

Assembly Editor Guide

April 15, 2019



PROPRIETARY NOTICE

Copyright © 2019 APIANT, Inc. All Rights Reserved.

The information herein is the property of APIANT, Inc., and any misuse or abuse will result in economic loss. DO NOT COPY UNLESS YOU HAVE BEEN GIVEN SPECIFIC WRITTEN AUTHORIZATION FROM APIANT, INC.

DISCLAIMER

THE INFORMATION IN THIS DOCUMENT WILL BE SUBJECT TO PERIODIC CHANGE AND UPDATING. PLEASE CONFIRM THAT YOU HAVE THE MOST CURRENT DOCUMENTATION. THERE ARE NO WARRANTIES OF ANY KIND, EXPRESS OR IMPLIED, PROVIDED IN THIS DOCUMENTATION, OTHER THAN THOSE EXPRESSLY AGREED UPON IN THE APPLICABLE APIANT, INC. CONTRACT, ERRORS AND OMISSIONS EXCEPTED.

For additional information, please contact:

APIANT, Inc.
196 West Ashland Street
Doylestown, PA 18901

Email: support@apiant.com

Table of Contents

Introduction	6
APIANT overview	7
What is APIANT?	7
Who can use APIANT?	8
Chapter 1: Assembly Editor Overview	13
Assembly Editor Overview	14
File Menu	15
Opening Recently Saved Assemblies	16
Saving Assemblies	17
Edit Menu	19
Tidying Diagrams	19
More Menu	20
Account Menu	21
Account Management	21
Keyvault Management	22
Switch Accounts	22
Developer Menu	24
Open Admin Console	24
Open Assembly Editor Guide	24
Open appRPC Javadoc	24
Open VTD-XML Parser Javadoc	25
Open Trace Log	25
Purge Trace Log	25
Open Webhooks Log	25
Purge Webhooks Log	25
Execute Assembly on Server	25
Request a Review of this Assembly	26
Settings	27
Snap drag	27
Show service account fields	28
Reuse module data when possible	28
Autoload data	29
Debug mode	29
Report execution errors	30

Debug module timings	30
Debug nested modules	30
Debug module input/output data	30
Verbose module debug output	30
Versions	31
Comparing Assembly Versions	31
Catalog	33
Sharing Catalog Content	36
Tagging Catalog Content	37
Information Area	38
Quick Picks	41
Chapter 2: Working with Modules	42
Modules	43
Chapter 3: Working with Assemblies	47
Module Wiring	48
Building Assemblies	51
Editing Assemblies	57
Cloning modules	59
Documenting Assemblies	60
Chapter 4: Subassemblies	62
Overview	63
Dropdown Parameter Subassemblies	66
Subassembly Input Parameters	67
Chapter 5: API Integrations	70
Overview	71
Service Accounts	72
App Assemblies	75
No Credentials Needed	76
User-Entered Credentials	77
OAuth Integrations	80
OAuth v2.0 Token Refresh	83
OAuth Access Token Expiration	83
Trigger and Action Commonality	84
API Credentials	84
Error Handling	85
Automatic Error Retries	86
Configuration Settings	87
Trigger Assemblies	89

Webhook Triggers	90
Manually Configured Webhooks	91
API-Registered Webhooks	92
Dangling Webhooks	94
Webhooks with one event type	96
Webhooks with multiple event types per account	96
Webhooks with multiple events and multiple accounts	97
Protocol Thread Triggers	98
Unary Protocol Threads	98
Freeform Protocol Threads	98
Webhook Listener Threads	108
Starting/Stopping Unary Protocol Threads	112
Per-Trigger Protocol Threads	113
Polling Triggers	115
Data Row Identifier Storage	121
Date/Time Triggers	122
Gated Triggers	123
Export Triggers	124
Action Assemblies	126
Find Actions	128
Two-Way Sync	129
Two-Way Sync Triggers	130
Two-Way Sync Actions	131
Chapter 6: Other Assembly Types	133
Web Services	134
Managing Web Services	138
RSS Feeds	139
Batch Jobs	140
Managing Batch Jobs	141
Chapter 7: Other Functionality	142
Simple DB	143
Importing and Exporting	145
Exporting	145
Importing	147
Publishing	149

Introduction

The APIANT Assembly Editor Guide is for semi-technical and technical people who want to learn the basics of using the Assembly Editor to integrate API's and build solutions.

