameter DTO
def param = new SavedFormParam()

// Setup the form parameter
def form = DaoFactory.getFormDao().getFormByFormCode( serviceParameters.FormCode ?: "" )
if(!form){ throw new Exception("Form Code is invalid, form not found.") }
param.form = form

// Setup the portal parameter
// uses the portal provided, if none provided, used the one configured as a service parameter.
def portalName = serviceParameters.PortalName ?: ""
def portal = DaoFactory.getPortalDao().getPortalByName( portalName )
if(!portal){ throw new Exception("Portal Name is invalid, portal not found") }
param.portal = portal

// Check the email serviceParameters
if(serviceParameters.SendEmailFlag){
    if(!serviceParameters.ContactEmailAddress){ throw new Exception("Contact Email Address must be provided"); }
    if(!serviceParameters.EmailSubject){ throw new Exception("Email Subject must be provided"); }
    if(!serviceParameters.EmailMessage){ throw new Exception("Email Message must be provided"); }
}

// taskSubject
param.taskSubject = ""
// taskMessage
param.taskMessage = ""

// saveChallengeAnswer
param.saveChallengeAnswer = ""
// submissionXml
if(!serviceParameters.FormXml){ throw new Exception("Form Xml must be provided") }
param.submissionXml = serviceParameters.FormXml

// Create the task
def submissionTaskService = new SubmissionTaskService()
def submission = submissionTaskService.createAnonymousSavedForm(param)

// Task created, send notification email if required
//=====================================================================
if(serviceParameters.SendEmailFlag){
    def emailService = new EmailService()

    // Get the URL to resume the saved form
    formUrl = PortalUtils.getFormSavedUrlWithReferenceNumber(portal, submission)
```

```
    // Create data map for use in Velocity templates
    def dataMap = [:]
    dataMap.put("formUrl", formUrl)

    def bodyStringTemplate = new StringTemplate(serviceParameters.EmailMessage)
    def emailBody = bodyStringTemplate.merge(dataMap)

    emailService.sendMessage(
        serviceParameters.EmailSubject,
        emailBody,
        null,
        serviceParameters.ContactEmailAddress,
        "",
        null)
}

return formUrl
```

Service Parameters:

In order for the script to run, it will need the following service parameters configured:

ContactEmailAddress - <String> the email address that the URL of the created task is sent to
EmailSubject - <String>
FormCode - <String> The form code of the form used to generate the task. This form code must exist on the TM server.
FormXml - <String> The xml data used to prefill the generated task
PortalName - <String> Then name of the portal that the form belongs to.
SendEmailFlag - <Boolean> Select 'True' to send the email to the assigned contact
EmailMessage - <String> A velocity template of the email body HTML. Velocity templates contain HTML with placeholders for dynamic content. In this case, a placeholder must be configured for 'formUrl', which will be populated with the link to the created task.

An example of an email message velocity template is:

```
<html>
    <head>
        <style type="text/css">
            /* custom CSS here */
            body {
                font-family: Verdana, Arial, Helvetica, sans-serif;
            }
        </style>
    </head>
    <body>
        <!-- custom Content here -->
        <div>
            This is my message.<br>
            Form Link: <a href="$formUrl">$formUrl</a>
        </div>
    </body>
</html>
```

Related articles

- How to mix standard delivery functions with custom groovy processing
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating new form tasks in a Groovy delivery service
- Creating anonymous tasks via Groovy service

Creating new form tasks in a Groovy delivery service

Unknown macro: 'redirect'

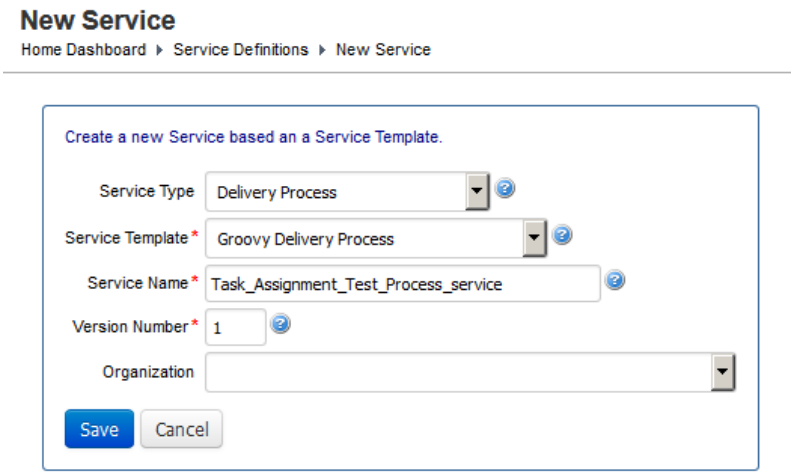
A Form can be configured to create a Form Task on Submission via configuration of a Delivery Process which is then assigned to the Form.

Configuring a Delivery Process to create a new Form Task

Creating a Delivery Process

The System -> Service Definitions screen is used to create a new Delivery Process.

In this instance, we create a new Groovy Service Delivery Process



New Delivery Process Groovy Service Definition

The Service Definition's Groovy Script service parameter is configured with Groovy Script to create a new Form Task using the Task Server createFormTask method.

The following script example shows the simplest example possible of the use of the TaskService to assign a task to a user based on the same form and submission data from the initial user submission. The task is assigned to a single user for review. In real world scenarios a different form would be used for the review or the submission Xml would be manipulated to set the form into review mode when rendered from a task.

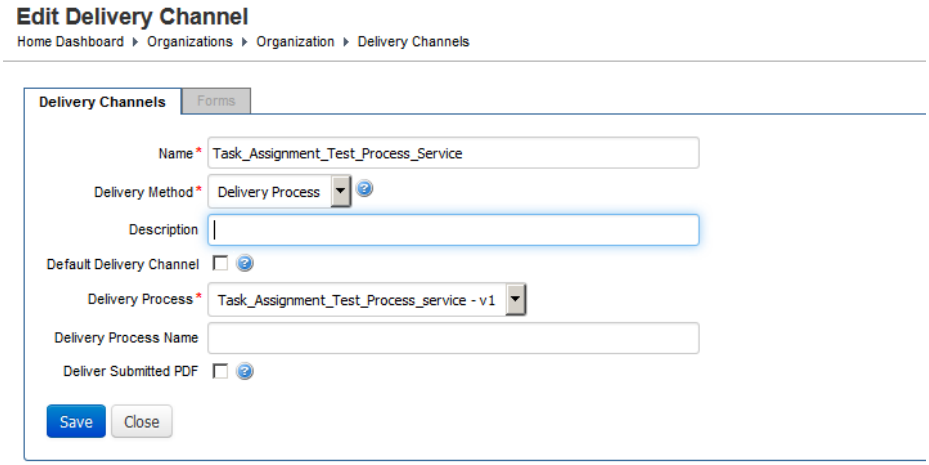
Simple assign task example

```
import com.avoka.fc.core.service.ServiceFactory
import com.avoka.fc.core.dao.DaoFactory


def user = DaoFactory.getUserAccountDao().getUserAccountForLogin("reviewUser");

def taskService = ServiceFactory.getTaskService();
def submission = taskService.createFormTask(submission.form,
                                            "Please Review",
                                            "You have a task to review",
                                            null,
                                            user,
                                            null,
                                            submissionXml,
                                            null,
                                            null,
                                            null,
                                            null,
                                            null,
```

