temenos

Exchange

JOURNEY MANAGER

VERSION 22.10

TOC

Journey Exchange Overview **8**

Contacting the Exchange Team **10**

Exchange Package Architecture **11**

Exchange Development Schedule **12**

Exchange Partnerships **13**

Exchange Developer Resources **14**

Access to Exchange Packages **15**

Access to Exchange packages for Journey Manager 15

Access to Exchange packages via ZIP for Journey Manager 16

Exchange Permissions **18**

Exchange Package Installation **20**

Installation via Journey Exchange 20

Video 22

Installation via a ZIP archive file (TPac) 22

What is a Transact Distribution Package (TPac) 23

Exchange Package Configuration **25**

Exchange package architecture 25

Package Configuration 25

Typical configuration tasks required in Journey Manager 26

Typical configuration tasks required in Maestro 26

Exchange Development Schedule **28**

Development Schedule 28

Unscheduled 28

Contact the Exchange Team **32**

What we require 32
How to log a support request **33**
Log a support request 33
Exchange Partner Program **35**
Who can apply? 35
What is involved? 35
The Exchange Publishing Process **36**
Step 1 - Apply 36
Apply for your Exchange Partner Account 36
Step 2 - Build 36
Build your Extension Package on our platform 36
Step 3 - Certify 37
Submit your Package for Exchange Certification 37
Step 4 - Publish 38
Release your Package to the Avoka Exchange 38
Exchange Certification **39**
Certification Checklist 39
Certification Report 40
Adequate Package Documentation **42**
Fluent SDK Document Generation 42
Documenting Groovy Services 43
Example service parameter definition from a service-def.json file 44
Documenting Maestro Assets 44
Package Overview Documentation 46
Example Custom Section 47

Example Custom Section 47
Example Release Notes 48
Example Release Notes 48
Tip 49
Language Translation Support 50
Maestro Standard Components 50
Maestro Custom Components Properties 50
Maestro JavaScript String Literals 51
Switching language at runtime 52
Example 53
Maestro Item Property References 53
Example 53
Package Compatibility Requirements 55
Groovy Service Compatibility 55
Example service-def.json 56
Maestro Library Compatibility 56
Example .library-config.xml 57
Package documentation 57
Example package.html 58
Security of Customer Data 59
Unacceptable practices 59
Common use cases 59
Acceptable practices 60
Unique Asset Naming Strategy 62
Groovy Service Naming 62

Service Connection Naming 63
Maestro Component Naming 63
Exchange Developer Resources 65
Exchange package architecture 65
Developer Resources 65
Exchange Framework 65
Certification 66
Legacy 66
Exchange Framework 67
Exchange Framework (TIF) v1.5.0 release 67
A standard fluent function base class 67
A powerful standard response processor out of the box 68
Exchange Framework (TIF) v1.6.1 release 68
References 69
The Exchange Domain Shared Code Library (for Fluent SDK 5.x) 70
Shared Code Library 70
Accessing the Exchange Git Repository 71
The Exchange Sandbox Environment 72
Using GitFlow in Exchange projects 73
Installing GitFlow 75
75
Migrating existing projects 75
When 75
How 75
GitFlow command 75

Git command 76
IntelliJ GitFlow plugin 76
77
Make changes to the project (Enhancement or bug fix) 77
77
Step 1 Create a feature branch 77
GitFlow command 78
Git command 78
IntelliJ GitFlow plugin 78
Step 2 Make changes 78
78
Step 3 Finalize the feature branch 78
GitFlow command 79
Git command 79
IntelliJ GitFlow plugin 79
Releasing a version 79
Step 1 Create a release branch 79
GitFlow command 80
Git command 80
IntelliJ GitFlow plugin 80
Step 2 Create distribute pack 80
80
Step 3 Finalize the release branch 80
GitFlow command 81
Git command 81

IntelliJ GitFlow plugin 81
Logging third party service calls **83**
Background 83
Logging convention 83
Migration Policy 84
More on logging third party service calls 84
Usage 84
Sample code 85
Inputs 85
Managing multiple service endpoints and credentials for external service calls **87**
Creating Service Connections 87
Using Service Connections in Groovy Services 88
Deployment Practices 90
Calling Multiple Endpoints in a Single Service 91

Journey Exchange Overview

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

The Journey Exchange is a catalog of pre-integrated FinTech[1] applications written for Maestro, such as Mitek, FIS, Mastersoft, and LexisNexis, to extend the power of the Temenos Journey Manager (TJM) platform.

These third party integration packages are made available to our Journey Manager customers for use in their online customer applications. A list of published and available packages per region is maintained at: Exchange Packages

The Journey Exchange catalog has been integrated into Journey Manager from 18.05 to allow you to easily download Exchange packages directly into your Manager environment.

You will need Exchange permissions to be able to do this. However, for all earlier versions of Manager, a zip file gives you access to the Exchange package services and Maestro assets. To find out more, refer to:

- Package Installation
- Exchange Permissions
- Package Configuration

Watch this video for an introduction to the Exchange.

[1]Financial Technology, also known as FinTech, is a new industry that uses new technology and innovation to improve activities in finance. It aims to compete with traditional financial methods in the delivery of financial services to customers by improving customer experience, reducing time-to-market, decreasing cost to mention a few.

Contacting the Exchange Team

You can contact the Exchange Team to suggest an idea or raise a support request through the Temenos Support portal:

Need to log an idea or issue?

The Support Handbook shows you how to log them in the Support Portal.

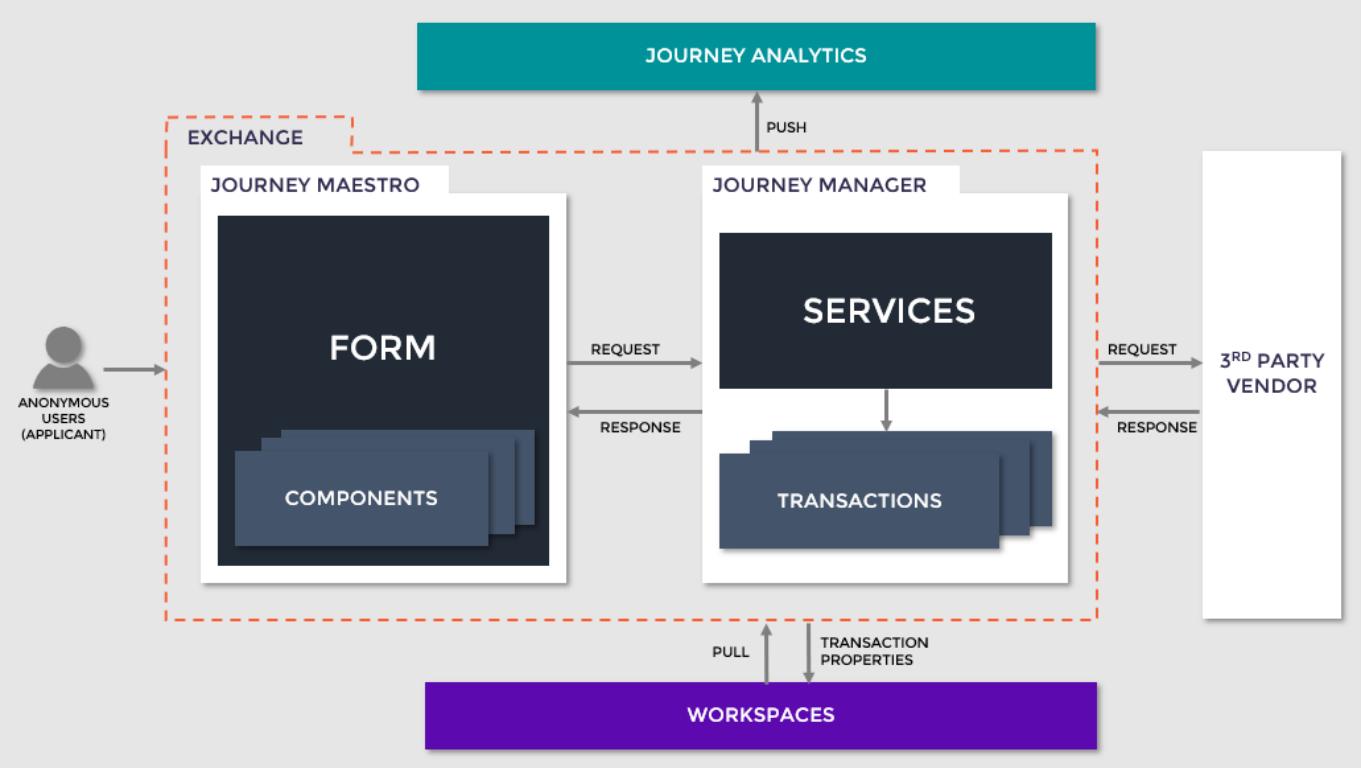
1. Ensure you're using the original Exchange package (not modified services or Maestro assets). *Support is only provided if the Manager services or Maestro assets have not been changed.
2. Include your Manager, Maestro and Exchange package versions.
3. Capture / reproduce the steps taken using a Maestro template (blank, Maguire, Blue Steel or Avalon). This is essential so that we can replicate the issue in our own environment.

Exchange Package Architecture

All Exchange packages are based on the same architecture and interact with Manager and Maestro in the same manner.

As the user interacts with an Exchange package implemented in their application (form), the third party services are called via Manager, and these events are recorded in Journey Analytics, the platform's analytics tool, to be further analyzed in the future.

The transaction properties of the user's application are also sent to the Workspaces portal, so the application life cycle can be reviewed by the Workspaces users.



Exchange Development Schedule

More Exchange packages are always being developed.

Packages being considered for development are here: Development Schedule

Exchange Partnerships

Interested in becoming an Exchange Partner and publishing your own solutions?

The Exchange Partner Program is designed to allow your team to participate in the Avoka Exchange service by developing, certifying and publishing your own solutions. Find out what is involved:

- The Exchange Publishing Process
- Exchange Certification
- Developer Resources

Exchange Developer Resources

Looking for answers as you develop? Let us assist you to get your Exchange package built, certified and published.

We've got some important tips and guidelines that are a great place to start: Developer Resources

Access to Exchange Packages

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

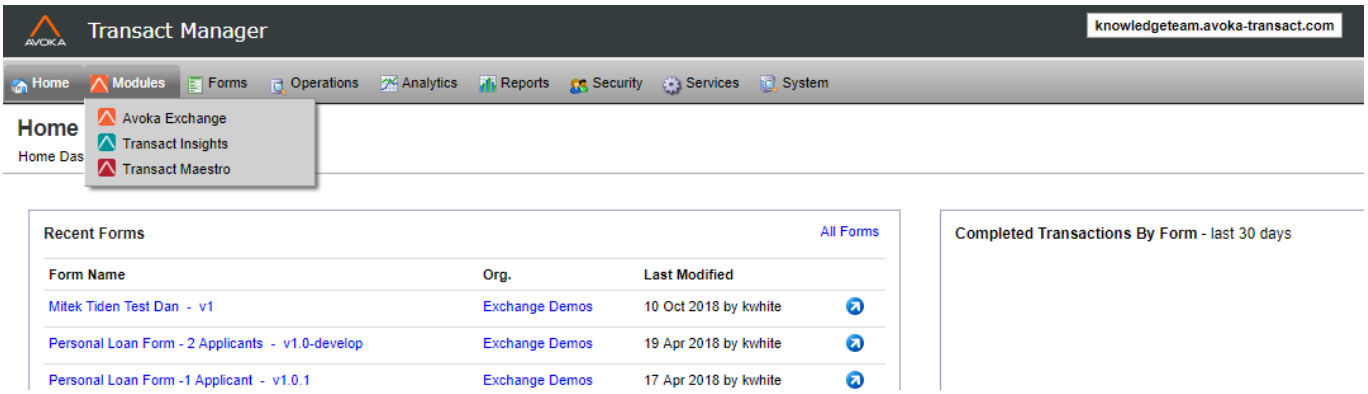
There are two ways to access Exchange packages:

- You can easily download the packages directly from the Journey Exchange catalog into your Journey Manager environment.
- Or via an archive ZIP file for all earlier versions of .

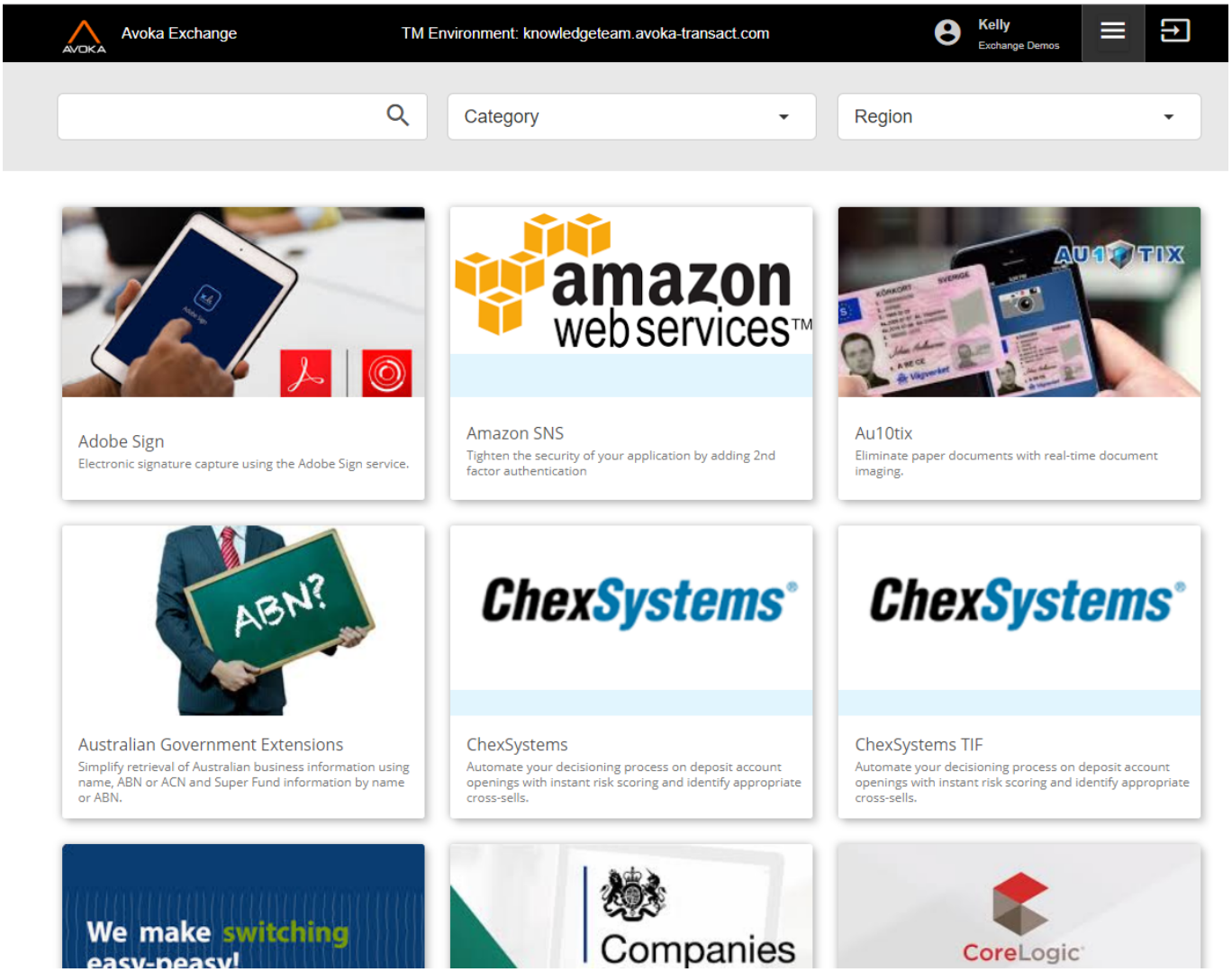
Access to Exchange packages for Journey Manager

| 18.05 This feature was introduced in 18.05.