APIANT overview

What is APIANT?

APIANT is a system for leveraging API's to build automated workflows and visual widgets. APIANT provides advanced technology for building solutions that access, process, and output data easily and flexibly.

APIANT is browser-based. All solution development occurs within a web browser.

APIANT can build automations. APIANT can build automations that perform data processing among multiple API's. Automations consist of one or more triggers and one or more actions plus optional conditional logic. If the trigger criteria are met, the actions are invoked. Conditional logic can be used to execute certain actions. Conditional logic can be nested.

API integrations can require no coding. Visual wiring diagrams, called Assemblies, allow data processing modules to be connected and configured to send/receive data from API's and perform data processing logic. Nested assemblies can be created, called subassemblies. Subassemblies allow modularized, re-usable data processing components to be built. Extension modules allow inline coding when necessary, using JavaScript, Java, or PHP.

Solutions can run in the browser or on the server. APIANT can not only build automations, but also web services (REST-style API endpoints external systems can invoke), batch processes, and even visual browser widgets. Browser widgets can access server-side assemblies, allowing solutions to distribute their processing as needed.

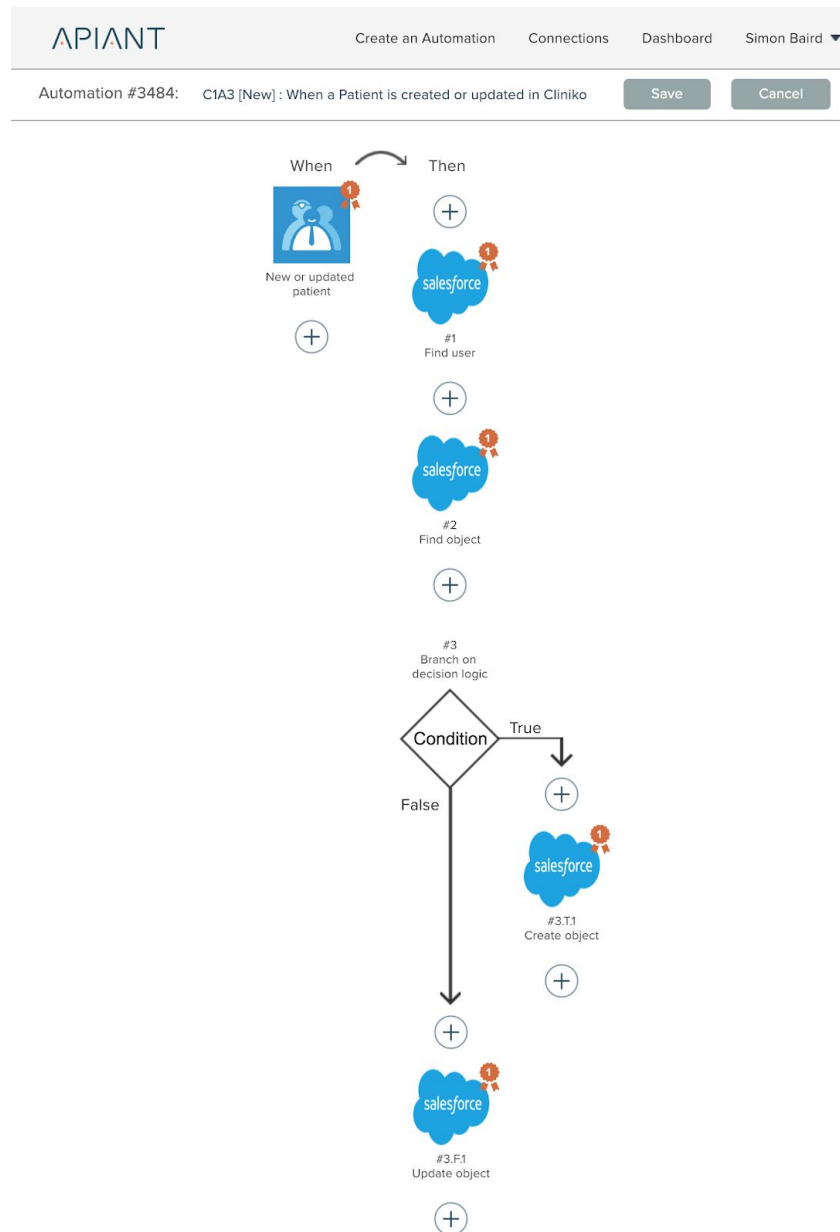
APIANT is fully extensible. Software developers can extend the system either with inline JavaScript/Java/PHP code or by writing custom software modules using the built-in browser-based Integrated Development Environment (IDE). A full-featured debugger aids module development.