```
                                        null,
                                        true,
                                        "Please Review",
                                        "You have a task to review",
                                        false);
```

Adding the Delivery Process to an Organisation

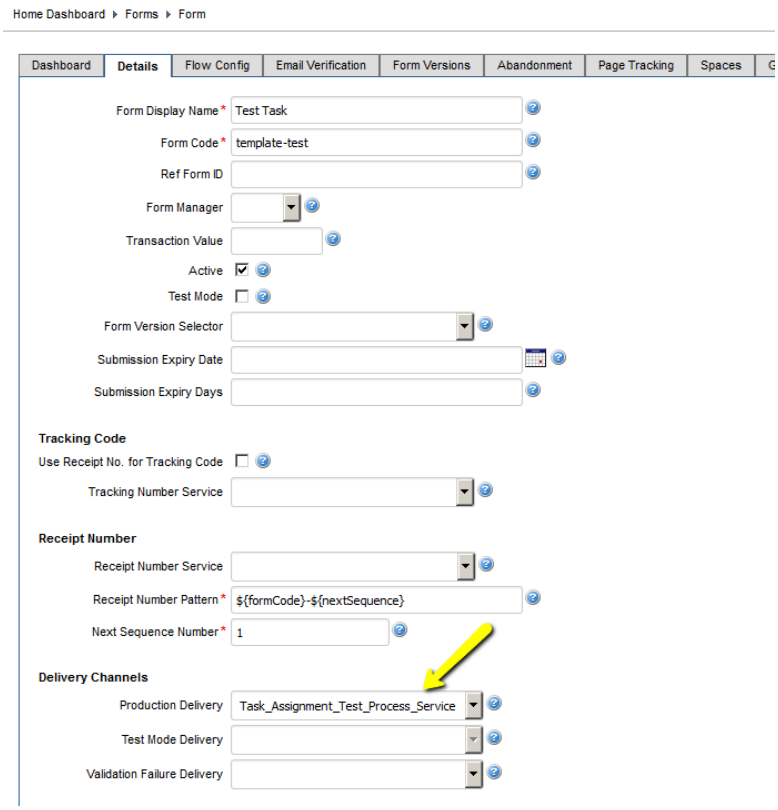
This Delivery Process method is then added to the Form's Organisation Form Delivery Detail configuration. This will enable the Delivery Process to be available on the organisation's forms configuration.



Organisation configured with a Delivery Process method

Configuration a Form's Delivery Method

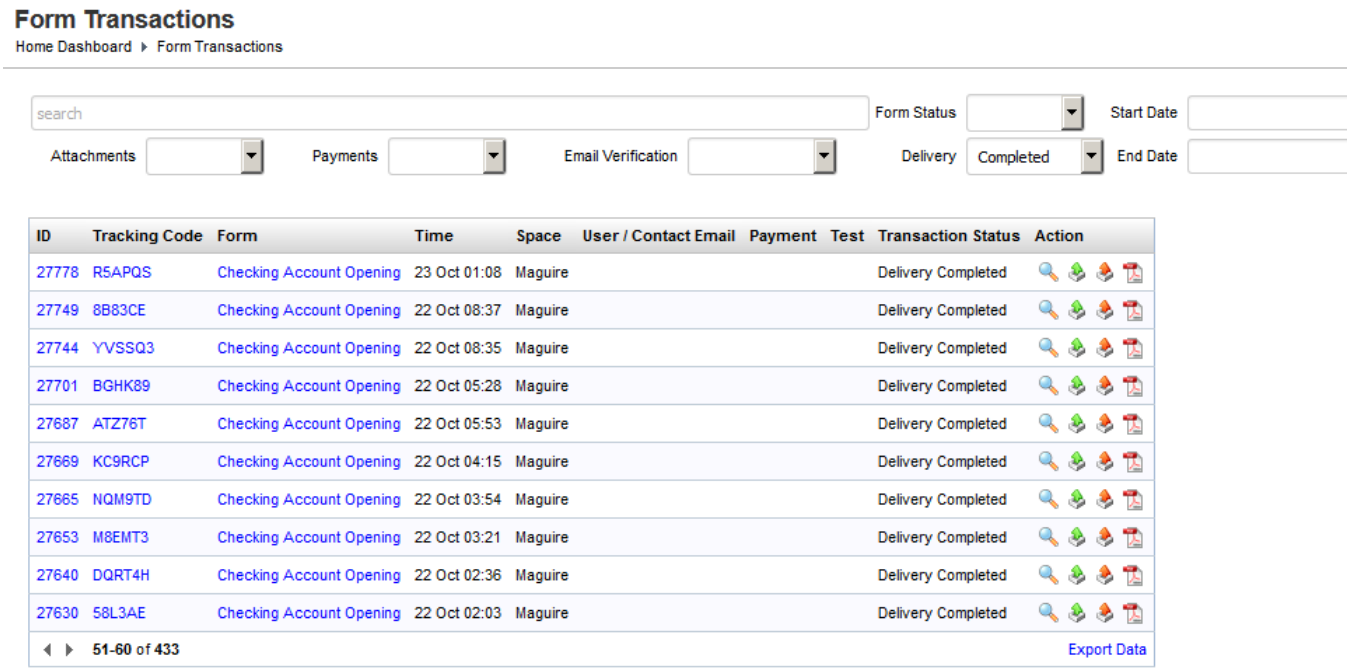
Finally a Form can have its Details section configured with the Organisation's Delivery Details for Production and Test Delivery (This is impacted by the Test Mode flag in the Details screen).



Form Details Delivery Configuration

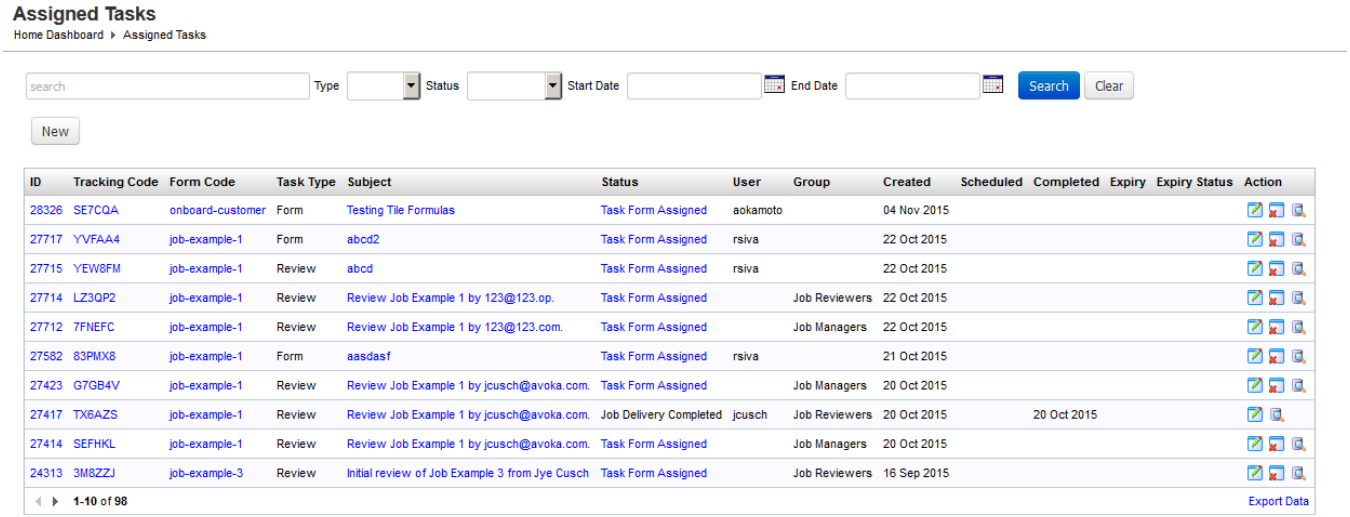
Viewing the Assigned Tasks

Once a Form is submitted, the Transaction can be viewed from the Operations -> Form Transactions screen. This screen will show the Transaction Status which will include the Delivery Status, and also allow Delivery to be retried.



Form Transactions with Delivery Complete transactions

The Operations -> Assigned Tasks screen displays the lists of Assigned Tasks, the assignee and access to the transaction XML.



Assigned Tasks

Related articles

- How to mix standard delivery functions with custom groovy processing
- Accessing the service parameters of another service in Groovy
- Creating new form tasks in a Groovy delivery service
- Creating anonymous tasks via Groovy service
- Validating Form Data Against a Schema in Groovy

Reloading all Submission Data Extracts for a form in Groovy Console

Unknown macro: 'redirect'

If a client requests new submission data extracts after a number of initial submissions have already taken place, you can add these new data extracts to the form data config and then reload the data extracts for every past submission of the form using the script below in the groovy script console.

Note: this will only work on existing xpath locations (i.e. existing fields in the form)

Groovy Script

```
import com.avoka.fc.core.dao.SchemaSeedDao
import com.avoka.fc.core.dao.FormDao
import com.avoka.fc.core.service.SubmissionService
import com.avoka.fc.core.service.ServiceFactory
import java.util.List;
import com.avoka.fc.core.dao.SubmissionDataDao
import com.avoka.core.util.XmlUtils
import org.apache.cayenne.BaseContext;
/*
Author: David Moore
Reloads data extracts for said form
*/
//Set the form code to the form you want to update the extracts.
def formCode = "PUT-FORM-CODE-HERE"

def submissionDataDao = new SubmissionDataDao();

def formDao = new FormDao();

def schemaDao = new SchemaSeedDao()

def submissionService = new SubmissionService();

def form = formDao.getFormByFormCode(formCode)

def curVersion = schemaDao.getSchemaForVersion(form.getCurrentVersion())

def submissions = form.getSubmissions()

/*
Loop submissions and update extracts
*/

if(submissions) {
  submissions.each(){
    def submissionXml = submissionDataDao.getSubmissionXmlString(it)
    // only if we have a submission XML string value
    if(submissionXml?.trim()) {
      def document = XmlUtils.parseDocumentFromString(submissionXml)
      submissionService.populateSubmissionExtractData(document, it, curVersion)
    } else {
      println "Skipping blank submission XML for submission trackingCode:${it.getTrackingCode()}"

    }
  }

}


BaseContext.getThreadObjectContext().commitChanges();
```

Related articles

- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating anonymous tasks via Groovy service
- Accessing Form properties in Groovy Services
- Validating Form Data Against a Schema in Groovy

Accessing the service parameters of another service in Groovy

Unknown macro: 'redirect'

I had an interesting requirement from a client the other day, so I thought I'd share the use case and a generic solution so that others can benefit.

The requirement was for a 'Form Prefill' service to be able to access the service parameters configured within an existing 'Delivery Process' service. The use case being that there was some common configuration between the services and that they wished to maintain these settings in a single location.

As it turns out this functionality is built into the TM Java libraries and it was fairly straight forward to implement the solution.

Start by adding the following imports to the top of your Groovy script:

```
import com.avoka.fc.core.service.IServiceDefinitionAware
import com.avoka.fc.core.service.ServiceLocator
import com.avoka.fc.core.entity.ServiceDefinition
```

Next, use the 'ServiceLocator' class to find the service with the parameters you want to access and retrieve its service definition:

```
IServiceDefinitionAware tmService = ServiceLocator.getServiceForName("The Existing Service's Name")
ServiceDefinition serviceDefinition = tmService.getServiceDefinition()
```

Finally, get the service parameter you're looking for and where necessary access the value it contains:

```
def param = serviceDefinition.getServiceParameter("The Parameter's Name");
def paramValue = param.getValue();
```

Done.

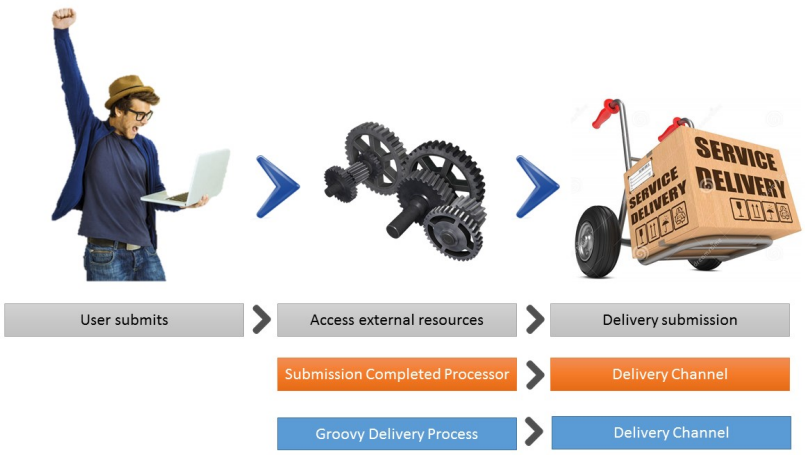
Related articles

- How to mix standard delivery functions with custom groovy processing
- Accessing the service parameters of another service in Groovy
- Reloading all Submission Data Extracts for a form in Groovy Console
- Creating new form tasks in a Groovy delivery service
- Creating anonymous tasks via Groovy service

How to mix standard delivery functions with custom groovy processing

Unknown macro: 'redirect'

More than once will you be needing to access an external resource (eg web service, file share or perhaps an external mail server) to do something on submission. Typically you would create a Groovy delivery service and do whatever needs to be done in there. Sometimes this approach is a little problematic if your submission requires a delivery method other than Groovy, like Web Service, REST or Email. It's also possible that you already have an existing delivery channel that is used for other forms, but for this one form you need to do something extra before delivery.

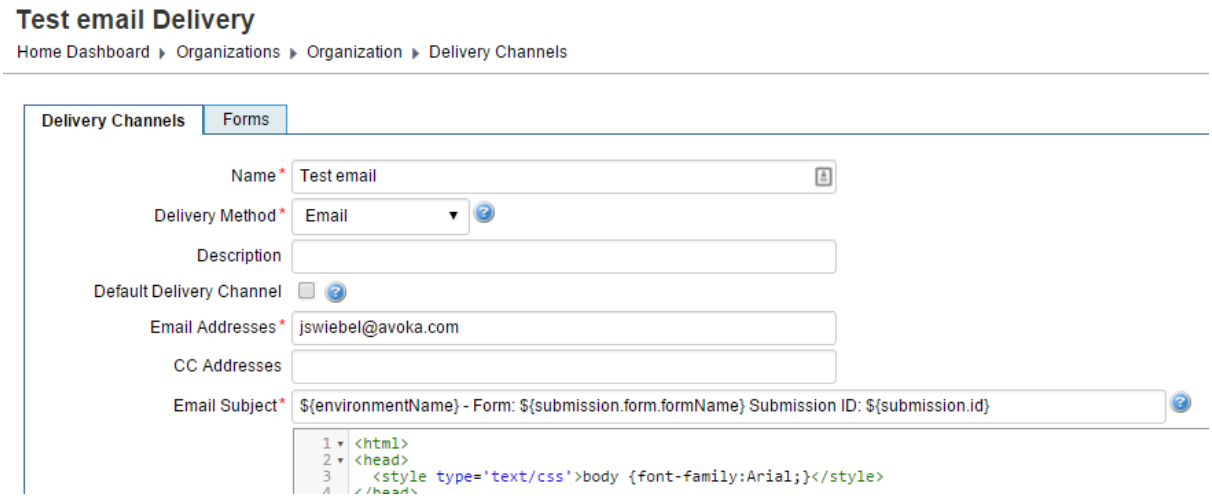


The simple solution is to add the logic to the Submission Completed processor, your external resource will be accessed first and then the normal delivery process will kick. But what if the external resource is temporarily not available, the submission completed process will fail and unfortunately cannot be retried.

This How-To guide describes how the external resources is accessed from a Groovy delivery process, and after that the process will change the delivery for that submission to the final (non-Groovy) delivery process.

Step-by-step guide

For this we assume that we already have a non-Groovy delivery channel configured, called "Test Email", using the standard config.



Now create a new Groovy delivery process, and in this you will be able to do whatever it is that needs be done using something external.

Groovy delivery Service

```
import com.avoka.fc.core.service.DeliveryResult
import com.avoka.fc.core.service.DeliveryResult.Status
import com.avoka.fc.core.dao.DaoFactory
```

```
import com.avoka.fc.core.dao.DeliveryDetailsDao
import com.avoka.fc.core.entity.DeliveryDetails
import com.avoka.fc.core.dao.ClientDao
import com.avoka.fc.core.entity.Form

// here starts the work
// step one, do your external resource stuff like call an external database
if (deliveryCheckpoint.doCheckpoint("Access External Resource", "Write db entry")) {
  try {
     // do the actual method call that does the thing
     storeToDb(.....)
     deliveryCheckpoint.completedCheckpoint("Access External Resource")
  } catch (Throwable e) {
    deliveryCheckpoint.errorCheckpoint("Access External Resource", e)
  }
}
// now make the submission available for Email delivery service
if (deliveryCheckpoint.doCheckpoint("Email Delivery", "Make submission available for email delivery")) {
  try {
    // First get a new deliveryDetails object for the client (organisation) and the name of the delivery
service you'd like to use
    Form form = submission.getForm()
        DeliveryDetailsDao deliveryDetailsDao = DaoFactory.getDeliveryDetailsDao()
        DeliveryDetails deliveryDetails = deliveryDetailsDao.getDeliveryDetailForClientAndName(form.getClient(),
"Test email")

    // Set the delivery details on the submission
        submission.setDeliveryDetails(deliveryDetails)

    deliveryCheckpoint.completedCheckpoint("Email Delivery")
    // and return a ready status so that the submission will be ready for delivery via the newly set delivery
process
        return new DeliveryResult(Status.READY)
  } catch (Throwable e) {
    deliveryCheckpoint.errorCheckpoint("Email Delivery", e)
  }
}
```

Now create a new Delivery Channel for your form's organisation, this channel needs to be of the type "Delivery Process" and make sure you configure it to use your newly created delivery Groovy process.

The last step is to change the delivery channel on your form to use the new Groovy one and not the old Test email service.

Don't try to change the submission's Delivery details object

If you try to manipulate the submission's delivery details be prepared for some funky result. Most likely the configuration of the delivery channel itself will be set in an inconsistent state. So don't do this.

Retriggering delivery

The submission will now go through the submission process twice, so be patient. Re-triggering the submission form TM will now only execute the newly email delivery process, not the original Groovy service.

Related articles

- How to mix standard delivery functions with custom groovy processing
- Accessing the service parameters of another service in Groovy
- Creating new form tasks in a Groovy delivery service
- Creating anonymous tasks via Groovy service
- Validating Form Data Against a Schema in Groovy

Log files best practice

Unknown macro: 'redirect'

There are lots of options when using Transact Manager's logs, this article is a best practice guide on how and where you should be logging when working on TM Groovy services..

Compatibility

Since	
Deprecated	

Log file Types

The three logs we will discuss in this article are :-

1. The Event log
2. The Error log
3. The Groovy Service log

The Event Log

This is where you can log important solution warning entries. The event log should never be used for general debug, info and warning logging.

Try to keep these to a minimum so that entries like "Possible fraudulent activity detected on submission 6aa71d2d118ca3e2a66e58acf822c3b6 - Veda hash check failure" are not burried in thousands like "Not entering step 25 of 43 in Jim's awesome service" (the former should of course be sending an alert email also).

The Event Log can also be used for recording errors.

The Error Log

This is where you should be logging serious errors that require some action to investigate and resolve. The error log on a well developed resilient solution should be empty.

This is where the Ops guys are going to be looking so only log here if you want them to come and talk to you!

The Groovy Service Log

The Groovy service logs were introduced with TM 4.2(?) AND contain records logging the execution of specific Groovy services when they complete successfully or in specific error cases.

This should be the primary location you should be logging. Additional actions such as links to the submission and error log may also be included.

There are parameters on your service to turn groovy logging and debug logging on/off.

The only exception is the Security Manager groovy scripts, which are not supported by GroovyLogger recording.

Log file Dos and Don'ts

Do choose the correct log file for the scenario

Do make your log entries meaningful

Do add a correlation key if multiple log entires will need to be tracked (submission key is often sufficient)

Do use try/catch blocks for known issues

Don't log any Personally Identifying (PI) data - if you really must in order to write a peice of code then make a note or create a story to remove it later

Don't log any Personally Identifying (PI) data - did I mention that already? Oh, well it is really important!

Don't be overly chatty

Other Logs

There are lots of other logs in the TM product stack such as the Apache server, access and SSL request logs. You may need to look at these when triaging issues but should not be logging to them directly.

Related articles

The TM admin guide for your version of TM.

https://<your TM server>/manager/admin/help/topics/service-development/service-logging.html

Objects, collections and JSON in Groovy

Unknown macro: 'redirect'

Groovy is a very powerful and dynamic language, and there are several different ways to work with and manipulate objects of different types.

You also need to be aware that Transact Services implement static type checking, in order to improve error checking and security. This creates some restrictions in the coding styles you can use.

Typed Objects

The best way to represent objects is as classes. (Some will no doubt have alternate opinions or a different use case, and this is an arguable point.)

Some simple code looks like this:

```
class Customer {
    String first
    String last
}

Customer customer = new Customer()
customer.first = "Mary"
customer.last = "Smith"

assert "Mary" == customer.first
```

Using classes is a little more work, because you have to define the class. And you have to modify it if there are any changes to the data structure. (This is compared to collections, which are inherently dynamic.)

But the advantages are:

- A very simple syntax: "customer.first"
- If you are using an IDE (you should be) then the IDE will give you code completion when you are typing
- Safety - if you accidentally type: "customer.furst", you will get an error. In a dynamic Groovy collection, you have just created a new entry called "furst".

JSON

Groovy gives us a very easy way to convert our object to a string representation - JSON.