Use the Journey Exchange to download and install the requireJourney Managerd package directly into your Journey Manager environment. You will need permissions to access the Journey Exchange.



We suggest filtering by region to confirm the package is available in your region, before installing.



For more information on installing Exchange Packages using the Avoka Exchange see Package Installation.

Access to Exchange packages via ZIP for Journey Manager

| v5.0 - v17.10 This feature is related to v5.0 to v17.10.

An archive ZIP file gives you access to the Exchange package for all earlier versions of Journey Manager (v5.0 - v17.10). If you are working on a project on Journey Manager that requires access to one or more Exchange packages, contact the Exchange team with the following details:

- Customer Name - Identify the customer who will be using the package.
- Manager Version - Note the Manager version number of the target environment.
- Intended Usage - Describe the use case(s) that the package will be applied to.

The Exchange team will review your request to ensure the package is compatible with your target environment and intended use case(s) before making the requested package available, in the form of an archive file in the ZIP format, called a Transact Distribution Package or TPac.

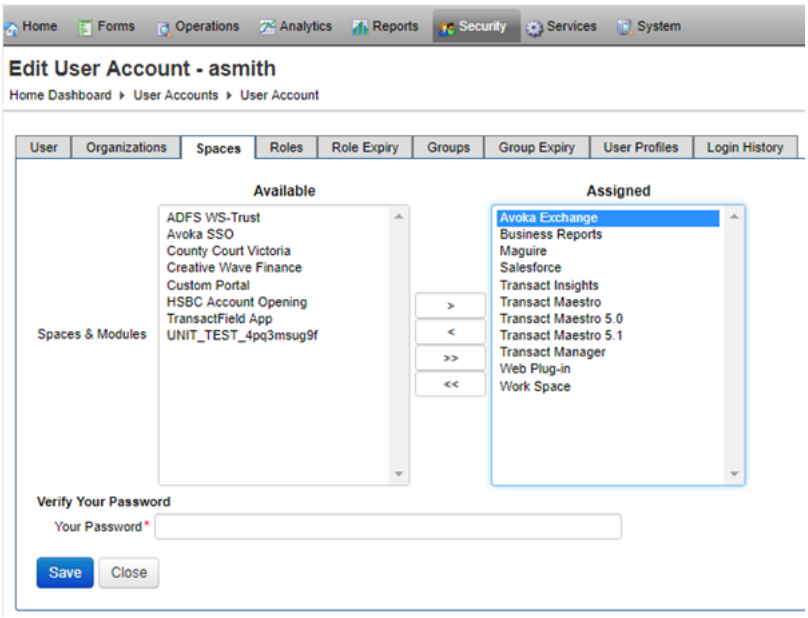
For more information on installing Exchange Packages using a ZIP file see Package Installation.

Exchange Permissions

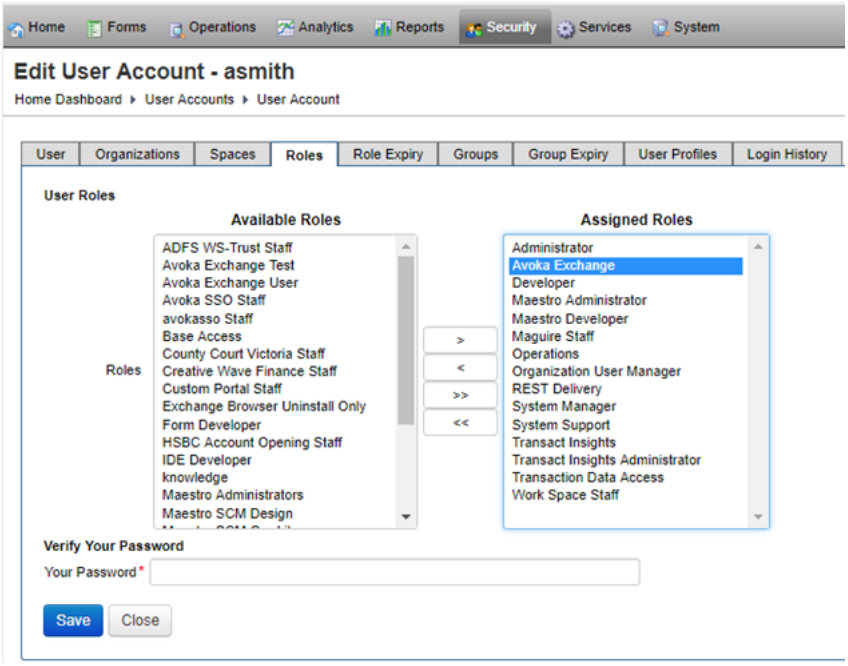
Exchange Pre-configured Maestro services. | Platform Developer | 18.11 This feature was introduced in 18.11.

The following permissions are required to access the Exchange in your Journey Manager environment to install Exchange packages to use in your Maestro applications.

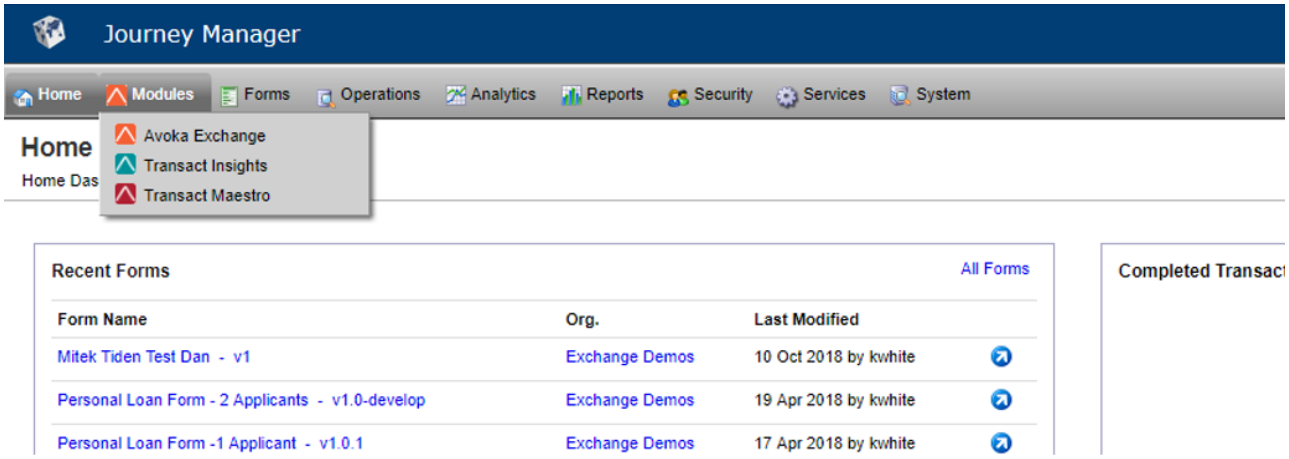
1. Login to Journey Manager.
2. Update your user's Journey Manager user account.
3. Click on the Spaces tab and add Avoka Exchange to the user's Assigned Spaces list.



4. This allows user access to view the Avoka Exchange.
5. Click on the Roles tab and add Avoka Exchange to the user's Assigned Roles list.



6. This allows users to install, uninstall and upgrade Exchange packages in the Exchange.
7. Once these permissions are granted, users can now log on to the Exchange with their updated Manager credentials, and select Journey Exchange from the Journey Manager Modules menu.



Exchange Package Installation

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

There are two ways to install Exchange packages into your Journey Manager environment to use in your Maestro applications. The installation method to use depends on your Journey Manager version. They are:

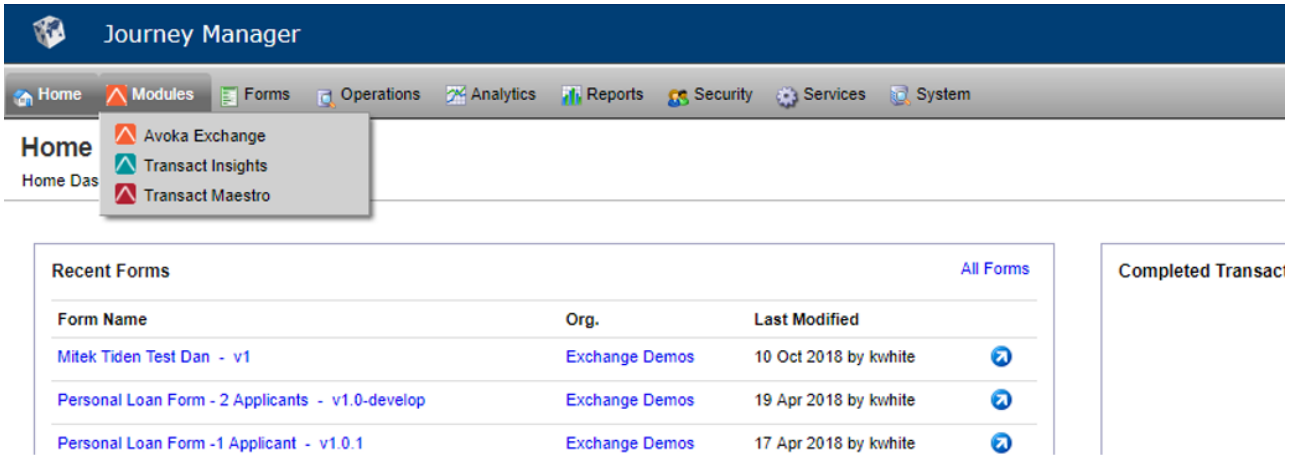
- Via the Journey Exchange
- Via a ZIP archive file

Installation via Journey Exchange

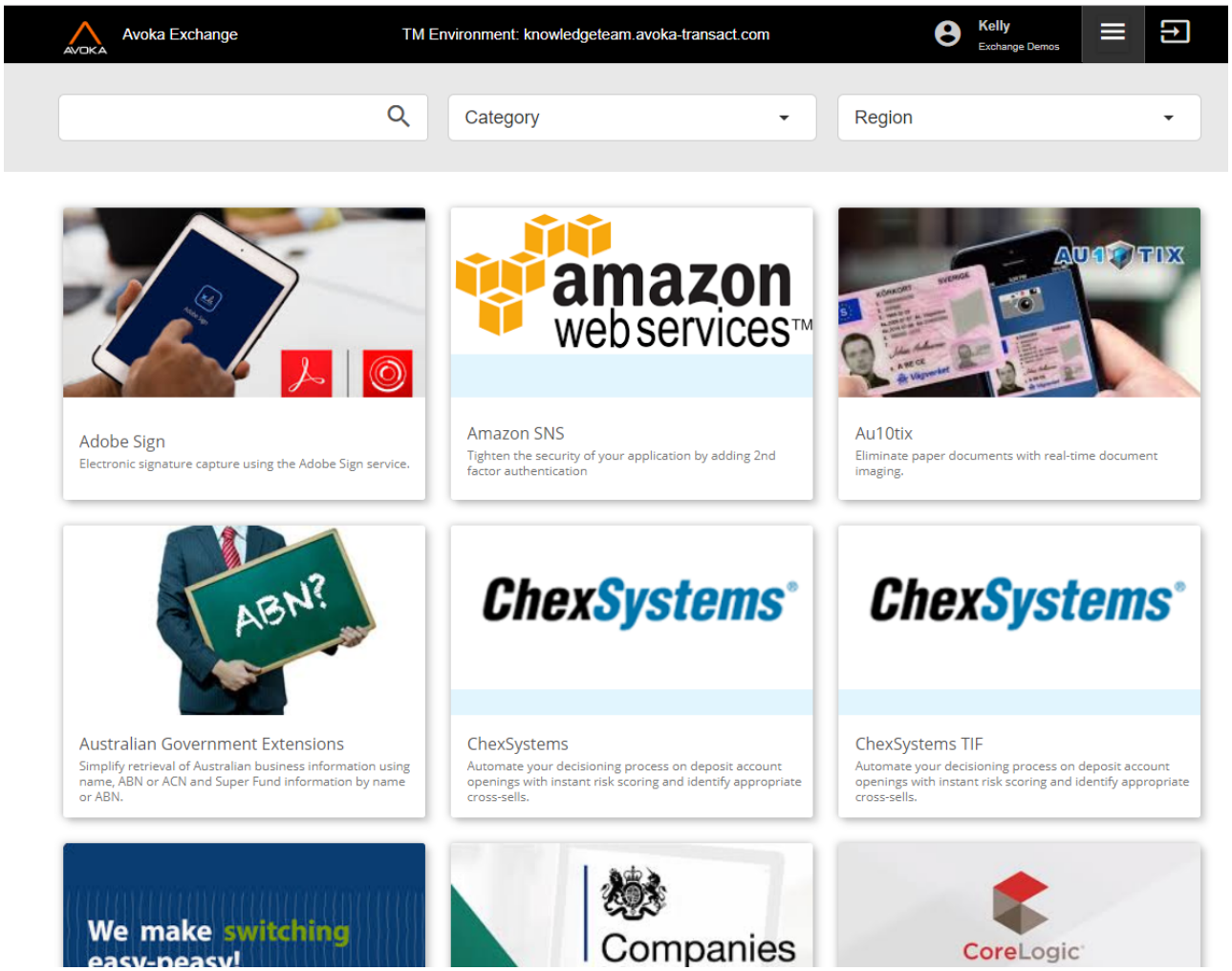
| 18.05 This feature was introduced in 18.05.

The Journey Exchange catalog has been integrated into Journey Manager from 18.05 to allow you to easily download Exchange packages directly into your Journey Manager environment. You will need Exchange permissions to be able to do this.

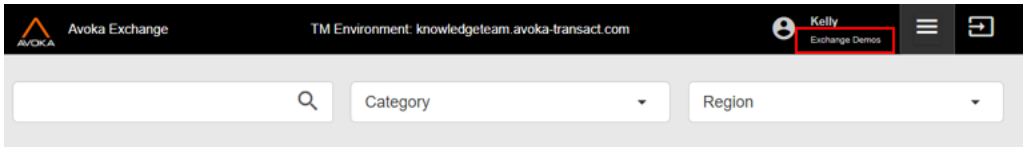
1. Select Avoka Exchange/Journey Exchange from the Journey Manager Modules menu.



2. Or login to the Avoka Exchange/Journey Exchange from your Journey Manager environment : https://<journeymanager.environment.com>/exchange/secure/web/exchange-browser



3. Locate the required Exchange package. Filter by Category and Region to confirm the package is supported in your region.
4. Make sure you have selected the correct organization. The package will be installed into the Organization you selected and will only be available for use in forms within that Organization.



5. Click INSTALL (Version 1.0)
6. The Exchange package will be downloaded and installed into your Journey Manager environment.

7. Then click on the Download Maestro library link to add the Maestro component of the package into the Maestro Palette.
8. The Exchange package services and Maestro assets are installed and ready for use.
9. Click on the View documentation link to understand what configuration is required to use the package.