Who can use APIANT?

APIANT is designed to be used by people of varying technical skill levels:

- Non-technical people can build automations with the Automation Editor that perform actions when trigger criteria are met.
- Semi-technical analysts with knowledge of XML can construct assemblies that process data and perform logic without coding.
- Business-level developers with some knowledge of JavaScript can create more sophisticated solutions, by writing logic and connecting components together to form larger solutions.
- Software engineers can extend the system by building custom software modules using JavaScript/Java and the provided Module API.

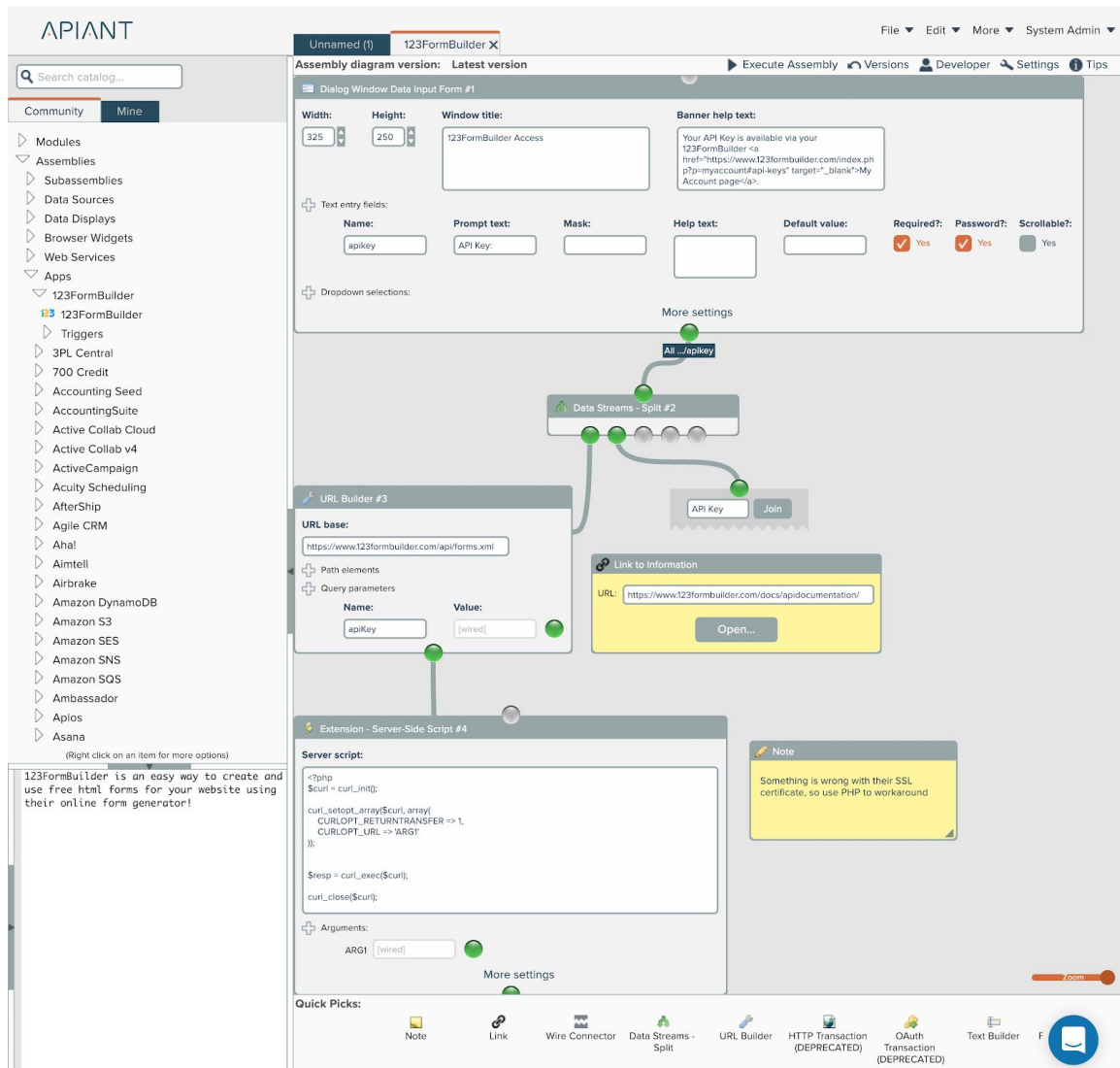
APIANT's Automation Editor can be used by non-technical people to build automated workflows.