```
import groovy.json.JsonOutput

String str =  JsonOutput.toJson(customer)
println str
println JsonOutput.prettyPrint(str)
```

This will produce: {"first":"Mary", "last", "Smith"}

The pretty version will produce a more easily human readable form including line breaks and indentation.

Collections, and converting JSON

You can alternately use collections - maps and lists - as an alternative to statically defined classes. Let's try this by converting our JSON string into a collection.

```
import groovy.json.JsonSlurper

def slurper = new groovy.json.JsonSlurper()
def customerMap = slurper.parseText(str)
```

This will convert the JSON sting into a collection. In this case, it's just a simple map, but if the JSON was more complicated, the map could contain other maps and lists, instead of just strings.

In regular Groovy, we can refer to the elements within the collection like this:

```
println customerMap.first
```

However, because Transact implements static type checking, you can't use this syntax directly. Because Groovy doesn't know what type "customerMap" is, it assumes it's just a regular object, which doesn't have a "first" attribute.

To fix this, you can either using casting (which gets a bit tedious, especially as your object graph gets deeper), or use the following alternate syntax:

```
println customerMap["first"]
```

Creating an object based on a collection

Groovy provides a very useful default constructor for any object based on a collection. This means that if we choose to go with typed classes, we can easily convert from our collection to a typed object.

The code below shows how we can get from a JSON string to a collection, and from a collection to a typed object. You could of course do this in a single line of code, but we've expanded for clarity.

```
import groovy.json.JsonSlurper

def slurper = new groovy.json.JsonSlurper()
def customerMap = slurper.parseText(str)
Customer customer2 = new Customer (customerMap)
```

Conclusion

Groovy provides some very powerful yet simple techniques for handling typed objects, collections and JSON, and very simple ways of converting between them.

This article promotes the use of typed objects (classes) for a variety of good reasons.

Transact places some restrictions above "pure" Groovy, but some simple syntax changes still allow us to have easy-to-write-and-read code.

Determining the IP address that a form is requested from

Unknown macro: 'redirect'

It may be necessary in certain instances to determine the IP address of the user who has requested a form, for example, so that internal users can view a different version of a form than external users, or so that users in different countries or areas see different form content.

The form must be prepopulated with either the IP address of the requesting user, or an indicator of what type of user they are, which has been determined using the IP address in a prepop Groovy script.

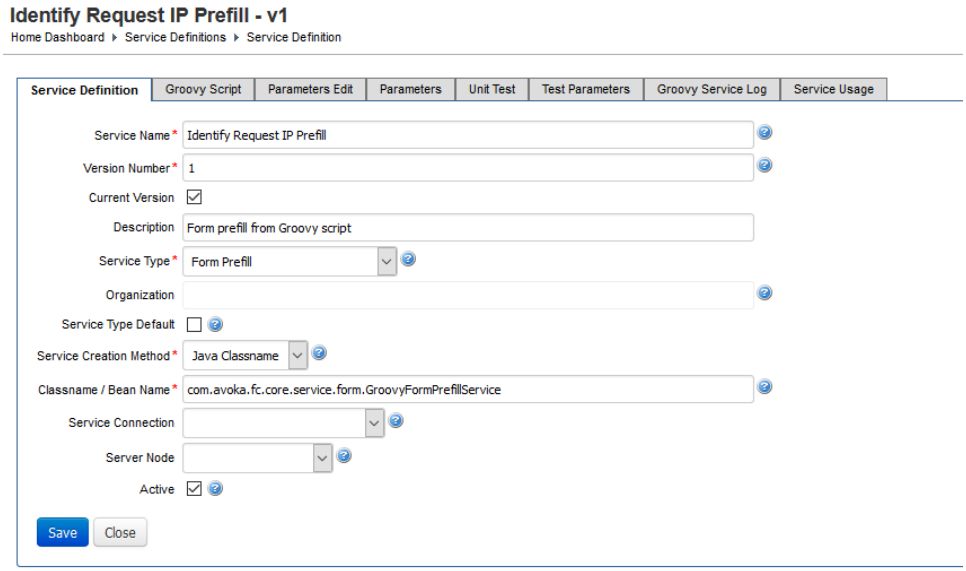
This can be achieved in Transaction Manager as follows:

1. Create the Groovy prefill service.

Groovy Code

```
// Script parametes include: form (Form), formRequestParams (Map<String, Object>), request
(HttpServletRequest), schemaSeed (Document), userAccountId (String), serviceDefinition
(ServiceDefinition), serviceParameters (Map<String, String>)
// Script return : Form XML data string value or return empty value and update the schemaSeed Document
parameter

def returnXml = '''
<prefillData>
    <value1>''' + request.getRemoteAddr() + '''</value1>
    <value2>''' + request.getHeader("X-Forwarded-For") + '''</value2>
</prefillData>
''';

return returnXml;
```

2. Create a Service Parameter containing the groovyScript.



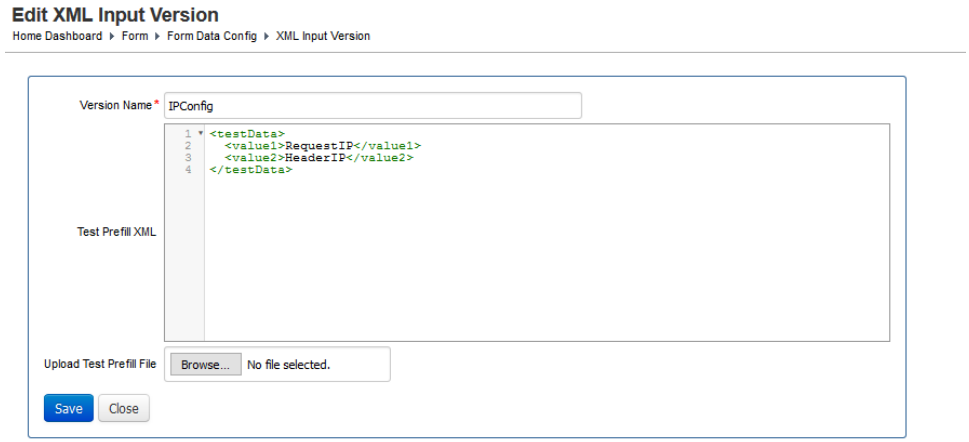
3. The groovyScript includes a request parameter. The request parameter is a HttpServletRequest Java object, which contains information about the request. The IP address that the request came from can be accessed using the following code:

> *request.getRemoteAddr()*

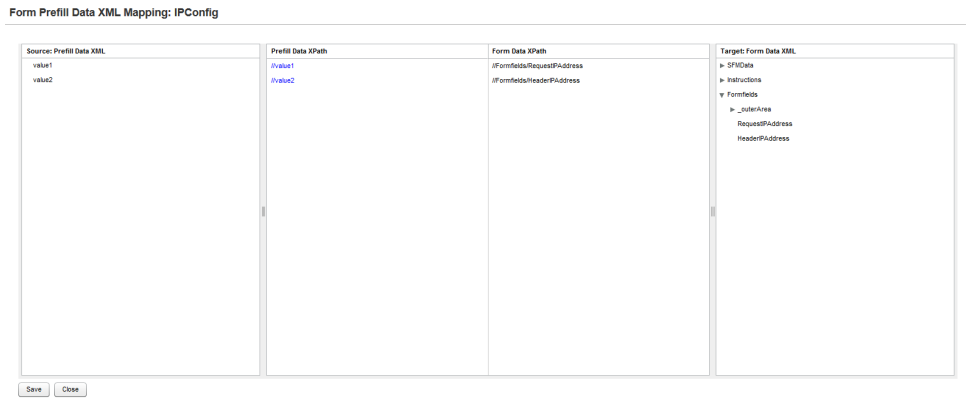
Depending how the TransactionManager server is set up, this may not always give you the user's address. For example, tm.demo.avoka.com sits behind a load balancer, which catches the user's HTTP request and uses it to create a new one. Using the above code will return the IP address of the load balancer, rather than the user. In this instance, the user's IP address must be accessed through the HttpServletRequest object's header, using the code:

> *request.getHeader("X-Forwarded-For")*

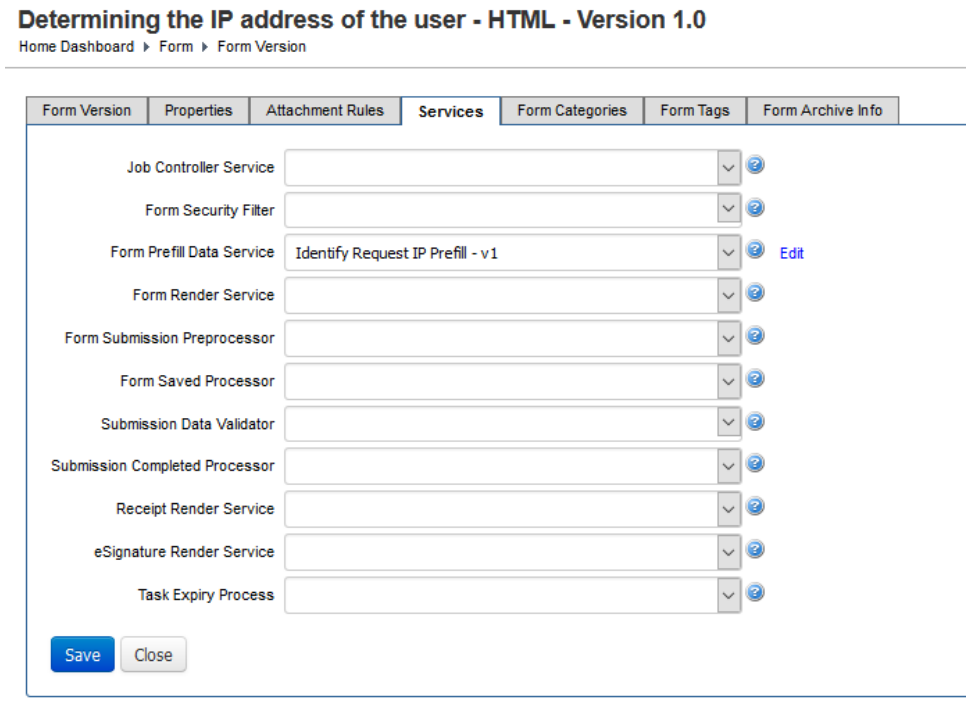
4. Create the sample input XML. In the form version data config, go to the input XML Prefill Mapping tab click **Edit Input XML**



5. Create the XML Mapping. On the same tab click the **Edit XML Mapping** button



6. Update the form version to include the Form Prefill Data Service.



Composer production --> Prepopulation > Cookbook > Determining the IP address of the user

Transaction Manager: https://tm.demo.avoka.com/govassist/servlet/SmartForm.html?formCode=determining-the-ip-a

Related articles

- How to pre-fill static map images into TransactField Tasks
- Determining the IP address that a form is requested from
- Repeating submission data extracts V4.1
- How to automatically extract the email address from a form and update the confirmation page
- How to define Form Submission Data extract fields in Composer V4

Getting to Know the UserBuilder API

Unknown macro: 'redirect'

The UserBuilder Fluent API is a great little api that can be used to create new TM users, update or change user properties such as their role or spaces assigned, and much more.

Let's take a look at an example of how to create a new user.

Create New User

A simple code example could look like this:

```
import com.avoka.tm.svc.*
import com.avoka.tm.vo.*

User thisUser = new UserBuilder(“Maguire”)
               .setEmail(“johndoe@avoka.com”)
               .setFirstName(“John”)
               .setLastName(“Doe”)
               .setRoleNames([“Developer”, "Maestro Developer"] as Set)
               .setOrgNames([“ACME”] as Set)
               .setSpaceNames([“maguire”] as Set)
               .setUserType(User.TYPE_LOCAL)
               .setLoginName(“alanreiley@gmail.com”)
               .setPassword(“testP@sswd12”)
               .create() // optionally use createOrUpdate() which will only update if the user already exists.
```

The above example will create a new user with a user name of johndoe@avoka.com, an email address of johndoe@avoka.com, has been assigned to the Maguire space, was granted Developer and Maestro Developer roles, Maguire space name, and the ACME organization. We had to add the **as Set** for the list types (which wasn't mentioned in the API documentation) so that Groovy knows to cast this as a Set. Otherwise, you will get an error. We also can choose between creating a Local account, LDAP-based, or an SSO-based users, as seen illustrated above with setUserType.

Updating an existing user is almost as simple, but comes with a caveat that you can easily overcome. The any property that is a Set, such as Roles, Spaces, and Organization, the update method **replaces** the existing properties assigned to that user. Let's take a look at updating an existing user's properties.

Updating an Existing User

Groovy gives us a very easy way to convert our object to a string representation - JSON.

```
import com.avoka.tm.svc.*
import com.avoka.tm.vo.*

User user = new UserBuilder("Web Plug-in")
  .setLoginName("johndoe@avoka.com")
  .setRoleNames(["Developer","Maestro Developer","Maestro SCM Design","Maestro SCM Org Library","Maestro SCM
Project"] as Set)
  .update() // Updates the user account specified by the loginName parameter
```

The above example will find the user: johndoe@avoka.com and updates their role assignment to become: Developer,Maestro Developer, Maestro SCM Design, Maestro SCM Org Library, Maestro SCM Project.

Using the Groovy Console

Remember to select "Commit DB Changes" if you're using this script in a Groovy Console or the DB write will be ignored and no user will be created.

Conclusion

The UserBuilder API is a very powerful tool for creating and modifying users and has become a very handy tool for user creation here within the Support Group. Some other helpful functions of the UserBuilder API are available to set the Account Status, remove an Org, and set the User Type to SSO.

Application Development Patterns

Unknown macro: 'redirect'

- Pattern for server-side persistence
- Pattern to Implement a Long Running Service
- Using the http session object

Pattern for server-side persistence

Unknown macro: 'redirect'

The main persistence model for Avoka Transact is the XML file. This is stored on the server, sent to the browser when a form is rendered, and sent back to the server when the form is submitted. It is also saved to the server in background save events.

However, there are cases when it may be useful to store information on the server which is not stored in the XML file. Some of these cases include:

- Storing results from a backend server call, such as an actual credit score– we want to record the actual score, but all we want to send to the form is a pass/fail. We don't want it to be in the XML because we don't want the end-customer to see their actual score, or the reasons why they failed.
- We want to temporarily store in-progress data that has been used in back-end service calls. This is to ensure that the end-user doesn't "game" the system, by changing the data in the form at different points in the journey.
- We want to store the results of a back-end service call for auditing or debugging purposes - but it's too much data to inject into the XML.
- There are probably many other cases.
- This data could be as simple as a string, or more complex like some JSON or XML.

There is a simple way to store and retrieve arbitrary string data against a Transaction using Fluent service calls.

Storing data

TxnUpdater has a method setProperty(String name, String value) which will allow you to create (or update) a property value.

Code will look something like this:

```
new TxnUpdater(txn)
    .setProperty("MyProp", "MyValue")
    .update()
```

Retrieving data

You can use TxnQuery to find the relevant Transaction, and then look at the Property Map within the Txn value object to retrieve the particular property.

Code will look something like:

```
Txn txn = new TxnQuery()
    .setSubmitKey(…) // or similar to find the correct transaction
        .withPropertyMap() // to retrieve the properties
        .firstValue() //to find the first matching transaction