Video

Watch the video for an introduction to the Avoka Exchange/Journey Exchange.

Installation via a ZIP archive file (TPac)

| All versions This feature is related to all versions.

To install an Exchange package on any version of Journey Manager follow the steps below:

1. Unzip the package to a directory on your computer.
2. Import each zip archive found in the Services folder to Journey Manager under the Services >> All Services menu item.

3. Review the Help Doc tab for each of the imported services and make any required adjustments to service parameters.
4. In the Service Definition tab of each service, check the Organization name to make sure it reflects the correct organization.
5. Configure any required Journey Manager Service Connections. Make sure they point to the package name and add your package credentials also. These requirements can be found in the Help Doc for each of the imported services.

   NOTE

   Service Connections can be configured in Journey Manager under the Services >> Service Connections menu item.

6. Test all package services are actively running in Journey Manager by clicking on the green arrows and make sure they display a green success message.
7. Import each archive found in the Libraries folder to Maestro.

   NOTE

   Importing these into the Organization level libraries folder is recommended as this will make the components available to all projects, however they can be imported at the Project level if required.

8. Review the package readme file to understand what configuration is required to use the package.

What is a Transact Distribution Package (TPac)

Transact Distribution Packages (TPacs) are the currency of the Avoka Exchange/Journey Exchange and contain all assets required to install and configure these extension modules. Typically these assets include:

- Package Documentation: HTML based read-me documentation detailing installation and usage instructions, compatibility notes, licensing requirements and detailed documentation for each asset in the package.
- Fluent Groovy Services: One or more Groovy services built using the Transact Fluent SDK to provide the API based integration to 3rd parties and supporting functions.

- Maestro Library: A library of design assets to support drag and drop usage in the Maestro designer.

A TPac is a simple archive file based on the ZIP format.

Exchange Package Configuration

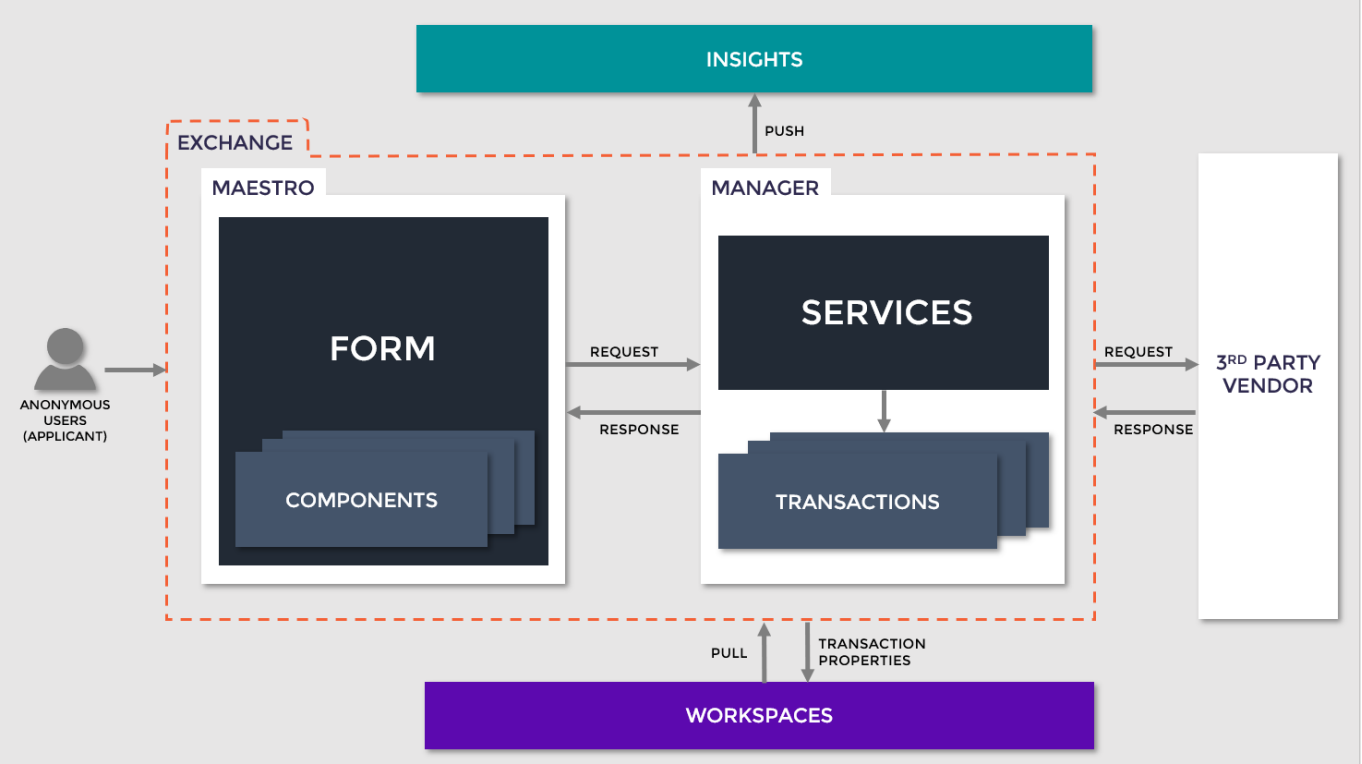
Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

Exchange package architecture

All Exchange packages are based on the same architecture and interact with Manager and Maestro in the same manner.

As the user interacts with an Exchange package implemented in their application (form), the third party services are called via Manager, and these events are recorded in Journey Analytics, the platform's analytics tool to be further analyzed in the future.

The transaction properties of the user's application are also sent to the Workspaces portal, so the application life cycle can be reviewed by the Workspaces users.



Package Configuration

Once an Exchange package has been installed from either the Exchange or a zip file (TPac), the Manager services and Maestro assets need to be configured. Most Exchange packages contain multiple experiences provided by the third party vendor, and it is this configuration that is

necessary for the package to behave as expected. Configuration is usually required in both Manager and Maestro.

Typical configuration tasks required in Journey Manager

Typically the configuration tasks required in Manager include the following:

- Add your third party credentials (username and password) to the service connection. Use the path All Services > required service > Service Connections.
- Update the Service Parameters (such as setting Max Attempts) on the service definition. Use the path All Services > required service > Service Definition > Parameters Edit tab.
- Configure any other settings as mentioned in the package documentation.
- Test each required service to confirm they all run successfully. Use the path All Services > required service > green arrow icon (Run Unit Test).

Some packages will also require additional configuration tasks in Manager if attachments are added to the transaction after the application is completed, such as DocuSign and AdobeSign. These additional configuration tasks may also include the following:

- Create a new scheduled service job and link it to the service status poller. Use the path System > Scheduled Jobs > New Scheduled Service Job.
- Create a new delivery channel and link it to the service delivery process. This will attach the signed document to the transaction in Manager. Use the path Forms > Organizations > Delivery Channel > New.
- Link the Maestro form to the new delivery channel. Use the path Forms > Forms > Details tab.

Typical configuration tasks required in Maestro

Typically the configuration tasks required in Maestro include the following:

- Add the Maestro component to the form.
- Click on the component in the Structure panel to configure the following items.
- Map the input data fields to the response fields in the Configuration section of the Properties panel.
- Set Role Identifiers under Options in the Configuration section of the Properties panel if multiple applicants can use the same application form.
- Set any other Options as required by the package to behave as expected.
- Check the progress, success, and failure messages provided in the component and update them in the Properties panel if required.

- Update the error messages to be consistent with the language on the form, if the form has been translated into a language other than English. Use the Translation button in the menu bar.
- Add any Rules in the Rules section of the Properties panel to change the behavior of the component when certain conditions are met, if required.

Exchange Development Schedule

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

Exchange Packages are released in waves, with a new wave about every 6 months.

The Exchange Packages list shows all Exchange packages that are already published and available globally. You can also filter the packages by region, to show only those packages that are available for use in your region.

Development Schedule

View the Exchange packages published during previous Temenos Journey Manager releases.

22.04

20.05

19.11

19.05

WARNING

Despite best efforts, the Exchange team cannot guarantee delivery of new integration packages, as our intended schedule undergoes frequent review and adjustment. Project teams should not build any dependency or reliance on these time frames into their implementation plans.

Unscheduled

The following packages have been identified as candidates for future cycles, but are not yet scheduled for release.

Package	Description
Bureau van Dijk	Global service providing company intelligence - https://www.bvdinfo.com/en-gb/home
DanalMobile Identity	Extend the Danalintegration to include additional mobile identity services - https://danalinc.com/platform/
Early Warn-	Online payments - https://www.earlywarning.com/

ing	
DigitalID	National identity verification service offered by Australia Post called DigitalID - https://www.digitalid.com/personal
Canada Post	International address finder using the Canada Post Web Service APIs - https://www.canadapost.ca/pca/
Experian Powercurve	Experian's Decision Management Solution - http://www.experian.com/strategy-management/powercurve.html
Experian CrossCore	The Experian CrossCore platform unifies a bunch of fraud and identity services into a single service - includes FraudNet, ID services and other 3rd party capabilities such as biometrics.
FaceMe	FaceMe is an omni-channel virtual assistant platform - http://www.faceme.com/
Fiserv FundNow	OpenNow®/FundNow® from Fiserv supports instant account opening and funding. - https://www.fiserv.com/resources/opennow-fundnow-brochure.aspx
Glass's Guide	Vehicle information services using the Glass's Automotive Business Intelligence system.
Mambu	Integration with the Mambu Saas Core Banking platform for straight-through-processing of product applications - https://www.mambu.com/
Microsoft Dynamics	Microsoft Dynamics lead generation.
QBE LMI	QBE Lenders Mortgage Insurance API provides LMI premium estimates for the Australian market.
ID Analytics	ID Analytics provides Credit and Fraud Risk Solutions & Analytics - http://www.id-analytics.com/
SuperMatch 2	SuperMatch 2 is an API provided by the Australian Tax Office to find Superannuation accounts registered against an individual.
Transunion	Integrate directly to the TransUnion Bureau.
ThreatMetrix	Fraud prevention using the ThreatMetrix suite of services - https://www.-threatmetrix.com/
Trulioo KYB	Trulioo KYB is KYC for businesses, a global service used to verify businesses entities in real time.
Vantiv	Online payments - https://www.vantiv.com/
Apple Pay	Register credit cards with Apple Pay at the time of online application approval for immediate use.
Auraya	Biometric voice signature using the Auraya Armorvox service.

Armorvox	
Proviso	API based income verification for the Australian market using the BankStatements.com.au service.
Cameo	Individual profiling and classification using the Cameo Online services.
Celebrus Analytics	An outgoing integration to Celebrus Big Data Analytics service.
Credit Reform	German based credit and insolvency information services - https://en.creditreform.de/index.html
DataVerify	IDV, Income verification and employment verification using the DataVerify services.
Delux Switchagent	SwitchAgent makes switching financial institutions in the US quick and easy.
DirectID	DirectID combines bank verification, live financial data, bureau checks and document authentication in a single platform.
Equifax Interconnect	Integrates Equifax federated Bureau, decisioning and identity checks into our platform.
Facebook Continued Flow	Prefill a Maestro form from a Facebook lead capture form.
Google Plus	Data prefill from an individuals Google Plus profile.
IDNow	European digital identification service supporting real-time video chat and e-signatures - https://www.idnow.eu/
Ignite Sales	Product recommendations & cross selling engine.
IKO	Swiss based credit services - https://www.iko-info.ch/de-ch/kredit-leasingnehmer
InAuth InBrowser	Fraud prevention service utilizing java script collectors, InBrowser provides device identification and risk assessment.
KickFire	IP address intelligence.
LiveAgent	Web chat using the Salesforce LiveAgent service.
Olark	Web chat using the Olark service.
OpinionLab	Customer feedback engine.
SAP SMS	Globla SMS messaging using the SAP SMS 365 service.
Schufa	German based risk / trust services for individuals and businesses including credit, fraud & compliance, decisioning - https://www.schufa.de/en

Socure	US identity verification service based on social bometrics as well as offline data.
Stripe	Stripe Payments is a payment service specifically designed for internet transactions.
Symitar	Core banking platform used by 40% of credit unions in the US.
Teminos	Core banking provider.
Tokenex	Pass sensitive information off to TokenEx and store a secure token in its place.
Twillow	Global SMS API
Vasco 2FA	Vasco 2 factor authentication services.
Equifax DP3	Integrates Veda Bureau, decisioning and identity checks into our platform using the Decision Point 3 platform.
Verifier	Income Verification http://www.verifier.me/ (a Dunn & Bradstreet FinTech partner)
WebId	German based real-time video chat identification service - https://www.webid-solutions.de/en/
Yodlee ACH	Yodlee's Account Clearing House (ACH) service.
ZEK	Swiss based credit services - https://www.zek.ch/de-ch/kredit-leasingnehmer
Zoot	Real-time decisioning system.

Contact the Exchange Team

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

You can contact the Exchange Team to suggest an idea or raise a support request through the Temenos Support portal:

Need to log an idea or issue?

The Support Handbook shows you how to log them in the Support Portal.

What we require

1. Ensure you're using the original Exchange package (not modified services or Maestro assets). *Support is only provided if the Manager services or Maestro assets have not been changed.
2. Include your Manager, Maestro and Exchange package versions.
3. Capture / reproduce the steps taken using a Maestro template (blank, Maguire, Blue Steel or Avalon). This is essential so that we can replicate the issue in our own environment.

How to log a support request

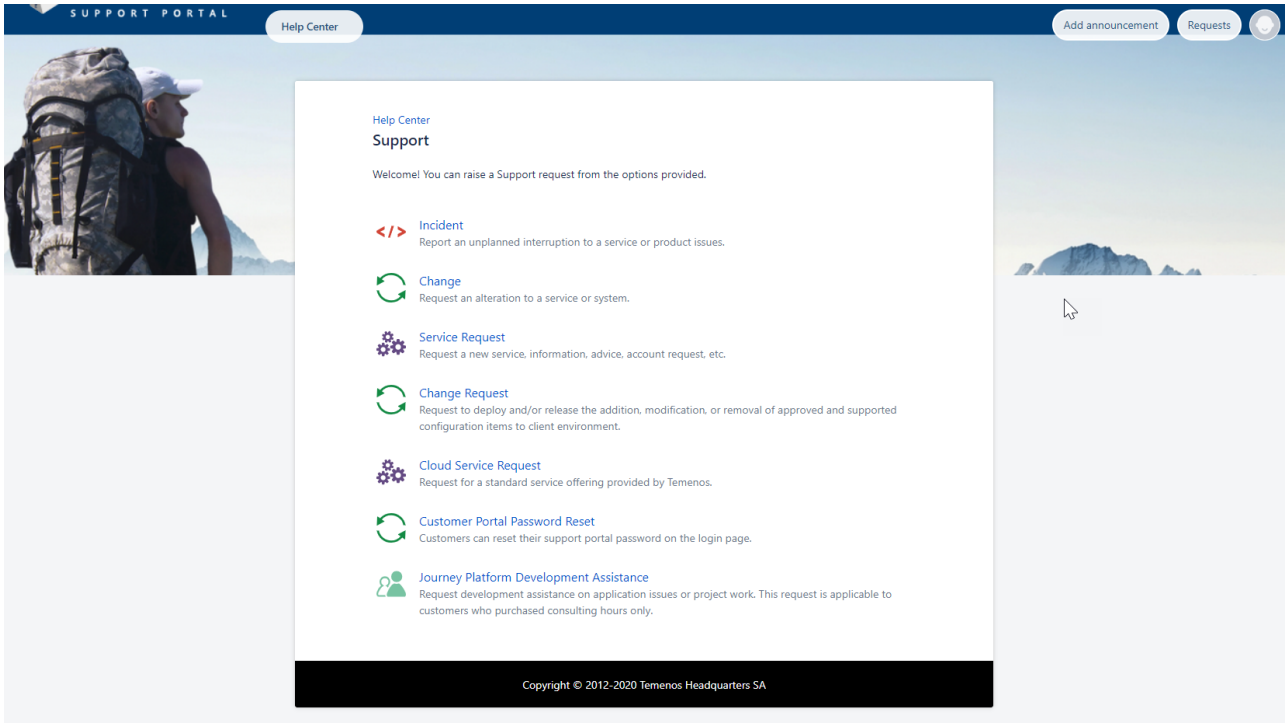
Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

If you have identified a defect or issue, the following information will be required to log a support request:

1. Ensure you're using the original Exchange package (not modified Manager services or Maestro assets). *Support is only provided if the Manager services or Maestro assets have not been changed.
2. Include your Manager, Maestro and Exchange package versions.
3. Capture / reproduce the steps taken using a Maestro template (blank, Maguire, or Blue Steel). This is essential so that we can replicate the issue in our own environment.