Automations can consist of one or more trigger criteria coupled with one or more actions to be performed. Actions can be branched using conditional logic. Nested conditional logic branches can be built.

Apps, triggers, and actions are all integrated via Assembly Editor diagrams, making the Automation Editor fully extensible.

All automations and widgets in the system are composed of one or more assemblies built with the Assembly Editor.

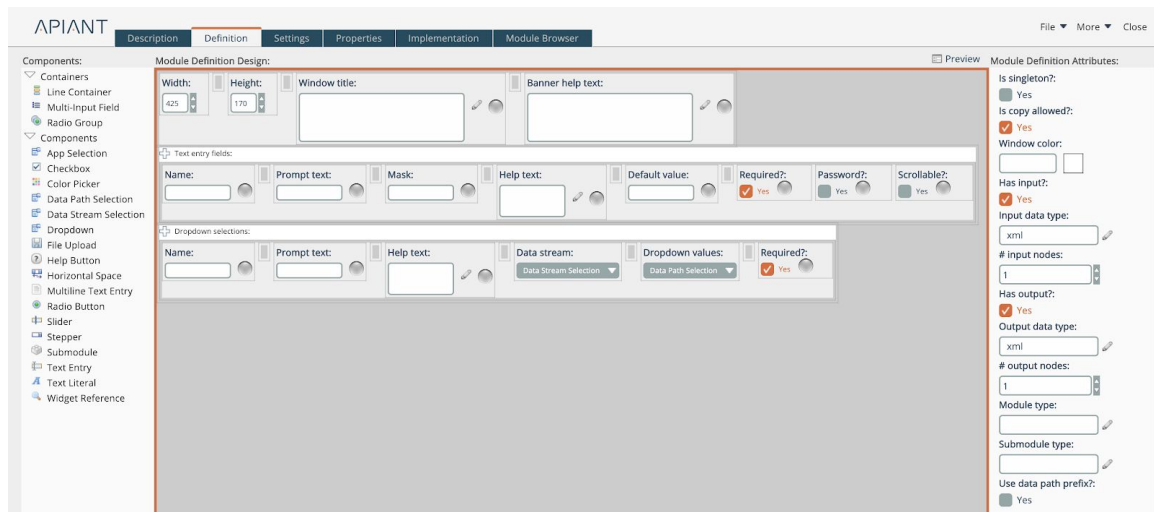


An assembly is a collection of software modules configured to perform processing. Software modules perform data input, processing, and output.

Nested assemblies can be created, called subassemblies. Subassemblies help to manage the complexity of assembly diagrams by forming modular, reusable components.

Assemblies may run either in the browser or on the server. An assembly running in the browser can invoke one or more assemblies on the server, which in turn can invoke other server-side assemblies.

Assemblies are composed of one or more software modules, created with the Module IDE.



The Module IDE provides all the facilities needed to build and troubleshoot software modules.

The module user interfaces that appear in assembly diagrams are built with the drag-drop WYSIWYG module definition designer shown above.

Modules can have properties that control their runtime behavior. A drag-drop WYSIWYG property designer is used to design how module properties appear.

A browser-based code editor is used to develop the module implementation using JavaScript/Java. The UI for visual modules can be designed with a WYSIWYG designer.

Class libraries can be developed, to create reusable modules that can be shared across module implementations. The system provides many class libraries used to construct the system itself.

Server scripts can be developed from the IDE, using languages installed on the server like JSP or PHP. Server scripts help extend the processing of modules, making use of existing libraries.

Module resources like graphic art and audio clips can be uploaded to the server from the IDE.

The Module IDE also provides a full-featured debugger to help troubleshoot modules. The debugger can inspect variables and object instances, monitor class attributes and methods, and evaluate entered expressions.

APIANT includes a comprehensive Admin Console for system administrators. The Admin Console provides system administrators extensive tools and functionality to help operate the system.