// The txn contains an attribute propertyMap of type Map<String,String>
```

Some notes

- Data stored in the Transaction Properties are encrypted and subject to data retention policies, just like XML. (In recent versions of Transact Manager.)
- In some cases, it may be necessary to store data against a collaboration job rather than an individual transaction. Please refer to this article: 66880620
- The PropertyQuery service is used for locating configuration properties for Form Versions, Workspaces, etc. This is for static configuration properties, not for per-transaction or runtime properties.
- It is also possible to use the standard HTTP session to store temporary data. However, this data will not be persisted beyond an individual session. A transaction property is persisted in the database.

References

TxnUpdater

TxnQuery

Txn

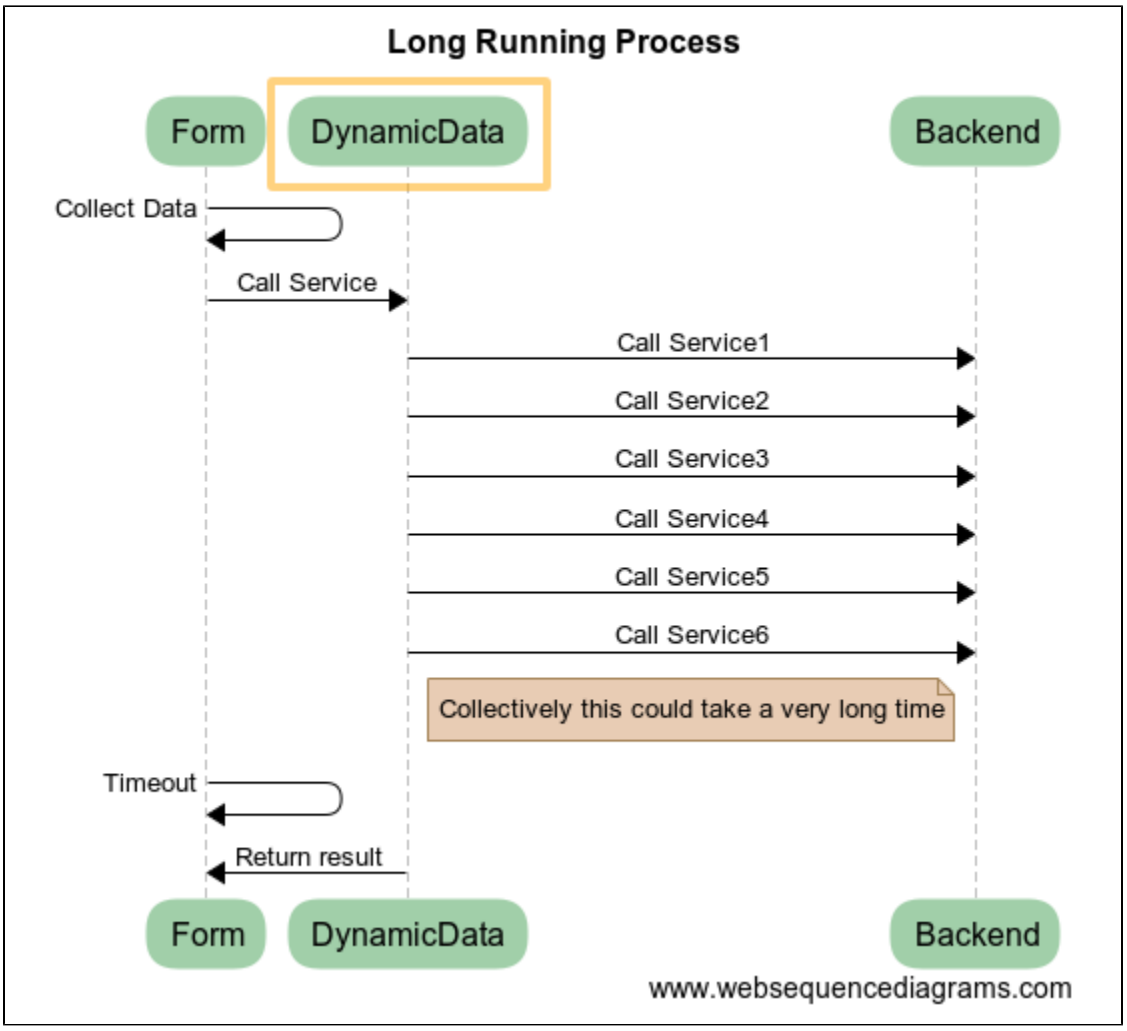
PropertyQuery

Pattern to Implement a Long Running Service

Unknown macro: 'redirect'

Sometimes you may need to invoke a Dynamic Data Service from a form, and you know that it may take some time. Perhaps there is a back-end service that is slow, or perhaps you are calling several different services in order to get your answer.

It might look something like this:



There is two problems with this:

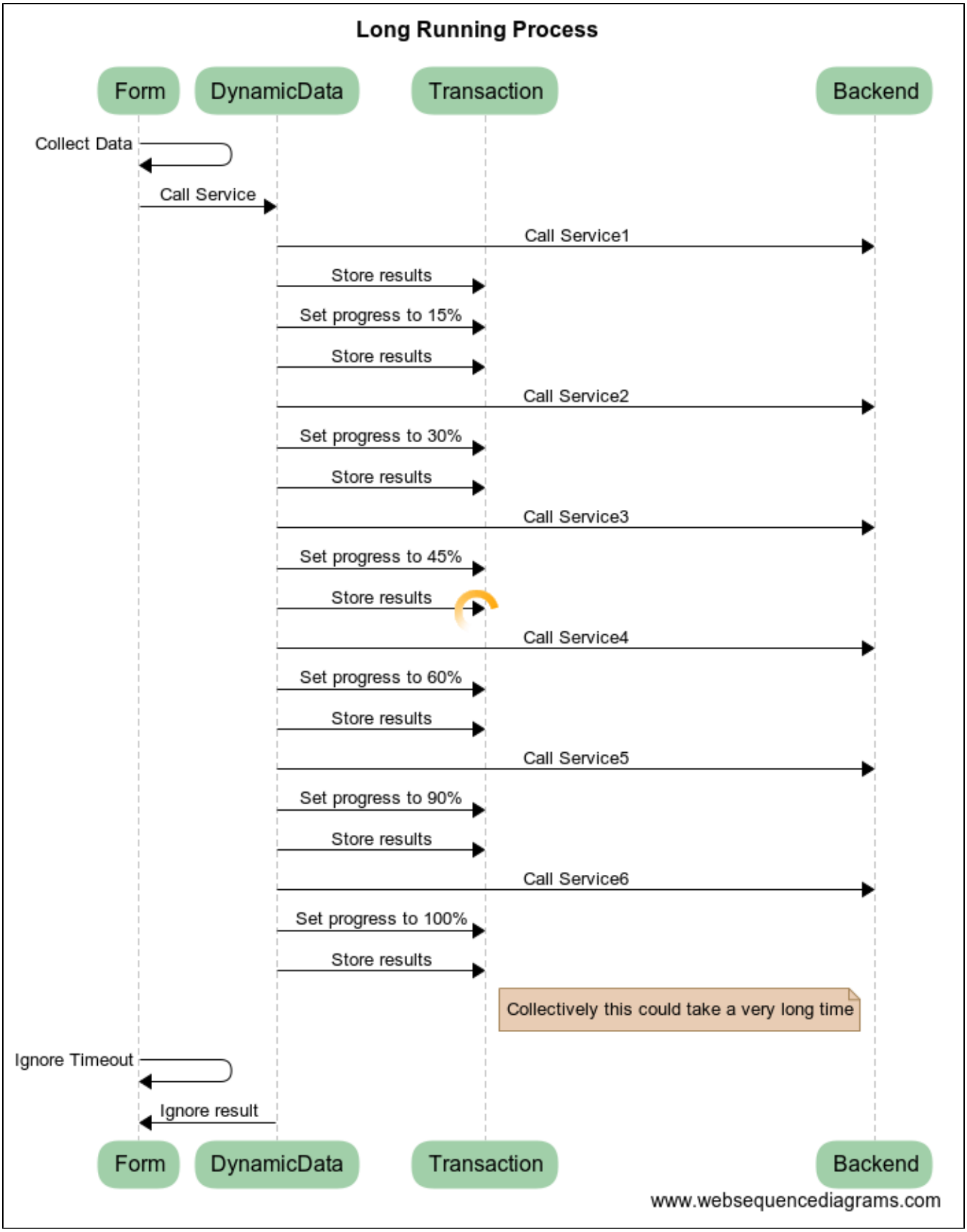
- Both Maestro and the Transact Manager web server (usually Apache) have a built-in service timeouts. If you exceed this timeout, you will receive an error in your application. This means that you will never actually receive a result.
- While the service is running, the user has no feedback on what is going on.

One solution would be to increase the timeouts. However, the timeouts are there for a reason - they are there to protect the application server from having a large number of long-running http connections open. Too many open connections, and the server will eventually run out of resources.

Here are two patterns that may be helpful to work around this problem.

Progress Polling

One of the useful things about the web (and Transact) is that services are all asynchronous. So we can modify the above sequence to look something like this:

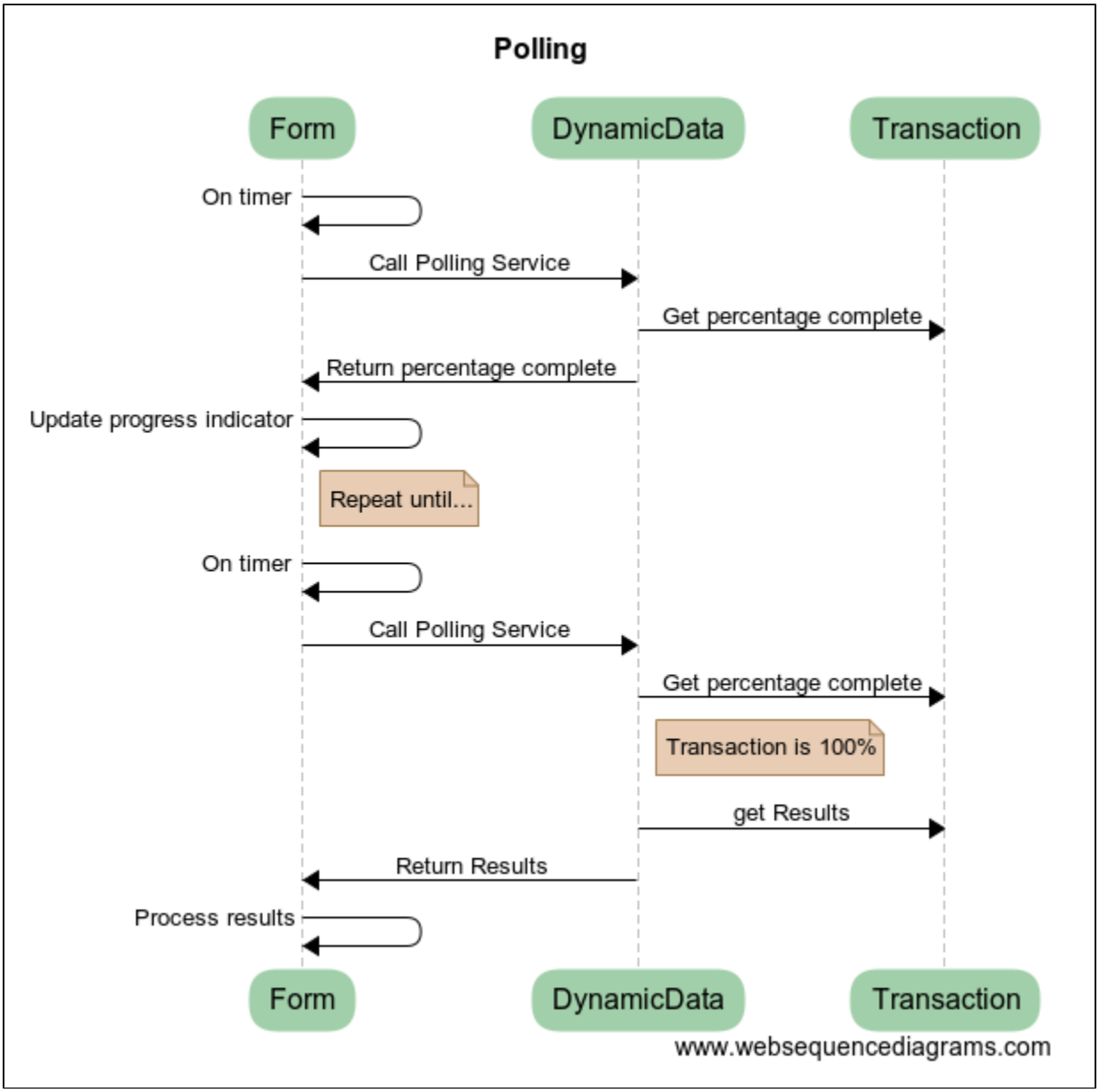


We've basically made three changes:

1. Update a transaction property to represent the percentage progress of the long running process.
2. Store the intermedate results of the service calls in a transaction property (or multiple properties).

3. Ignore both the timeout and the result of the initial Dynamic Data Service call.

Then, in parallel, we will implement another Dynamic Data Service call. This will be on a timer, and will call the TM server periodically.



The form will call the Polling Service periodically.

The polling service will check the percentage complete, and either;

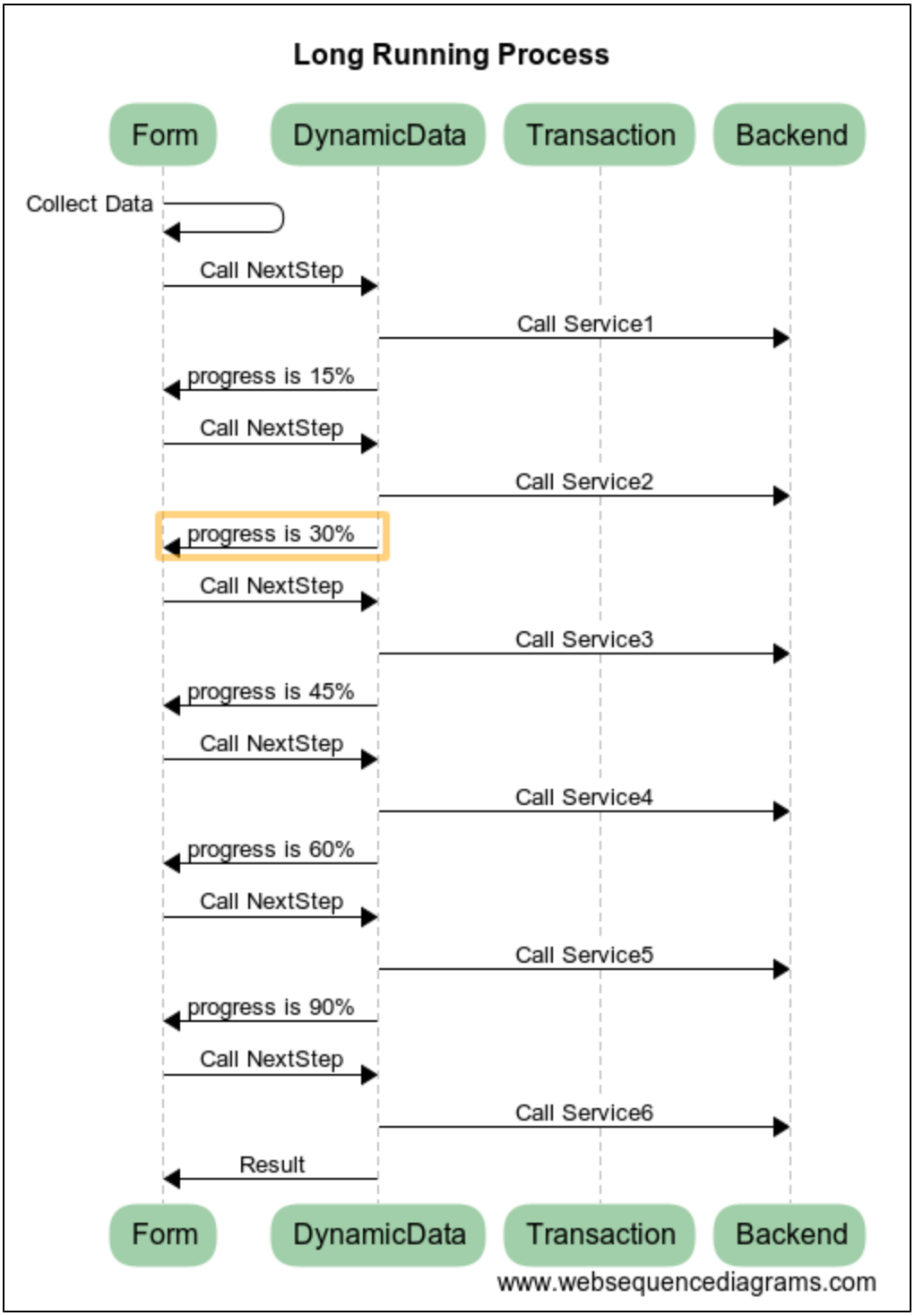
- Return the percentage complete, so that the form can update the progress meter (or similar) (This is shown in the top half of the diagram above.)
- If the process is complete, return the results, so the form can use them. (This is shown in the bottom half of the diagram above.)

This solves both problems.

Blind state machine

Another way to solve this is to use a blind state machine.

This breaks up the sequence of execution, but does so in a way that the Form is unaware of the details or logic that is occurring on the server. This helps to make sure that the form cannot be compromised by a malicious user.



In this pattern:

- We break up the calls to ensure that none of them exceeds the timeout value.
- In the same way as the above solution, we are giving feedback to the form that allows it to update a progress meter.
- We are careful not to expose the internals of the state machine, or how its logic works. All we are doing is calling "NextStep" until we get back a result rather than a progress indicator.

Enterprise Service Bus

A third approach is to use an Enterprise Service Bus or similar architecture. In this case, we would use message queues or similar patterns to "kick off" a long running process. We would then using the polling pattern to discover when the process is complete.

Conclusion

Both the patterns described should be fairly easy to implement, and solve the required problem. The progress polling pattern is arguably superior, because it completely hides the server implementation from the Form.

Using the http session object

Unknown macro: 'redirect'

You can store and retrieve data against the http session object in the usual way. (Not going to what the session object is, or explain how to do that in this article.)

However:

- The session object is (by definition) temporary/volatile, and may "go away" under several circumstances, such as save-and-resume.
- Data stored in the session object cannot be easily viewed in Transact Manager, and therefore makes debugging difficult.
- Don't mess with the session object itself, because it is used internally by Transact Manager. For example, the following code will cause all sorts of problems:

```
request.getSession(false)?.invalidate() // DON'T do this!!!!
```

As a general recommendation, you should strongly consider storing and retrieving data using the Transact Manager built-in services.

More information is here: Pattern for server-side persistence

TransactField Mobile App

Unknown macro: 'redirect'

- How to pre-fill static map images into TransactField Tasks
- How to rescue attachments from your iOS device
- Interacting with TransactField using Custom URLs
- How to disable the standard Submit/Attachment button in TransactField

How to pre-fill static map images into TransactField Tasks

Unknown macro: 'redirect'

TransactField has a feature that allows the user to view a map displaying the locations of tasks allocated to them. However this feature is reliant on the user having internet connectivity. For cases where the user is operating without connectivity, but they still want a map displaying the location of their task, a static map image needs to be prefilled into the task.

This can be achieved by prefilling the task XML with a base 64 string representation of the static map image, and then using Javascript to load the string into an image's **scr** attribute when the task is rendered.

Generating a static map in base 64

The following code can be used in the groovy script that generates the task to request a static map image from google, encode it in base 64, and prefill it into the task XML:

```
def lat = '-9.7515335'
def long = '143.4052768'
def url = "https://maps.googleapis.com/maps/api/staticmap?center="+lat+","+long+
"&zoom=16&size=600x400&maptype=satellite&markers=color:red%7Clabel:J%7C"+lat+","+long
def rawImg = (url).toURL().getBytes() //get image bytes
def img = "" + rawImg.encodeBase64() //encode in base 64
xmlPrefill.RecordTrapObservatio.TrapDetails.JobMap.mapSource[0].setValue(img) //store string in task XML
(assumes  xmlPrefill object already exists)
//include prefilled xml in task parameters
def param = new SubmissionTaskService.FormTaskParam()
param.formDataXml = XmlUtil.serialize(xmlPrefill)
```

Loading the pre-filled string into an image

The following code can be used within the form to load the base 64 string into an image's **src** attribute:

```
document.getElementsByClassName("jobMap")[0].children[0].setAttribute("src", {mapSource});
```

In this case I have used an **Image [Resource]** widget in the form, and applied a custom html class of 'jobMap' to the widget.

{mapSource} is the reference to the prefilled base 64 string.

Note: it is also possible to generate a road map image rather than a satellite photo by entering different parameters into the **googleapis** url.

Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Interacting with TransactField using Custom URLs
- How to pre-fill static map images into TransactField Tasks
- TransactField Mobile App
- Determining the IP address that a form is requested from

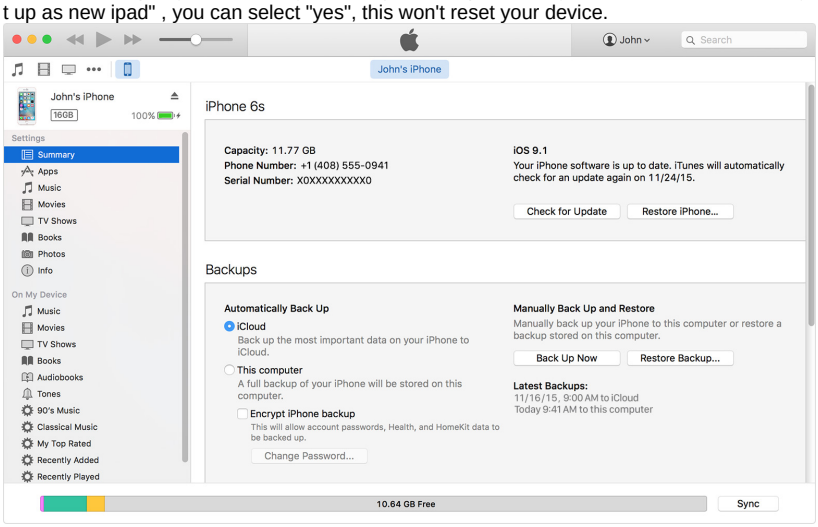
How to rescue attachments from your iOS device

Unknown macro: 'redirect'

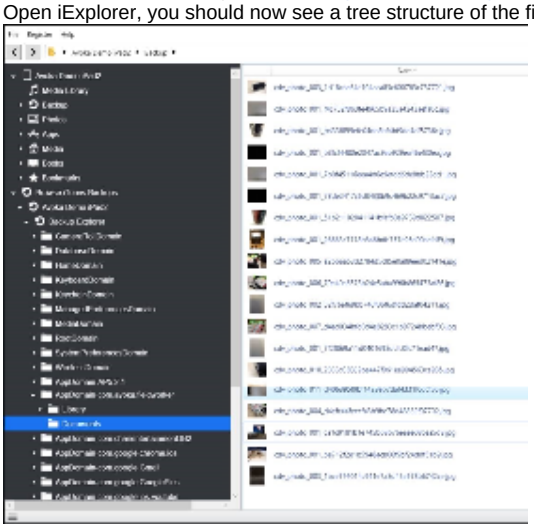
Every so often the unforeseen happens and you find yourself with an iOS device (typically an iPad) that is unable to synchronise successfully with Transaction Manager. TField would most likely already have syncronised your submissions but your photos are still on the device pending upload. The following steps will allow you to recover the attachments from a device backup using a 3rd party tool called iExplorer (https://www.macroplant.com/iexplorer/).

Step-by-step guide

1. Connect your device to your laptop/desktop computer and open iTunes
2. Create a backup of your device by selecting "Back Up Now". If you have never connected your device to iTunes you will be asked if you like to "Set up as new ipad" , you can select "yes", this won't reset your device.



3. Download and install iExplorer from https://www.macroplant.com/iexplorer
4. Open iExplorer, you should now see a tree structure of the files and backups of your device



5. You should be able to find your backup under "Browse iTunes Backups", device name, "Backup Explorer".

6. Within your backup you will find "AppDomain.com.avoka.fieldworker", "Documents". This is the folder that contains all your attachments.
7. Right click on the "Documents" folder and select "Export to Folder", select the destination location and click OK.
8. Your attachments are now available on your PC

Related articles

- How to rescue attachments from your iOS device
- Customising Attachment Field Widget

Interacting with TransactField using Custom URLs

Unknown macro: 'redirect'

On mobile devices it is possible to interact with the TransactField app(TField) using custom urls. This functionality can be used to open TField from browsers or emails and direct users to specific forms or task.

Custom Url

The following URL will open the TField app on a mobile device.

> *fieldworker://services.avoka.com*

Inorder to achieve further functionality from custom url you must pass additional parameters to TField. The following parameters are supported by TField:

Parameter values passed with custom urls must be url encoded. There are free tools online for encoding/decoding urls, for example. http://www.freeformatter.com/url-encoder.html

Server

Server provides the server context path to TField. Passing this means that a users doesn't need to worry about manually putting this in the App.

> *Example: fieldworker://services.avoka.com?server=https%3A%2F%2Ftm.test1.avoka.com%2Ffield-worker/*

User

Using the User parameter you can prefill the login name for the application.

> *Example: fieldworker://services.avoka.com?user=loginName*

The Server and User parameters are most useful for helping to setup new user or devices. For Example

> *fieldworker://services.avoka.com?server=https%3A%2F%2Ftm.test1.avoka.com%2Ffield-worker/&user=loginName*

Caution should be taken using the Server parameter as this will trigger an app purge. User will then need to resync with the server and any unsynced data stored on the device will be lost.

Formcode

The form code for a particular form can be passed in as a parameter. Users can be sent directly to a particular form. Note, the user will need to have configured the server and login in with valid credentials to open the form.

> *Example: fieldworker://services.avoka.com?formcode=showcaseform*

Taskkey

Users can also be directed to a particular task using a task key. This key can be found in Transact Manager within the assigned task.

> *Example: fieldworker://services.avoka.com?taskkey=5968c55dd4a894e942a6dce26881badd*

Prefill

The Prefill parameter can be used prefill data into a form. To prefill a field you need its XML xpath, using this the parameter is formatted as follows Prefill=XPATH=VALUE. Any number of prefill values can be passed with the custom url, delimited by a semicolon (Prefill=XPATH1=VALUE1; XPATH2=VALUE2).

> *Example: fieldworker://services.avoka.com?formcode=showcaseform&prefill=%2F%2FGettingStarted%2FAboutYou%2Fname=FIRSTNAME%3B%2F%2FGettingStarted%2FAboutYou%2Femail=em%40il.com*

Custom Url Example

For a task assigned from a collaration Job, an email is generated with the following link:

> *fieldworker://services.avoka.com?user=userName&taskkey=5968c55dd4a894e942a6dce26881badd*

This url will set the user for Tfield and direct the user to a specific task(once they have logged in).

How to disable the standard Submit/Attachment button in TransactField

Unknown macro: 'redirect'

Compatibility

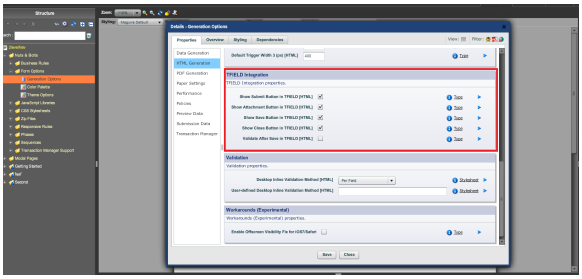
Since	4.0
Deprecated	

To disable the standard Submit and/or the Attachment button, go to "Nuts & Bolts > Form Options > Generation Options > Properties > HTML Generation > TFIELD".

Why would you use these options:

If a form has no mandatory fields, it would be useful to place the Submit button at the end of the form to prevent the form filler submitting the form too early. In this case, you would disable the mobile app submit button by unchecking the option, and place a submit button on the form.

If you had a form with no attachments, it is useful to remove the attachment button from the application to minimize confusion. Additionally, if you have elected to use the camera or gallery widget button in the form, and prefer your users to use the in form options, you may wish to uncheck this option.



Related articles

- How to disable the standard Submit/Attachment button in TransactField
- Interacting with TransactField using Custom URLs
- How to pre-fill static map images into TransactField Tasks
- TransactField Mobile App
- Accessing Form properties in Groovy Services

Transact SDK

Fixing SSL Certificate chain issues

Unknown macro: 'redirect'

When attempting to use the SDK to deploy a form to a Transact server (using the app-deploy Ant task), you may encounter the following error:

```
javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: PKIX path building failed: sun.
security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested
target
```

This error occurs when the server has a SSL certificate where the certificate chain cannot be resolved, and therefore cannot be trusted. This should only occur on non-production systems, if it occurs on a production system, it indicates an error that you should fixed.

Fixing this error is relatively straight forward, although a little tedious.

Download all the certificates for the Transact Manager server

Log in to the TM Administration UI in your browser, and then follow the following article to download all the certificates in the certificate chain:

http://docs.bvstools.com/home/ssl-documentation/exporting-certificate-authorities-cas-from-a-website

Note that you do need all the certificates in the chain - although some of them may already be installed in your Java default certificate store.

This online checker may also help to understand your site's certificates: https://www.sslchecker.com/sslchecker

Install the certificates into the Java SDK

First identify the correct JDK installation. This should be the one that is configured in your IDE to be used when running the Ant tasks. (Refer to IDE tool to determine this.)

Then open a command prompt, and navigate into the **java/jdk-xxx/bin** directory, then execute the following commands:

Command	Description
keytool -cacerts -list	List. This dumps a list of the existing certificates in the store. If this runs successfully, then you are ready to run actual commands.
keytool -cacerts -import -trustcacerts -storepass changeit -alias <somealias>-file "C:\temp\somecert.cer"	Import. This imports one of the downloaded certificates into the certificate store. You should start with the top level certificate, and work your way down. If you get a message saying the certificate already exists, don't import it.
keytool -delete -cacerts -storepass changeit -alias <somealias>	Delete. If you accidentally do the wrong thing, you can remove a certificate using this command.

Notes:

- More recent versions of **keytool** use the **-cacerts** command to identify the default cacerts store. If your version of Java does not support this option, replace with **-keystore ../lib/security/cacerts**
- **-storepass <password>** specifies the password is the cacerts file. The default password is "changeit".
- **-alias <somealias>** - you can use any alias you want, but it should match the certificate name.
- The commands are for a Windows installation - other platforms may vary.