Log a support request

1. Log a defect or issue through the Support portal.



2. Click on Incident.
3. In Affects Product Version/s make sure you select Exchange, PLUS the Maestro and Manager versions.

4. Select the Environment that is affected.
5. Select the Severity.
6. Enter a brief explanation of the issue in the Issue Summary.
7. Enter all other details needed to describe this issue in Description.
8. Attach a zipped file containing the Maestro form in Attachment(s) if available at this point.
9. Click Create.
10. Our Support team will contact you if they need anymore information before passing this issue on to the Exchange Team.

NOTE

For more information on how to raise a support ticket see the Contact Support documentation in the Journey Manager Support Handbook.

Exchange Partner Program

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

Interested in becoming an Exchange Partner and publishing your own solutions?

The Exchange Partner Program is designed to allow your team to participate in the Exchange service by developing, certifying and publishing your own solutions.

Who can apply?

Any team wishing to build a package for publishing on the Exchange, or enhance an existing package, is eligible to apply for a partner account, including:

- Temenos Journey Manager (TJM) implementation teams producing reusable integrations to new 3rd party systems.
- 3rd party vendors wishing to build a connector to their own system.
- Internal teams producing generic and reusable assets that would benefit the broader community.

What is involved?

- The Exchange Publishing Process
- Exchange Certification
- Developer Resources

The Exchange Publishing Process

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

The process of publishing to the Journey Exchange involves four steps.

Step 1 - Apply

Apply for your Exchange Partner Account

Contact your Partner Account Manager. Your application will be reviewed and if successful you will receive access to:

- The Maestro design tool for UI development work.
- A sandbox Journey Manager environment for publishing and testing your solution.
- The Temenos Journey Manager (TJM) resources website with extensive product documentation, how-to articles and Q&A forum.
- Self-paced online training courses.
- An Exchange hosted GIT repository.

Step 2 - Build

Build your Extension Package on our platform

Develop your solution on the our platform using Maestro (our award winning UI design tool) and the Fluent SDK (our Groovy based development kit that can be used in any Java-based IDE).

Be sure to closely review the Exchange Development Guidelines as consideration of these guidelines will produce benefits when it comes to certifying your package.

You may use your own source code management system, however we recommend that you use the Exchange hosted GIT repository as your solution must be pushed to this repository before submitting for certification.

For technical support during your build phase, the Temenos Journey Manager (TJM) resources website is the best resource. You will also receive limited support from the Exchange technical team or the Support team if required.

See Exchange Developer Resources

Step 3 - Certify

Submit your Package for Exchange Certification

The Exchange team maintains a set of minimum requirements that every package must satisfy before it can be published to the Avoka Exchange. To ensure the standards are met, all packages must be submitted through the Exchange Certification Process.

To streamline this certification process and minimize any delays, make sure that your project team have reviewed the Exchange Certification Checklist and considered all items in your solution. If anything is uncertain it is better to seek clarity before submitting for certification than it is to submit a non-compliant solution and receive a certification failure.

When you submit your package for certification, all assets must be available in the Exchange GIT repository. Your submission will be added to the work queue for the Exchange team and scheduled for review. This review may take up to 6 weeks and may require one or more scheduled conference calls between the certification team and your project team to discuss aspects of your solution.

At the completion of the certification review, a report will be issued detailing the outcome of the review (CERTIFIED or UNCERTIFIED) and may contain a number of feedback items classified as either:

- Comment - A benign note made by the certification team relating to your solution. Comments do not impact the outcome of the review;
- Recommendation - A suggested (but not required) modification to the solution that should be considered by your project team; or
- Requirement - A mandatory requirement that must be satisfied by your solution before it is certified. The presence of any such items will result in an UNCERTIFIED outcome.

Any solution that does not get certified on the first submission may be resubmitted once the feedback items have been addressed.

See Exchange Certification

Step 4 - Publish

Release your Package to the Avoka Exchange

When you have a certified package you will be asked to provide information required to get your package published to the Avoka Exchange. This may include descriptive content and imagery, reference links, demo videos etc. We strongly recommend you create a functional demo of your solution so that customers can try it out and see how it works before installing it. Demos can be hosted on the Exchange Demo Server and demo links can be added to your listing.

When your provider profile and package listing are ready, you can submit a request to activate your listing. Once activated, the listing will be immediately available on the Avoka Exchange for all customers.

Exchange Certification

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

During the Exchange Publishing Process you will be required to get your package certified before it can be made active on the Avoka Exchange. When you submit your package for certification it will be added to the work queue for the Exchange team and scheduled for review. This review may take up to 6 weeks and may require one or more scheduled conference calls between the certification team and your project team to discuss aspects of your solution.

The Exchange team will review your package to ensure that it meets certain standards and minimum requirements in terms of how it is designed and built. We consider many aspects of the solution you've produced, including those listed under Certification Checklist below. We encourage you to become very familiar with this checklist and ensure that you are considering these items from the outset of your project so that delays are minimized when you wish to publish your solution on the Avoka Exchange. If anything is uncertain it is better to seek clarity before submitting for certification than it is to submit a non-compliant solution and receive a certification failure.

When you submit your package for certification it will be added to the work queue for the Exchange team and scheduled for review. This review may take up to 6 weeks and may require one or more scheduled conference calls between the certification team and your project team to discuss aspects of your solution.

Certification Checklist

Before submitting your package for certification, ensure you have reviewed the following checklist:

1. Project pushed to Exchange GIT Repository.
2. Adequate Package Documentation.
3. Security of Customer Data.
4. Language Translation Support.
5. Package Compatibility Requirements.
6. Unique Asset Naming Strategy.

7. Best practices used as per Groovy Coding Conventions & Practices, in particular:
a. Naming Conventions for Groovy Services.
b. Logging Third Party Service Calls.
c. Encapsulating Common Code in Shared Groovy Classes.
8. Maestro Library named exchange.<project-name>
9. Maestro Component Design Considerations:
a. Responsive design - make sure it looks and behaves appropriately on all screen sizes.
b. Data Binding and no binding where appropriate.
c. Can be used more than once in a single form, and in repeats with no duplicate binding issues.
d. Receipt presentation considerations - how should this appear on the receipt?
e. State retained between save and resume events.
f. Rule templates - consider adding rule templates so users of the component can add code for specific triggers.
g. Rule helpers - think about right click menu and any features to expose.
h. Ensure no errors in JavaScript console during use.
10. Journey Analytics Considerations:
a. Insights Field Names configured in Maestro to override original field names and add clarity to duplicate field names shown in Field Analysis View.
b. Journey Analytics Milestones (e.g. GreenID VERIFIED) - test that they are being sent.
11. Sufficient test cases to cover usage scenarios.
12. Review of TODOs.
13. Demonstration form provided in Maestro project.

Certification Report

At the completion of the certification review you will receive a report detailing the outcome of the review and feedback relating to the areas highlighted by the review team.

The report will detail the overall outcome of the review (CERTIFIED or UNCERTIFIED) and may contain a number of feedback items classified as either:

Comment - A benign note made by the certification team relating to your solution. Comments to not impact the outcome of the review;

Recommendation - A suggested (but not required) modification to the solution that should be considered by your project team; or

Requirement - A mandatory requirement that must be satisfied by your solution before it is certified. The presence of any such items will result in an UNCERTIFIED outcome.

Any solution that does not get certified on the first submission may be resubmitted once the feedback items have been addressed.

Adequate Package Documentation

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

Each package on the Exchange catalog must be adequately documented so that customers can be self-sufficient when installing, configuring and using the package. This article describes the areas where documentation should be applied in your package.

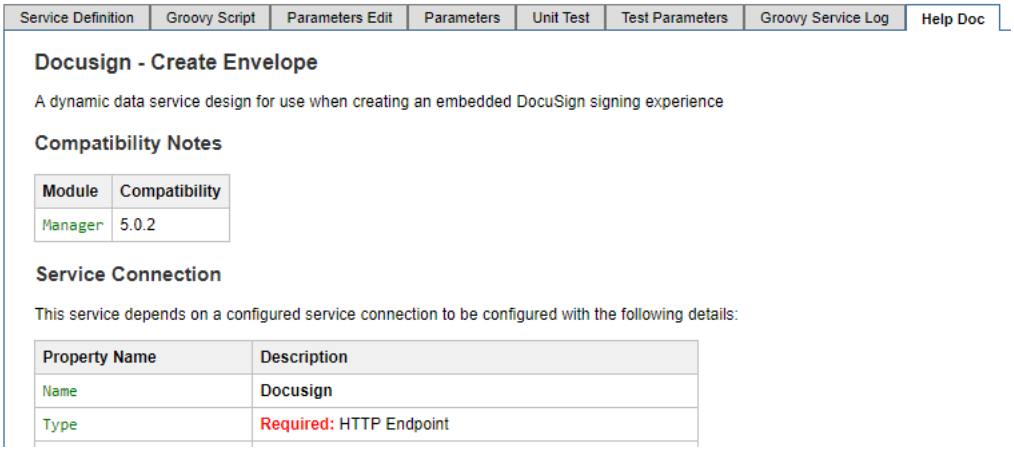
Fluent SDK Document Generation

When using the Fluent SDK in the IDE, documentation is generated based on the assets found in the project source folder and the descriptive content added by the development team.

There are 2 levels of generated documentation produced by the Fluent SDK:

1. **Service Level Documentation**

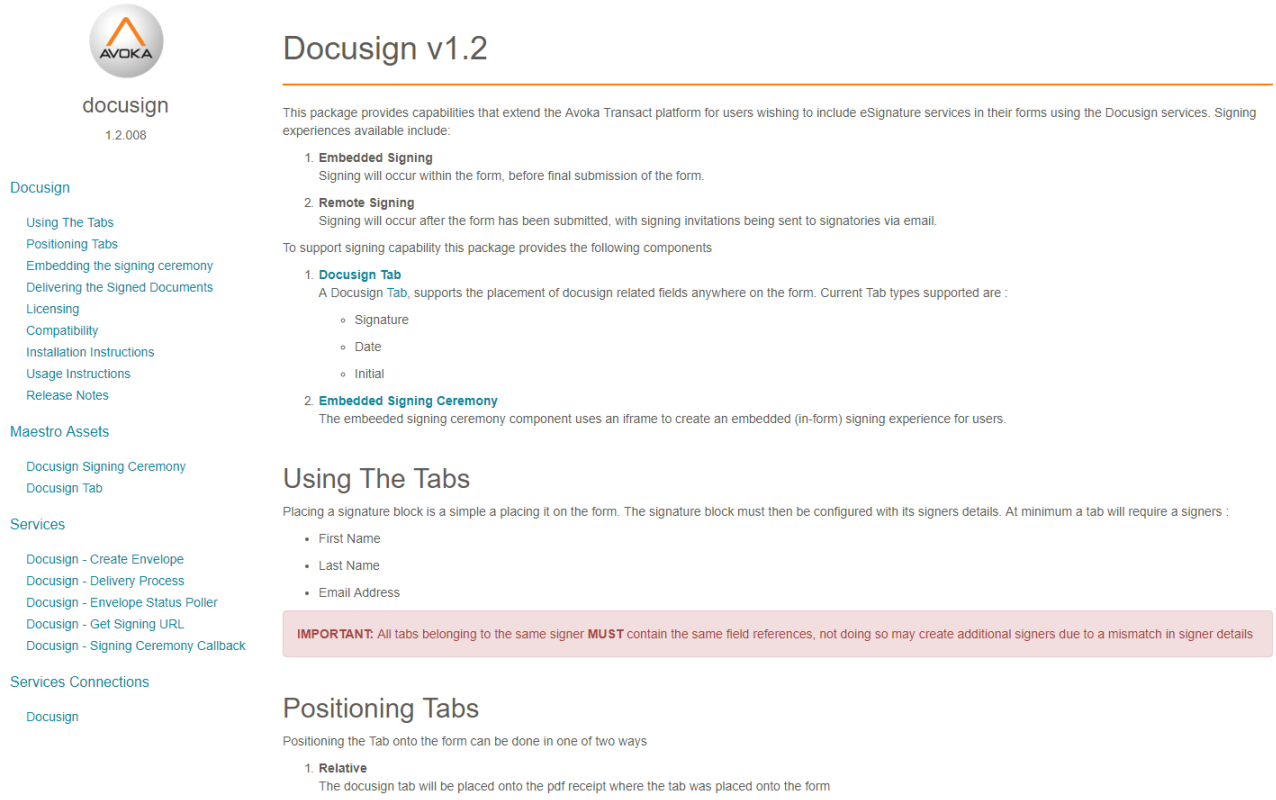
   In the source folder for each Groovy service you will find a **service-help.html** file which is generated each time you package or deploy your service based on the annotations in the Groovy script itself and the descriptive content in the **service-def.json** file. When deployed to Journey Manager this documentation is visible on the **Help Doc** tab when viewing the service definition.



   See ServiceDoc Annotations to learn how to embed documentation in your Groovy script using annotations.

2. **Package Level Documentation**
   The package documentation generator pulls information from a range sources in the project source folder to produce a comprehensive manifest and guide for users of the

package. Much of this information is created based on system generated files but must be augmented with descriptive content by the package developers. When a TPac is deployed to Journey Manager (under **System** → **TPacs**) the **readme.html** document will be accessible via the package listing.



Documenting Groovy Services

The documentation generator for Groovy services pulls information from the following 2 sources located within the service source folder:

1. The annotated Groovy script - see ServiceDoc Annotations.
2. The service-def.json file is inspected to locate any service parameters that have been added and the parameter JSON definitions are used in the generated service-level documentation. Ensure you provide a clear description for each service parameter you add.