APIANT Admin Console

- Activity
- Automations
- Batch Jobs
- Web Services
- Protocol Threads
- Native Work Queue
- Memory Usage
- Tenants
- User Accounts
- User Roles
- App Catalog
- Catalog Categories
- System Compile
- System Export
- System Keyvault
- System Settings
- System Upgrade

Recent User Activity:

System Admin	Nov 05 22:46:35 EST
Anonymous	Nov 02 20:36:28 EDT
System Setup	Jul 31 20:59:10 EDT
Cloud Setup	Dec 17 11:13:30 EST

Refresh Activity List

System Announcement:

The system will be going offline 15 minutes from now. Save any work and login again about 30 minutes from now.

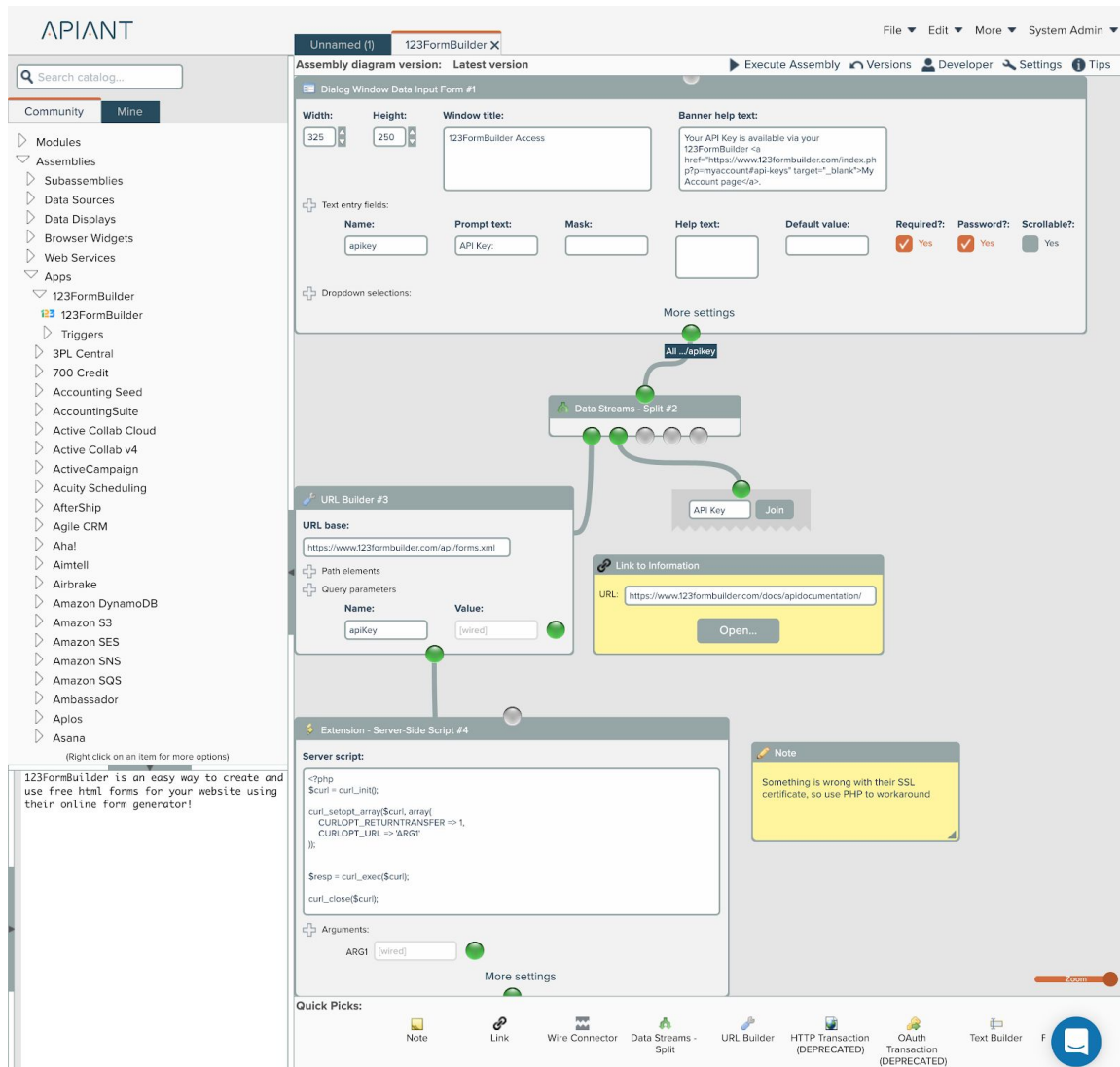
Activate Announcement

Compose E-mail to Broadcast...