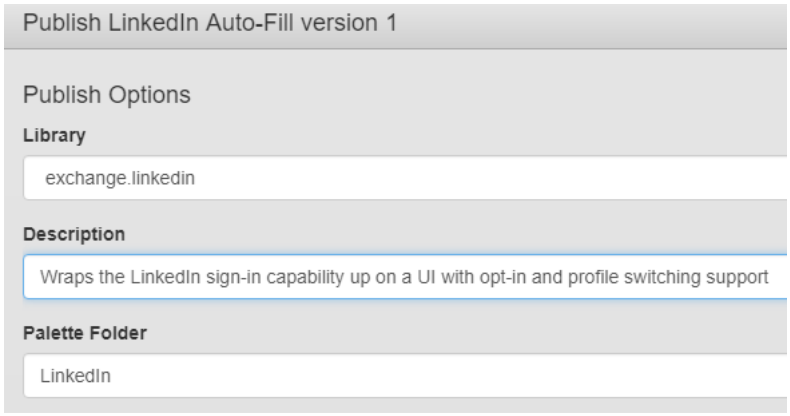
Example service parameter definition from a service-def.json file

```
{
  "name": "Docusign Email Subject",
  "description" : "The email subject to display for Docusign email delivery",
  "type": "String",
  "value": "Docusign $formName",
  "bind": false,
  "required": false,
  "clearOnExport": false,
  "readOnly": false
 }
```

Documenting Maestro Assets

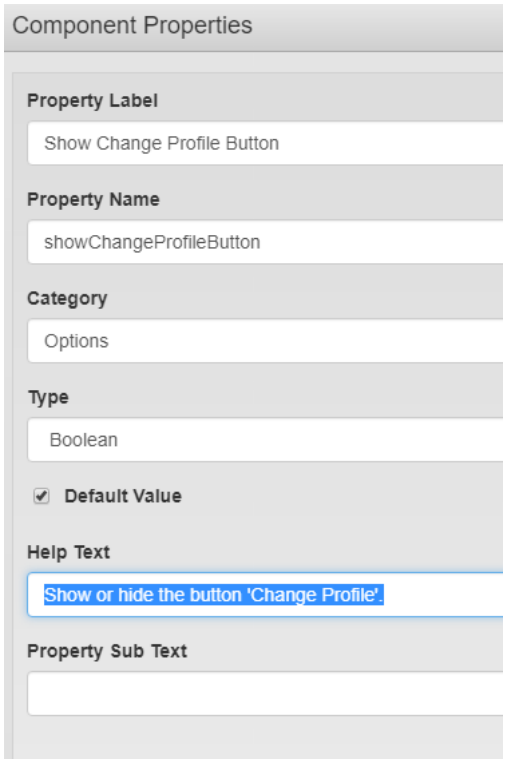
All descriptive content for Maestro assets should be added by the developer via the Maestro UI. Here we list some examples of the descriptive content contained in Maestro assets that is used in the generation of the package-level documentation:

- Shared Component Descriptions - when you publish a custom component to a Maestro library you are able to specify a description of the component.



- Component Properties and Rule Templates - adding properties or rule templates you are

also able to add descriptive content.



The following is an example Maestro component documentation generated into the package-level readme:

Maestro Assets

Docusign Signing Ceremony Library: exchange.docusign Category: Docusign

A widget for embedding the docusign signing ceremony inside the form

Rule Templates

You may add your own logic to handle the following rule types triggered by this component:

- **On Complete :** A script to run when all signers have completed their signing
- **On Failure :** A script to run when the signing ceremony has failed and cannot be complete

Properties

Property	Category	Description	Type	Default
Height	Configuration	The height of the iFrame the ceremony will appear in	integer	`300`
Progress Message	Configuration	The message to display while the agreement is being generated for signing	text	`Please wait while we prepare your agreement`

Package Overview Documentation

In addition to the automatically generated documentation on Groovy services and Maestro assets, package developers can inject free-form overview instructions and guidelines into the package documentation by editing the package.html file located in the project source folder. All content in this file is added at the top of the package readme.html file and any <h2> headings are treated as a navigable menu item, being added to the left hand navigation.

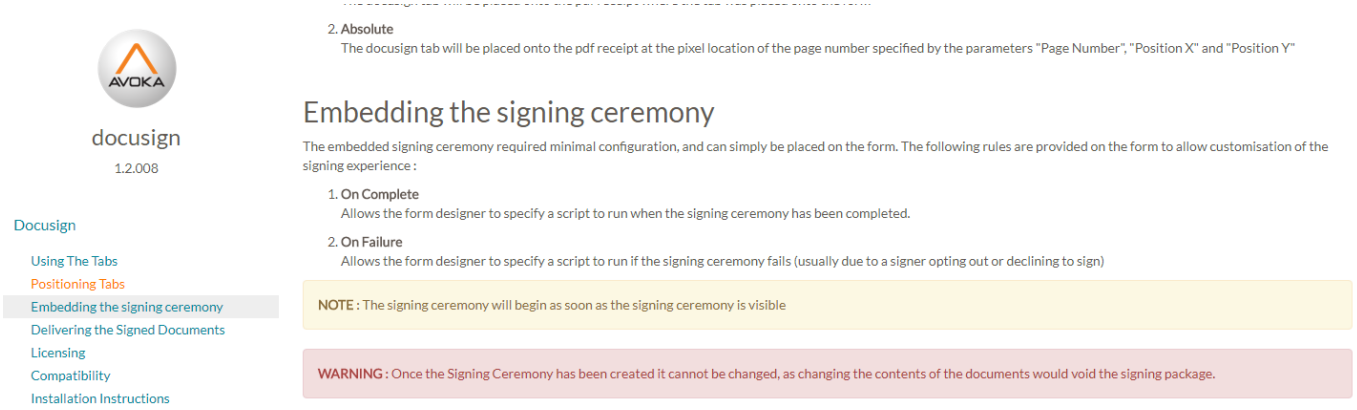
In the package.html file, developers should provide adequate documentation to enable customer to be self-sufficient in their use of the package. This may include sections on:

- The purpose of the package and the use cases it supports
- Compatibility requirements for the package core modules and any 3rd party dependencies
- Licensing information and contact details for obtaining a license
- Installation and usage instructions
- Details around special features supported by the package
- Release notes detailing all relevant historical releases and the main change between releases
- Legal considerations for those intending to use the package

Example Custom Section

```
            <h2>Embedding the signing ceremony</h2>
            <p>
                The embedded signing ceremony required minimal configuration, and can simply
be placed on the form.
                The following rules are provided on the form to allow customisation of the
signing experience :
                <ol>
                                <li>
                                <strong>On Complete</strong>
                                <p>Allows the form designer to specify a script to run when the signing
ceremony has been completed.</p>
                        </li>
                        <li>
                                <strong>On Failure</strong>
                                <p>Allows the form designer to specify a script to run if the signing ce
mony fails (usually due to a signer opting out or declining to sign)</p>
                        </li>
                </ol>
             <p class="alert alert-warning"><strong>NOTE</strong> : The signing ceremony
will begin as soon as the signing ceremony is visible</p>
             <p class="alert alert-danger"><strong>WARNING</strong> : Once the Signing
Ceremony has been created it cannot be changed, as changing the contents of the documents
would void the signing package.</p>
            </p>
```

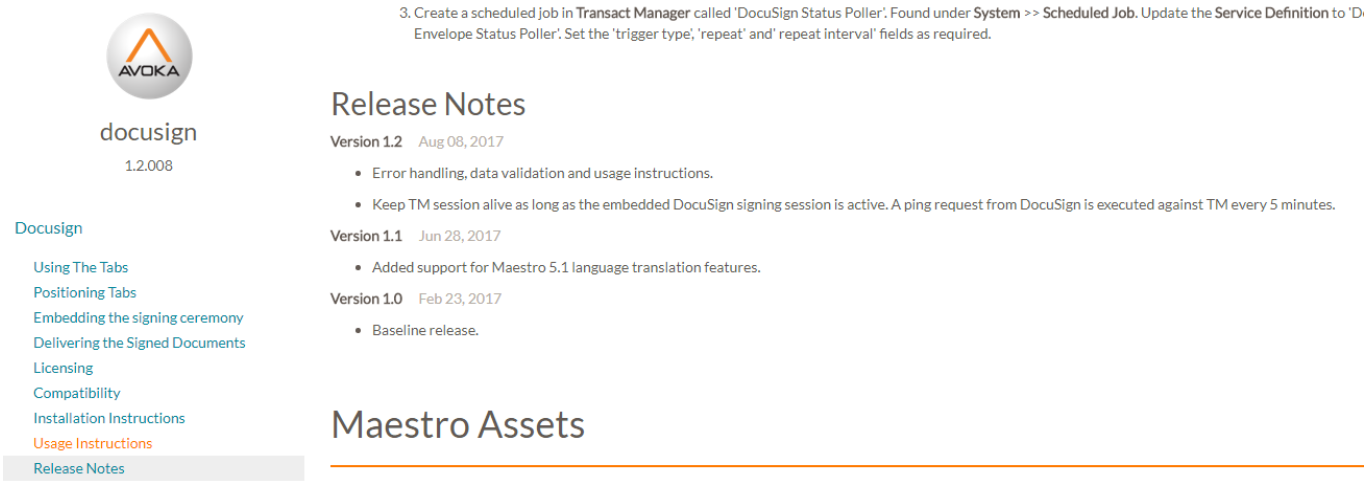
Example Custom Section



Example Release Notes

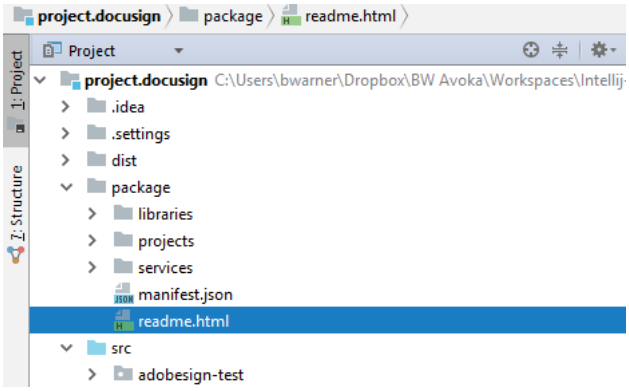
```
            <h2>Release Notes</h2>

            <p><strong>Version 1.2</strong>  <span class="label label-default">Aug 08,
2017</span></p>
            <ul>
               <li>Error handling, data validation and usage instructions.</li>
               <li>Keep Manager session alive as long as the embedded DocuSign signing ses-
sion is active. A ping request from DocuSign is executed against Manager every 5 minutes.
               </li>
            </ul>

            <p><strong>Version 1.1</strong>  <span class="label label-default">Jun 28,
2017</span></p>
            <ul>
               <li>Added support for Maestro 5.1 language translation features.</li>
            </ul>

            <p><strong>Version 1.0</strong>  <span class="label label-default">Feb 23,
2017</span></p>
            <ul>
               <li>Baseline release.</li>
            </ul>
```

Example Release Notes



Tip

You should occasionally run the package generator and review the content to ensure that there are no gaps or errors in the documentation. You can manually trigger the generator by running the generate-package-docs Ant target in the project build file. The generated readme.html will be generated into the package folder in your project.



See also Working with the Ant Build Files

Language Translation Support

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

Journey Manager has in-built support for language translation, introduced in Maestro 5.1.0, so the form user may interact with the system in other languages while maintaining the same great user experience. In order to support this multi-lingual capability across the board, package developers are required to review their package for translatable content and ensure that it is appropriately flagged so that it is picked up by the translation engine. Fortunately this is a relatively simple process that is made easy by platform tools as follows.

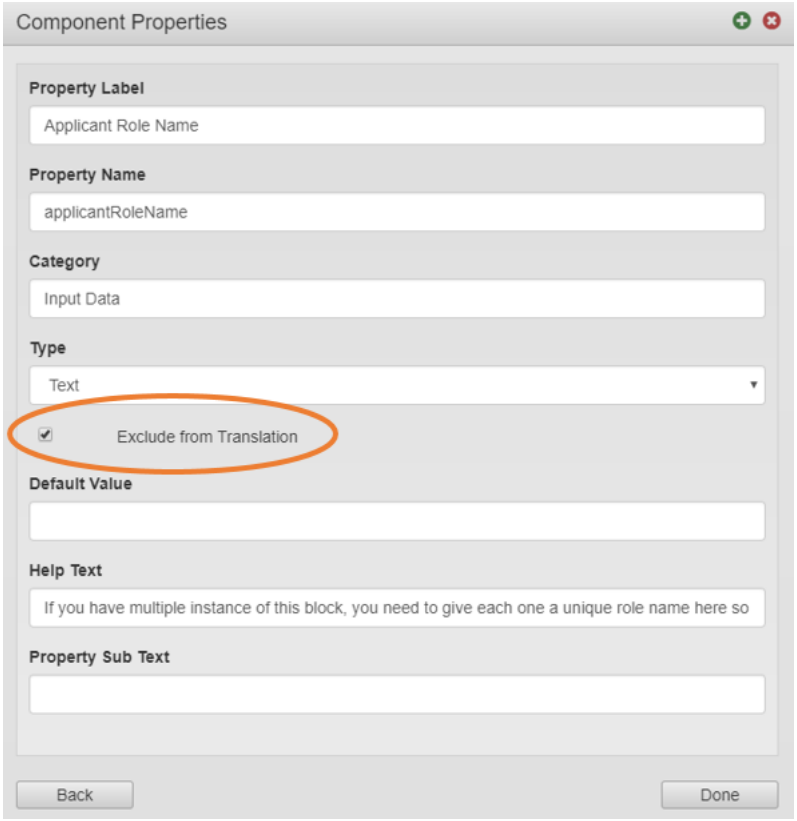
Maestro Standard Components

All standard Maestro components (provided in the Maestro release libraries) are already configured for translation so you are not required assess these in your review. This includes all visual elements of standard fields including labels, captions, help text and error messages.

Maestro Custom Components Properties

Where you have created a custom component that has its own component properties you will need to review these for translation support as follows:

1. Open the Component Properties dialog and inspect each property of type Text, Rich Text or Help Text.
2. Determine if this field should be translatable. If it is a text value that is presented to the user then it will likely require translation, but you may have text properties that should not be translated under any circumstances (e.g. country codes, insights milestone event names, api keys etc.).
3. Open the component property configuration and set the Exclude from Translation flag according to the determination made in step 2 above.

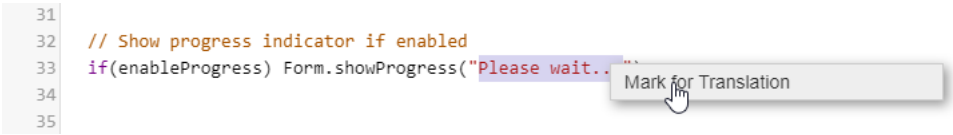


4. Save.

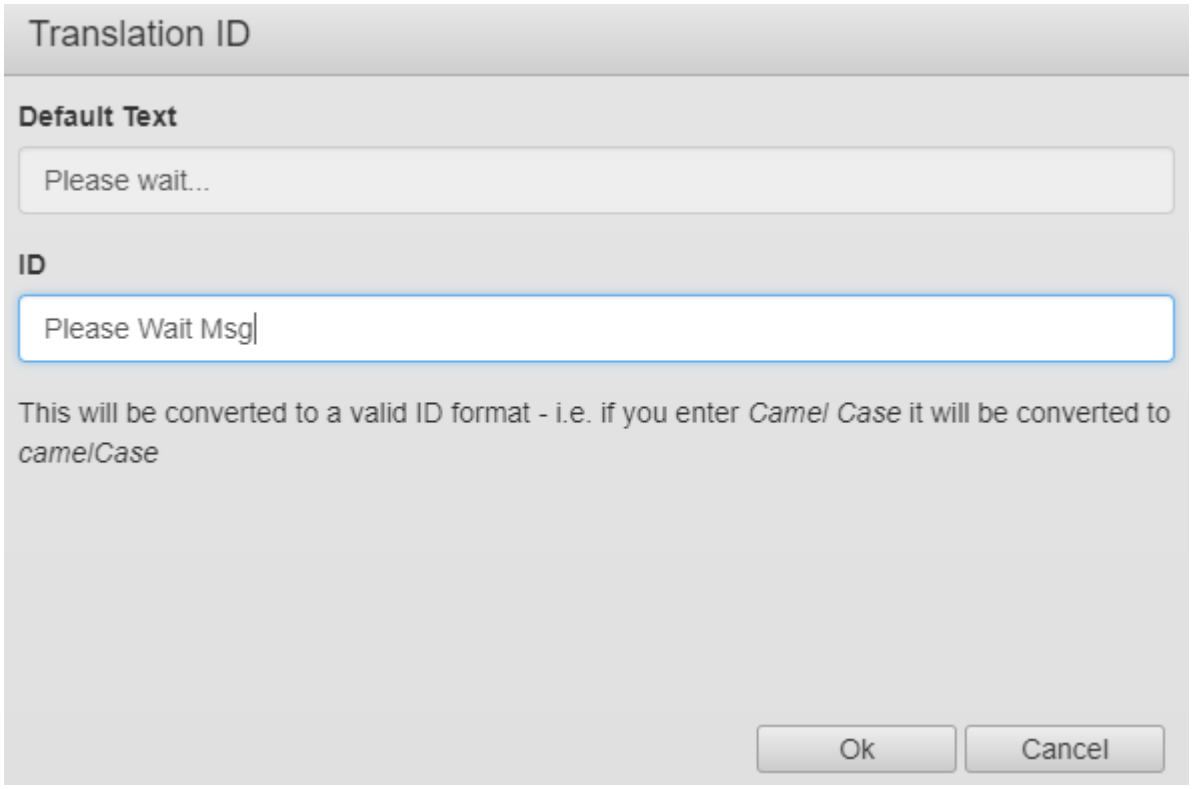
Maestro JavaScript String Literals

Where you have used JavaScript code in custom components you must review these scripts for hard-coded string literals and manage the translation support as follows:

1. Determine if the string literal should be flagged for translation support (i.e. will it be shown to the user of the solution?).
2. If translation is required, right click on the string in the Maestro script editor and choose Mark for Translation.



3. In the Translation Dialog that appears, give the field a translation ID:



4. Note that a translation tag has been pre-pended to the string literal, which alerts the translation engine to include this string in the translation files:

```
31
32  // Show progress indicator if enabled
33  if(enableProgress) Form.showProgress("~T~pleaseWaitMsg~T~Please wait...");
34
```

5. Save.

Switching language at runtime

It's worth noting that the component should be able to show the correct language when switching the form language at runtime. For example, if the component want to display a error message based on certain error code. Typically you need to provide a javscript to do the mapping and shows the related error message based on error code. An example would be:

Example

```
            function getErrorMessage(errorCode) {
               switch (errorCode) {
                       case "NF":
                               return '~T~wewereunabletodetectthebarcodefromthebackofyourlicense~T~We
were unable to detect the barcode from the back of your license.';
                               break;
                       case "OOF":
                               return '~T~theimageistooblurry~T~The image is too blurry.';
                               break;
                       case "DARK":
                               return '~T~thereisnotenoughlightonyourdocument~T~There is not enough lig
on your document.';
                               break;
               }
            }
```

The correct pattern that supports switching error message at runtime would be having two data fields, one for the error code, one for the error message. When a service gets called and returns with error code, we set the error code value into error code data field. And we also create a calculation script for the error message data field to get the correct error message based on error code like function above. This way, everytime a user changes the form language, the calculation script gets triggered so that error message always presents in current language.

Maestro Item Property References

If you are accessing component properties in Maestro JavaScript rules and those properties are translatable, ensure you are accessing them using the Form.getItemProperty() JavaScript function not the item.properties object.

Example

Do not use:

- item.properties.progressMessage

  in your scripts, but instead use

- Form.getItemProperty("data.someItem", "progressMessage")

Do not use:

- item.$$component.properties.successMessage

  in your scripts, but instead use

- Form.getItemProperty("data.myParentComponent", "successMessage")

Package Compatibility Requirements

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

This article describes how to configure the minimum compatibility requirements of the assets in your Exchange package.

There are a number of areas where the compatibility requirements for your package must be specified, as detailed below.

Groovy Service Compatibility

In your service source folder you will find a service-def.json file which contains some configurations for your service. Groovy services run in the Journey Manager service container and so the minimum version of Journey Manager required to run the service must be specified. To configure the compatibility of your service you need to set a top level property named tmMinVersion with a value that specifies the full Manager version notation (e.g. 5.1.4).

This property needs to be set for each service in your package. If you are unsure if your service is compatible with versions of Manager earlier than the version you are developing against then it is safest to configure your development version in this property.

Example service-def.json

```
            {
               "name": "Docusign - Create Envelope",
               "description": "Create a Docusign Envelope for an embedded signing exper-
ience",
               "type": "Dynamic Data",
               "version": 1,
               "serviceConnection" : "Docusign",
               "tmMinVersion": "5.0.2",
               "parameters": [
                              {
                              "name": "groovyScript",
                              "filePath": "DocusignCreateEnvelope.groovy",
                              etc...
                        }
                         ]
            }
```

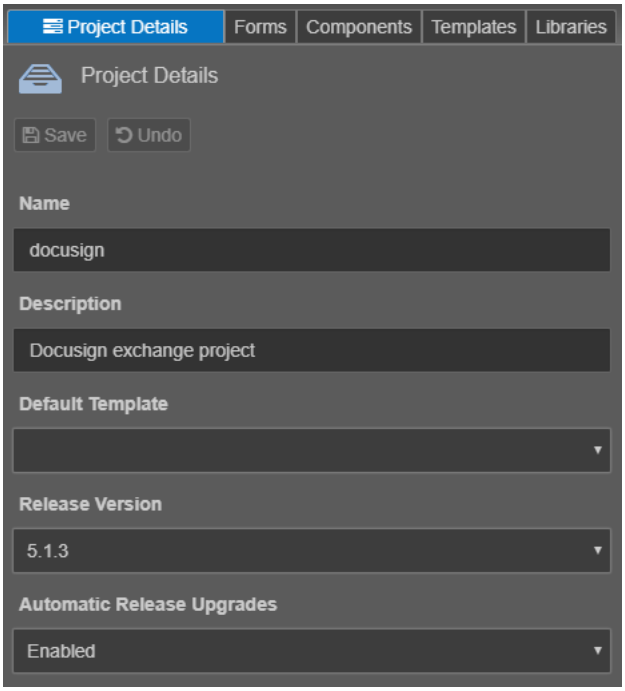
See Configuring Service Definitions with service-def.json for more detail around the service-def.json file.

This version should also be annotated in the Groovy script itself using the @since annotation. See ServiceDoc Annotations for more detail around the use of annotations.

Maestro Library Compatibility

If your package contains a Maestro library then you will need to specify the minimum (and optionally maximum) release version of Maestro that will support the library. Once you have downloaded the Maestro library to your IDE you will find a .library-config.xml file in your library source folder. To configure the minimum Maestro release version required you will need to set a top level attribute named releaseMinVersion with the full release notation of the supported Maestro release (e.g. 5.1.3).

If you are unsure if your Maestro package is compatible with release versions earlier than the version you are developing against then it is safest to configure your development version in this attribute. You can find your development release version in Maestro by looking at the project details tab as shown below.



You may also specify a minimum Manager version if it is relevant by setting the tmMinVersion attribute at the top level.

Example .library-config.xml

```
<?xml version="1.0" encoding="UTF-8"?><Library databaseVersion="1687" envir-
onmentName="Test50 Environment" exportDate="2017-02-23" revisionNumber="4abd604" sfmVer-
sion="5.0.3">
       <description>Docusign Widgets</description>
       <readOnlyFlag>false</readOnlyFlag>
       <releaseMaxVersion/>
       <releaseMinVersion>5.0.14</releaseMinVersion>
       <tmMinVersion>5.0.2</tmMinVersion>
       <createdAt type="Date">1478130023000</createdAt>
       <createdBy>---@avoka.com</createdBy>
       <lastModifiedAt type="Date">1487732038000</lastModifiedAt>
       <lastModifiedBy>---@avoka.com</lastModifiedBy>
       <name>exchange.docusign</name>
       <type>Organization</type>
</Library>
```

Package documentation

In addition to the configuration values you need to document the minimum versions in the package documentation.

In your project source folder you will find a package.html file that contains a section on Compatibility. You will need to edit that section to specify the minimum requirements at the package level. For example, if your project has 2 groovy services requiring Manager 5.0.2 and one Maestro library requiring Manager 5.0.8, then your package documentation should specify the largest version (5.0.8).

In this section you should specify any compatibility requirement for Journey Manager and Maestro, as well as any compatibility requirements for 3rd party solutions.

Example package.html

```
                ...
<h2>Compatibility</h2>
<p>
       This package has the following compatibility requirements:
</p>
<div class="table-responsive">
       <table class="table">
               <thead><tr><th>Module</th><th>Compatibility</th><th>Notes</th></tr></thead>
               <tbody>
               <tr><td>Journey Manager</td><td>5.0.2 or above</td><td></td></tr>
               <tr><td>Journey Maestro</td><td>5.0.14 or above</td><td></td></tr>
               </tbody>
       </table>
</div>
...
```

Security of Customer Data

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

When developing solutions for the Temenos Journey Manager it is critical that personally identifying information[1] (PII) provided by users in the process of completing a form are kept secure. This is an area that will receive close scrutiny during the certification process.

As a general rule, any type of storage of customer data outside of the form data payload should be avoided or at least minimized, but we recognize that there are some situations where customer data is required to be stored elsewhere for auditing, debugging and reporting purposes. This article describes the practices that should be avoided and provides recommended strategies to accommodate some common use cases.

Unacceptable practices

The following practices are considered high security risks and will result in a certification failure if identified:

- Logging of PII to the Journey Manager Event Log using EventLogger.
- Logging of PII to the Journey Manager Error Log using EventLogger.
- Logging of PII to the Journey ManagerService Call Log using TxnUpdater.addServiceCallLog()

Storing customer data in these these logs is considered high risk as the contents are used for monitoring and operations and are often stored for long periods in off-site databases. Storage of customer data in these logs will not be accepted under any circumstances.

Common use cases

1. Sensitive 3rd Party Data - where information about an individual has been retrieved from 3rd parties (such as credit agencies, fraud services, banks etc.) and the sensitivity of this data requires that it is not returned to the client browser where it can be interrogated and

[1]Personally Identifiable Information (PII) is information about an individual that can be used to distinguish or trace an individual‘s identity, such as name, social security number, date and place of birth, mother‘s maiden name, or biometric records; and any other information that is linked to an individual. In Europe, PII is known as personal data.

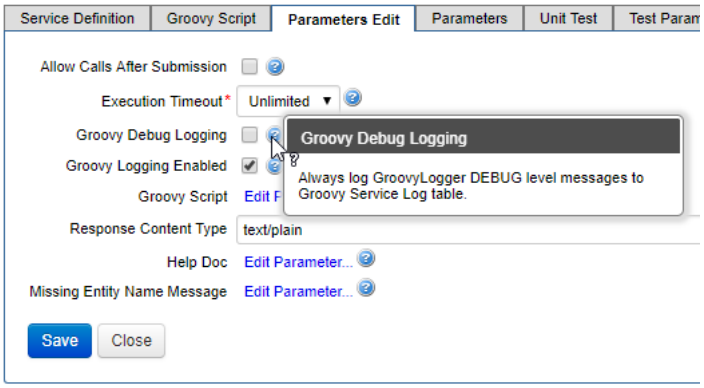
manipulated by savvy users.

2. Audit Logging - where the information used in certain actions during an application process needs to be retained for audit purposes (e.g. an audit of each call made to an identity verification service and the resulting response from that service).
3. Debug Logging - often when trying to locate unexpected behavior in your solution the ability to activate and access detailed debug information helps to identify the issue and resolve it. This type of information is usually only required during the period of debugging and should not be always activated.
4. Reporting - where information gathered from many related submissions can be quickly loaded into a CSV for simple reporting purposes.

Acceptable practices

The following practices may be used to accommodate the use cases where storage of customer data outside the form data payload is required:

1. Submission Properties - Data can be stored in submission properties using the TxnUpdater.setProperty() function in the Fluent SDK. These submission properties are subject to the same purge rules configured for the submission XML data. Submission properties can be accessed by authorized Manager administrators via the Admin Console by viewing the submission record and selecting the Properties tab.
2. Submission Data Extracts - Where reporting of form data across all transactions for a particular form is required you are able to create submission data extracts and quickly generate CSV reports on demand. Like submission properties, this extract information is accessible by authorized Manager administrators and is subject to the same purge rules as the submission XML data.
3. Groovy Logging - using GroovyLogger.debug() function in the Fluent SDK. Note that the all debug() statements are ignored by the Groovy Logger unless you have enabled Groovy Debug Logging for your Groovy service. This supports the use case where you would like to enable additional logging for the period of investigation only, then disable that logging when the investigation is complete.



See also [Data Retention Management]

Unique Asset Naming Strategy

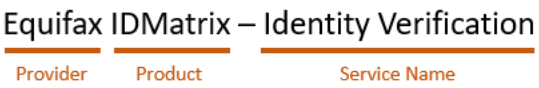
Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

In many areas Journey Manager uses the name of assets (or normalized name) to uniquely identify them within an Organization scope, therefore it is important to ensure that you use an asset naming strategy that avoids clashes with other similar solutions that may be deployed in the same organization. It is important to get this right from the start of your project because it is sometimes difficult and disruptive to modify this late in the development program.

Groovy Service Naming

By way of example, if you are building an integration to an identity verification provider you will have a Groovy service that performs the remote call to execute the ID check. You should name your Groovy service in a manner that will not clash with similar functions from other identity verification providers on the market. Naming your service 'Identity Verification' will not ensure uniqueness so we ask that you use a naming strategy that includes a specific identifier to avoid clashes. This also helps to locate and distinguish services from each other when searching in the Journey Manager console.

We recommend prefixing your service name with the target provider and/or product name. For example:



Some examples of Exchange packages include:

- Equifax IDMatrix - Identity Verification
- LexisNexis InstantID - Identity Verification
- Trulioo Global Gateway - Identity Verification
- VIX Verify GreenID - Identity Verification
- FIS ChexSystems - QualiFile
- ClickSwitch - Enroll Customer

- ESignLive - Create Package
- Melissa Data - Address Verification

NOTE

Do not include redundant information such as the type of asset. For example, do not add the word Service to your service name Melissa Data - Address Verification Service.

Service Connection Naming

Service connections should be named according the name of the external endpoint being accessed, for example:

- Equifax IDMatrix
- Au10tix
- DocuSign
- Google Places
- Iovation
- Melissa Data
- PCA Predict
- Yodlee

See also Managing multiple service endpoints and credentials for external service calls.

Maestro Component Naming

When creating Maestro components, the name you give it initially is normalized to create a unique component identifier which is used in all references to (uses of) that component. Please ensure you plan for the components you are going to build in your package and determine an appropriate naming strategy before you start work as changing this unique identifier can be disruptive.

We recommend front-loading the target provider and/or product name in the component name. Some examples of Exchange packages include:

- Equifax IDMatrix Verify
- LexisNexis InstantID
- LexisNexis InstantID Q&A
- LinkedIn Auto-Fill

- Mitek Mobile Fill
- Iovation Verify

When creating a new component you should also be specific about the Palette Folder that you want it to appear in. It is often sufficient to name the palette folder to the provider being integrated. Typically, all Maestro components for a single integration package will be located in the same palette folder. Some example palette folder names include:

- Mitek
- Liveperson
- LexisNexis
- CoreLogic
- Equifax

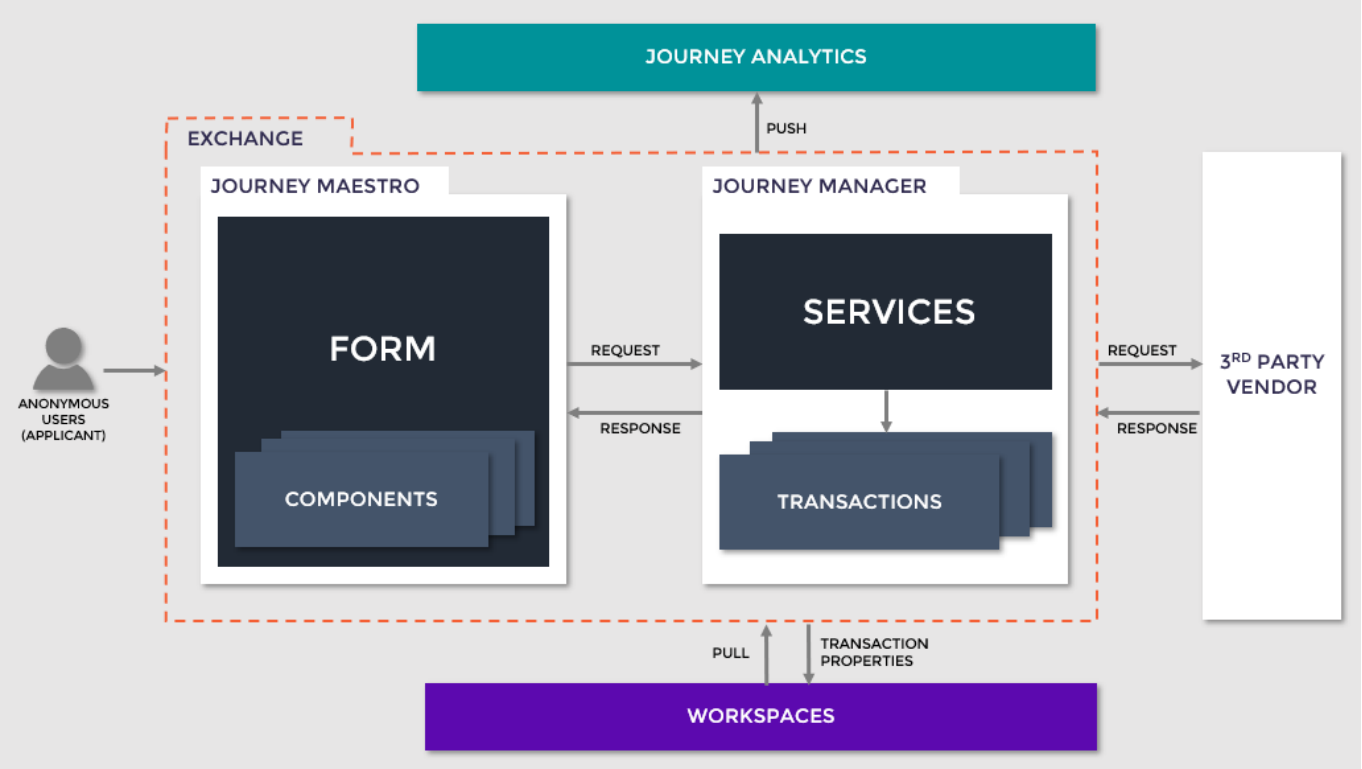
Palette folder names can be easily changed if required.

See also Maestro Shared Components, Maestro Native Components.

Exchange Developer Resources

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

Exchange package architecture



Developer Resources

Before starting your package development we strongly recommend you review the following articles.

Exchange Framework

- Exchange Framework

  The Exchange Framework (TIF) makes it easier for developers to build integrations on Journey products in a more efficient, standardised and scalable manner. Everyone is encouraged to use this framework to build their next integration.

- Transact Fluent API Javadoc

A Javadoc reference describing the packages, classes and more that you can use to develop a Journey Manager solution.

- Coding Conventions & Practices

  It is strongly recommended that you use these conventions and practices in your development effort.

- Techniques for Solving Common Requirements

  Some handy tips for dealing with common scenarios in Groovy.

- Working with the Exchange Sandbox Environment

  The sandbox environment is available for you to use during your development, to publish and test your solution.

- Accessing the Exchange GIT Repository

  You are required to push your project source file to the Exchange GIT Repository prior to the certification process.

Certification

- Exchange Certification

  Ensure you understand what is required to get your package certified.

Legacy

- Getting Started with Fluent Groovy Services

  Ensure you have your IDE configured and you've gone through the Hello World Example.

- The Exchange Domain Shared Code Library

  A library of reusable Groovy classes that are used extensively by the Exchange team.

Exchange Framework

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

Exchange Framework (TIF) v1.5.0 release

The Exchange Framework (TIF) makes it easier for developers to build integrations on Journey products in a more efficient, standardised and scalable manner to reduce the ongoing costs of upgrades and maintenance. Everyone is encouraged to use the framework to build their next integration.

The Exchange Framework v1.5.0 is a significant release with two major features:

- A standard fluent function base class for integration purposes that provides the most commonly used code for a typical integration fluent function. The code pattern in this class has passed hundreds of regression test cases that provide you with a reliable foundation so you can save time and concentrate on your specific project code.
- A powerful standard response processor out of the box that dramatically reduces the code required in your main fluent function and the code required to manually parse the raw responses.

A standard fluent function base class

This base class provides the most commonly used code for a typical integration fluent function below:

- All initialized variables like logger, svcDef, txn, inputParams, user, appDoc, applicantRole, and fluent function results
- Setup service module
- Standard input parameters injected (recordResponseInTxnProperties, recordResponseInTxnXML)
- Standard milestone events on start and complete
- Standard exception handling and update of server VO and client VO when required

The code pattern in this class has passed hundreds of regression test cases that provide you with a reliable foundation so you can save time and concentrate on your specific project code.

A powerful standard response processor out of the box

Previously, you needed to create a response processor subclass and write code to process the raw response field by field and generate the Server VO and Client VO manually.

From 1.4.0, we started to provide simple annotations, that lets the response processor inject certain values from the raw response into the Server VO automatically. We go much deeper in 1.5.0, with a lot more auto response processing capabilities, as shown below, to finally provide a fully working response procesor out of the box, that saves you time parsing the response manually.

- Client VO auto-processing by annotation
- Properties auto-matching between Client VO and Server VO
- Auto data value mapping
- Turn response directly into Map object in VO and filtering on the result
- JSON string as a single property
- Clear VO properties when Errors Happen
- Default error handling in response processor

With this release, the framework dramatically reduces the code required in your main fluent function and the code required to manually parse the raw responses. We strongly encourage everyone to upgrade and start using the framework to build your next integration.

Exchange Framework (TIF) v1.6.1 release

We are pleased to add the built-in configuration service support in this release so that your fluent function using TIF will automatically be compatible with the configuration service with the following features without any extra code:

- Triggers the specified configuration service on demand.
- Uses the config data from the configuration service if available, otherwise fall back to the original service connection and service parameters from the service definition and make those up-to-date config data available to your fluent function.

Other updates to the Exchange Framework in this release include:

- Provide the capability to disable the Journey Analytics milestones in the Fluent function base class.
- Add capitalized Key annotation to match the capitalized tag name convention in Txn XML that generated by Server VO/Client VO (default is true)
- Add get and create form/organisation property in Utils class

- Add request object access in the base fluent function
- Support mocking a Groovy service or fluent function in MockUtils
- Support mocking a GET request in MockUtils
- Fix a minor issue in Utils.exceptionToString() method

References

For more information please refer to:

- TIF in Gitlab (user credentials required)
- Exchange Framework Developer Guide

The Exchange Domain Shared Code Library (for Fluent SDK 5.x)

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

A shared code library is published by the Exchange team that encapsulates functionality relating to commonly encountered tasks, including:

- Providing response data from Dynamic Data Services and consistency on communicating errors back to the client.
- Dealing with addresses in short or long format and standardizing the API to address lookup services.
- Performing data transformations using the Groovy Template Engines.
- Generating IDs using a range of casing strategies.

These capabilities are provided as a set of Groovy classes that can be referenced in your Groovy services using the fileIncludes directive. See Services Overview for more information.

Shared Code Library

https://avoka-gitlab.avoka.com/exchange/exchange.domain

Accessing the Exchange Git Repository

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

The Exchange GIT repository group for Exchange partners is located here:

https://cs-gitlab.avoka.com/partners

You should have received a URL to your specific repository when your partner account was setup.

The Exchange Sandbox Environment

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

The Exchange team will provide a sandbox Journey Manager environment for partners developing Exchange Packages. This environment can be found here:

https://exchange.avoka-transact.com/manager

You would have received a login credential to this environment when your partner account was created.

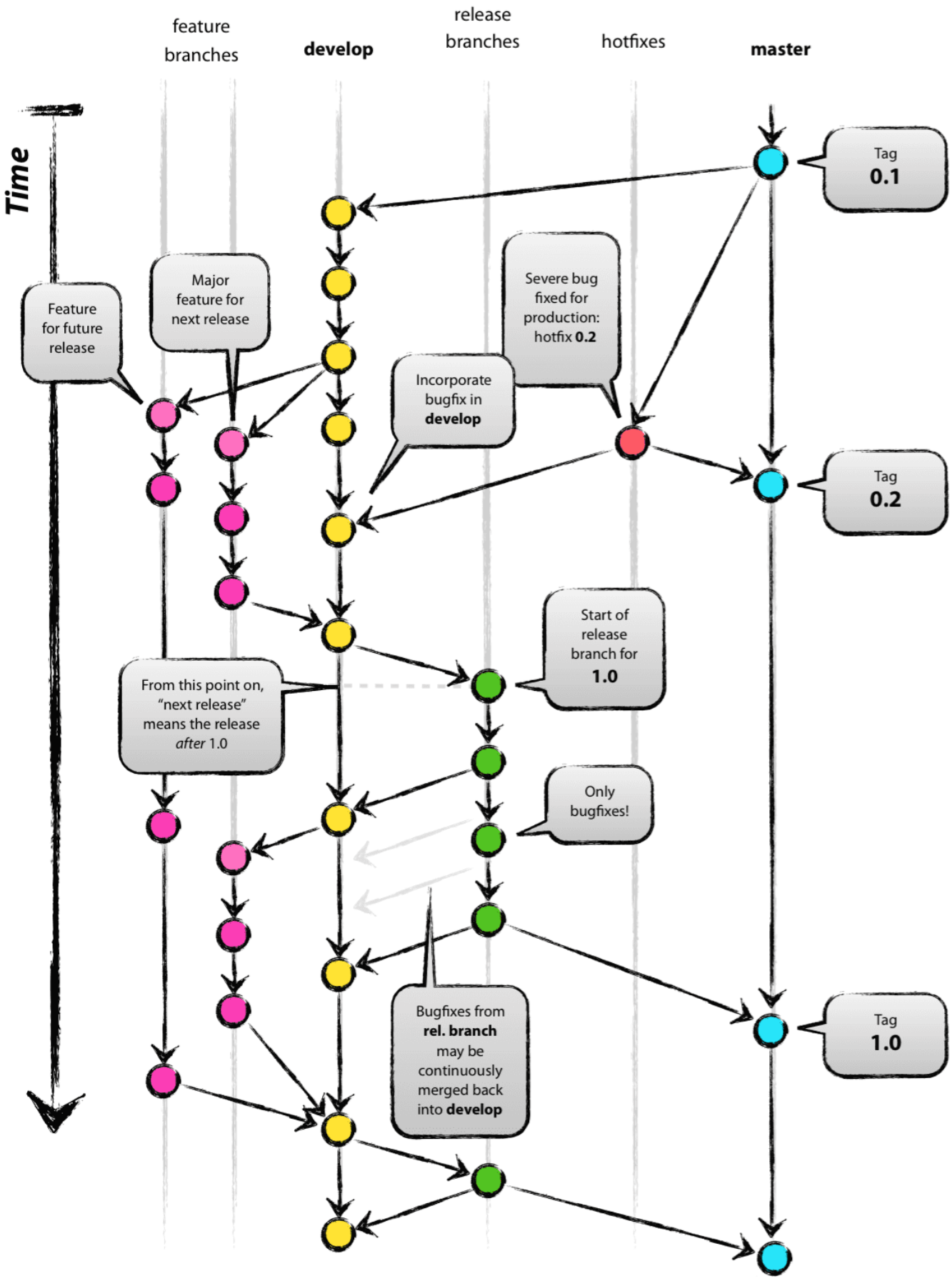
Using GitFlow in Exchange projects

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

GitFlow is a branching model for Git, created by Vincent Driessen. It has attracted a lot of attention because it is very well suited to collaboration and scaling the development team. We adopt this model for all Exchange projects source code management.

NOTE

Please note that it's not mandatory to use a GitFlow command to create/manage branches. It's completely valid to use normal git commands to create/merge/remove branches as long as it matches the principle of the GitFlow branching model below.


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Installing GitFlow

- Windows: Git (2.10+) includes GitFlow by default
- OSX: brew install git-flow
- IntelliJ: Git Flow Integration plugin

Migrating existing projects

In some of the earlier Exchange projects, they only have the master branches. We need to gradually migrate these projects to the new GitFlow model.

When

When we need to modify the project. Could be either a bug fix or feature enhancement.

How

The main difference between the old model and new model is introducing "develop" branch. So in order to migrate to the new model, we need to create a new develop branch based on current master branch.

GitFlow command

```
git flow init
```

NOTE

Make sure your local repositories are synced with remote repositories. Otherwise, it may fail when you run the above command. You can do this by executing "git fetch --prune".

NOTE

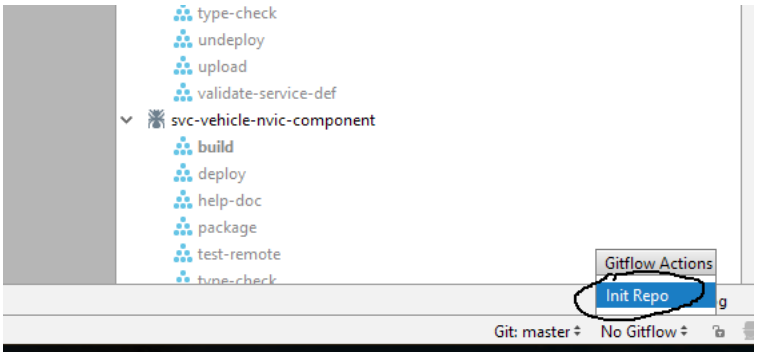
Also, make sure you delete any previous local branch you may have that already merged into master. You can remove local branch by "git branch -d branch_name".

Git command

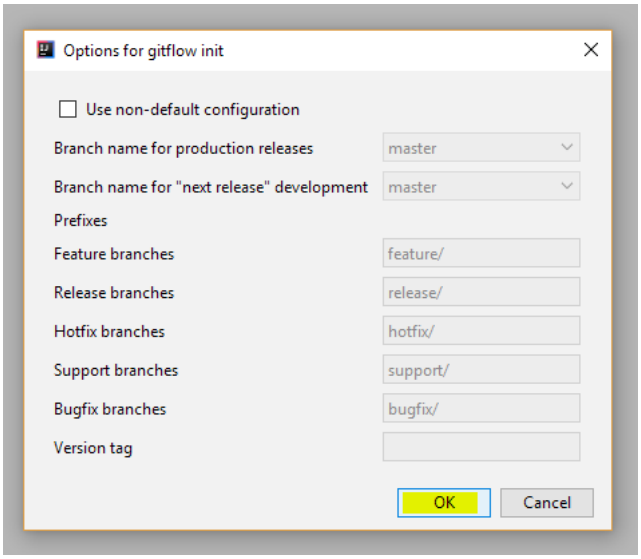
```
git checkout -b develop master
```

IntelliJ GitFlow plugin

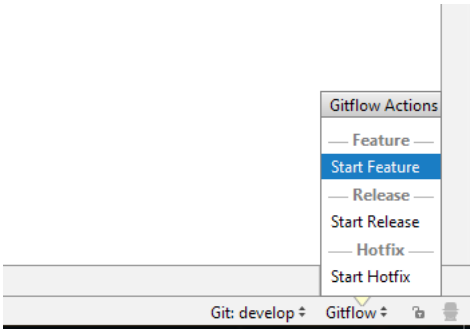
Make sure you are in the latest master branch and don't have any unstaged changes locally, and at bottom right corner, click on "No Gitflow" and select "Init Repo" in Gitflow Actions.



In the option pop-up dialog, just click OK to use the default configuration.



If it succeeds, it automatically creates a develop branch and switches to this branch. Also when you click on Gitflow, it will show options to Start Feature, Release, Hotfix as per below.



NOTE

GitFlow Init Repo will fail if you have unstaged changes.

Make changes to the project (Enhancement or bug fix)

You should always make changes based on develop branch. GitFlow recommends creating a feature branch based on develop branch and when it's finished, merge it back into develop branch and delete the feature branch. You can do this by using GitFlow command or pure git command. GitFlow command or plugin is recommended because it automatically creates the correct prefix branch name and handles merge and remove feature branch for you when finished.

NOTE

In Exchange projects, we tend to always release a newer version for both enhancements and bug fixes, hence we don't normally create a branch directly from master for bug fixes (hotfixes concept in GitFlow). Plus Exchange developers typically don't have the write permissions to the master branch, creating a hotfix branch may later on have trouble merging back to the master branch. So it's recommended to follow the same process for both bug fixes and feature enhancements as per below.

Step 1 Create a feature branch

Make sure you are on the latest develop branch.

GitFlow command

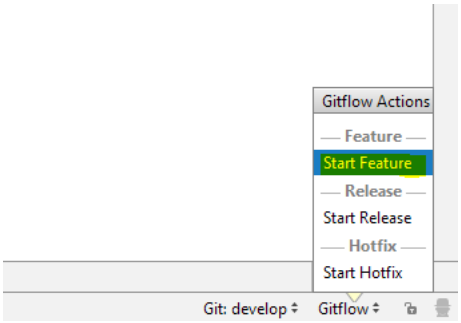
```
git flow feature start my-change
```

Git command

```
git checkout -b feature/my-change develop
```

IntelliJ GitFlow plugin

Click on Gitflow at the bottom-right corner, and select Start Feature like below.



Step 2 Make changes

Make changes to the feature branch you just created. If you are collaborating with team members, you might want to commit and push your newly created branch to remote so that other members can work on the same branch.

> NOTE
>
> It's recommended to use "git pull" to get the changes from other members regularly (daily) to avoid/resolve potential conflicts at early stage rather than merging large amounts of code towards the end.

Step 3 Finalize the feature branch

When the feature or bug fix is done and tested, check out develop, merge the feature branch back into develop branch and delete the feature branch.

NOTE

Make sure you've committed and pushed all your changes before performing the following command. Failure to do so may cause issues.

GitFlow command

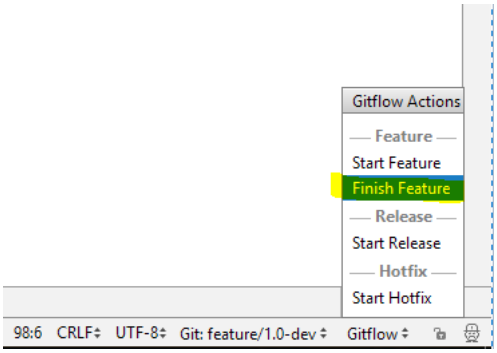
```
git flow feature finish my-change
```

Git command

```
git checkout develop
git merge --no-ff feature/my-change
git branch -d feature/my-change
```

IntelliJ GitFlow plugin

Click on "Gitflow" at the bottom-right corner, and select "Finish Feature" like below.



Releasing a version

When all the changes on develop branch have been tested and QA signed off, it's the time to create a release version based on develop branch.

Step 1 Create a release branch

We should use the TPac version MajorNumber.MinorNumber.buildNumber to be the release branch name. For example, to release TPac v1.2 build 004, use 1.2.004.

GitFlow command

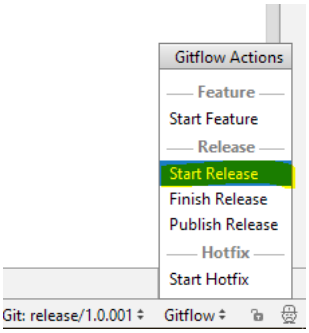
```
git flow release start 1.2.004
```

Git command

```
git checkout -b release/1.2.004 develop
```

IntelliJ GitFlow plugin

Click on Gitflow at the bottom-right corner, and select Start Release like below.



Step 2 Create distribute pack

Create the distribution package if you haven't already done so.

> NOTE
>
> Please note you should avoid code changes at this stage except a documentation update. Code review, functional testing and bug fixes should be finished before creating a release.

Step 3 Finalize the release branch

To finalize the release branch, it needs to merge the release branch back into both develop and master branch. Since sometimes developers don't have write permissions to the master branch, it can create problems when finalizing the release branch using a GitFlow command or GitFlow plugin. The recommended way is to either use a user with write permissions or use the following command to merge release into develop and manually create a merge request though Gitlab to master.

GitFlow command

```
git flow release finish 1.2.004
```

Git command

1. Execute the following command to merge release to develop branch.

```
git checkout develop
git merge --no-ff release/1.2.004

git push
```

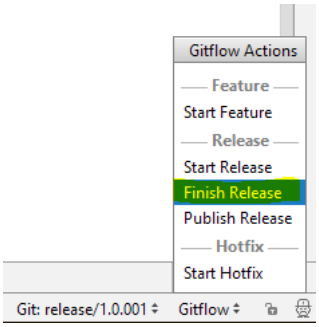
2. Create a merge request in gitlab from develop branch to master branch.
3. Execute the following command to tag the master and delete the release branch.

```
git checkout master
git tag -a 1.2.004
git branch -d release/1.2.004
```

IntelliJ GitFlow plugin

> NOTE
>
> Make sure you have the write permission to master branch before using this function.

Click on Gitflow at the bottom-right corner, and select Finish Release like below.



After clicking Finish Release, you still need to perform the following command to push local to remote and tag the master.

```
git checkout develop
git push


git checkout master
git push
git push --tags
```

NOTE

GitFlow will create tags automatically in the local master, or you can use git push --tags to push tags to remote if you don't have any other local tags. Otherwise please use git push origin <tagname> to only push the released version tag.

Logging third party service calls

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

Background

Currently, Journey Manager provides the following API to log third party service calls:

```
new TxnUpdater(txn).addServiceCallLog(serviceName, moreInfo, serviceEndpoint).update();
```

The problem with the existing data is that:

- We don't know if a certain service call log is from an Exchange package or not.
- Furthermore, we don't know if a certain service call log is from which Exchange package.

Logging convention

In order to solve this and be able to know which Exchange package a service log record belongs to, with minimum backend and API changes, we use the moreInfo parameter to provide the Exchange package information in the following format:

EXCHANGE|exchange-project-code|moreInfo

For example, the Mitek TIDEN project has the following code to record service log at the moment:

```
// record service usage

new TxnUpdater(txn).addServiceCallLog('MitekTidenId Prefill',"EvidenceId: $evidenceId",
endPoint).update()
```

It should be changed to the following to support the new convention:

```
// record service usage

new TxnUpdater(txn).addServiceCallLog('MitekTidenId Prefill',"EXCHANGE|mitek-tiden|Evid-
enceId: $evidenceId", endPoint).update()
```

Please note that the Exchange project code should match the "project.code" defined in the build.-properties. For example, in the Mitek TIDEN project, build.properties has the following line:

project.code=mitek-tiden

Hence, mitek-tiden should be used as the Exchange project code in the moreInfo string.

Migration Policy

Any new project needs to follows this convention.

For all existing projects, we need to follow this convention whenever we upgrade the package.

More on logging third party service calls

Usage

When utilizing third party services in groovy service definitions it is important to record each successful usage. Usage statistics are very useful and sometimes required to be tracked for reconciliation and billing purposes.

The TxnUpdater provides a function to record these events as demonstrated below.

The service call should be logged only if:

1. A successful response has been received from the service - this can be verified with the convenience function isSuccess() on the HttpResponse object.
2. No data errors were reported in the response from the service. Typically a 400 (BAD REQUEST) response is received when there is an issue with the data provided in the request, however some 3rd integration APIs will incorrectly return a 200 response type but still report data errors in the body of the response (e.g. 'Invalid SSN') - these will need identified and the service log skipped in this scenario as data errors do not typically represent a billable event.

Sample code

```
import com.avoka.tm.svc.TxnUpdater
import com.avoka.tm.http.GetRequest
import com.avoka.tm.http.HttpResponse
...
try {
       response = new GetRequest(serviceEndpoint).execute()
} catch(Exception e){
       // Log and return
       new EventLogger().setTxn(txn).setMessage(e.getMessage()).logError()
       return result.systemError()
}

if (response?.isDataError() || checkBodyError(response)) {
       return result.dataError(getDataErrorMsg(response))
}

if (response?.isSuccess()) {
       // Add the 3rd Party Service Call Log entry
       new TxnUpdater(txn).addServiceCallLog(serviceName, 'More Info (optional)', ser-
viceEndpoint).update()

       // Now process the response
       ...
}
```

Inputs

The inputs to this addServiceLog function are:

- svcName - The name of the third party service - this is not the SOAP Action name but rather relevant name that clearly identifies the vendor and service being called (e.g. FIS - QualiFile)
- info - Any additional details you may want to provide to assist in auditing, this may include key attributes included in the call (optional)
- url - Where the third party provides multiple endpoint URLs for different environments it is important to record this URL as calls to the test environment would be excluded from billing transactions. See also Managing multiple service endpoints and credentials for external service calls.

WARNING

Remember you must call the update() funtion on the TxnUpdater to commit the log record.

Managing multiple service endpoints and credentials for external service calls

Exchange Pre-configured Maestro services. | Platform Developer | All versions This feature is related to v5.1 and higher.

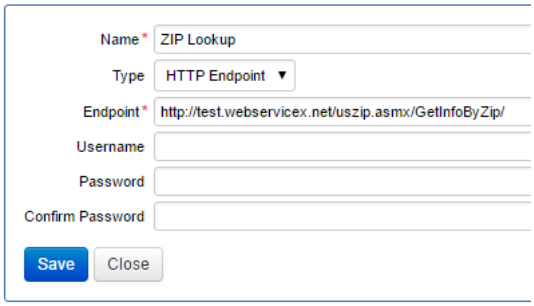
As we create services in Journey Manager that integrate with external sources via Web Service or REST we typically find that the external service endpoint and credentials change between Journey Manager environments (dev, test, prod). Or alternatively, you may have created a mock service endpoint for 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.

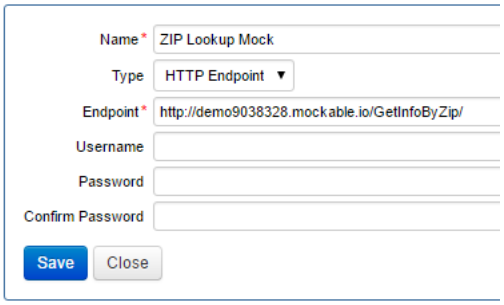
Say we have a requirement to build a Dynamic Data service for use by forms that takes a US Zip Code and returns the City, State, Area Code and Time Zone. This service will call out to an external API to retrieve this information. Assume the external service provider has exposed a test service for use 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.



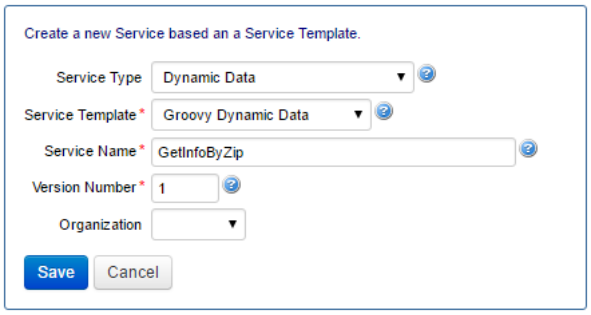
2. As we are developing this service we want to access a mock endpoint to control the

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.

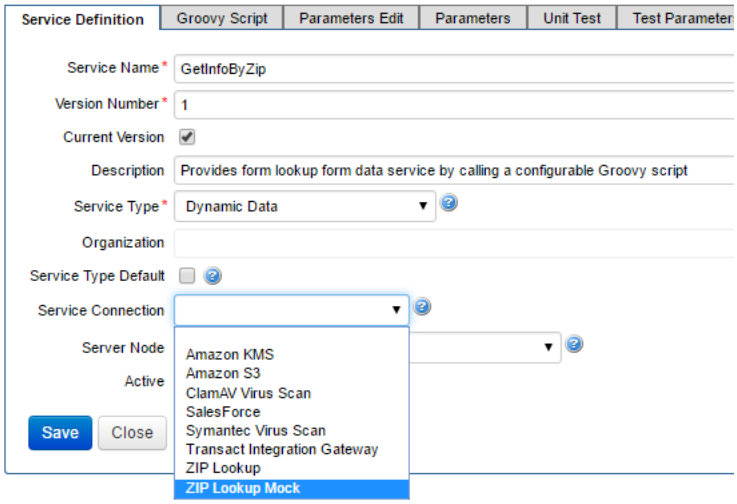


Using Service Connections in Groovy Services

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.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.