Chapter 1: Assembly Editor Overview

Assembly Editor Overview

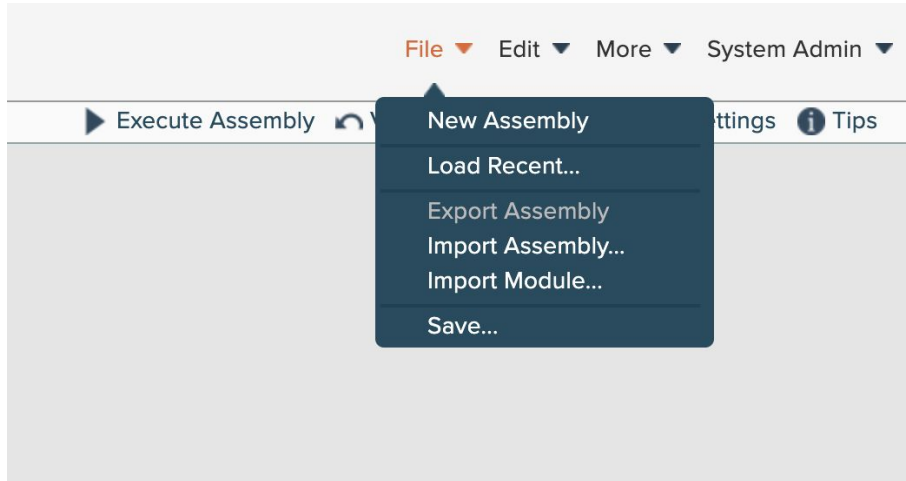
APIANT's Assembly Editor is a browser-based visual software development tool for building software solutions. Prefabricated or custom-built software modules are wired together and configured to perform functionality.



The system contains over 200 baseline modules, including extension modules that allow inline Java, PHP, and JavaScript code to be used. Modules are built with the integrated Module IDE, documented in a separate guide. New module development is typically no longer necessary.

File Menu

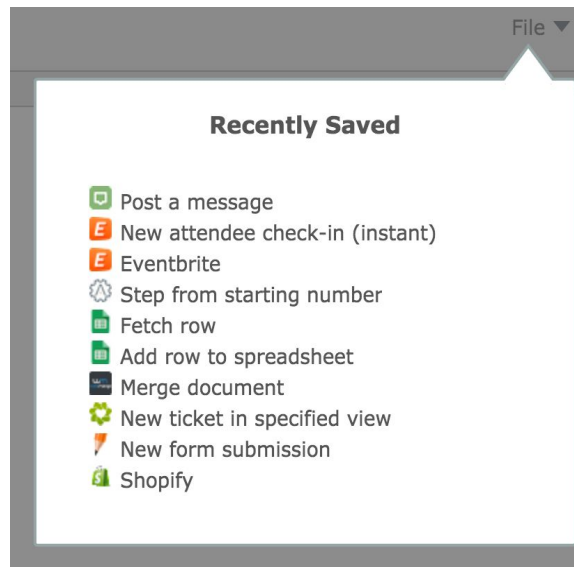
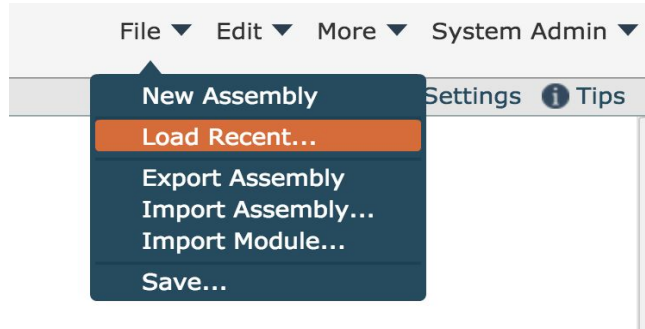
The **File** menu at the top right of the editor provides functionality for managing your assembly diagrams:



Note: The export and import menu options only appear for accounts having the "Export" and "Import" permissions.

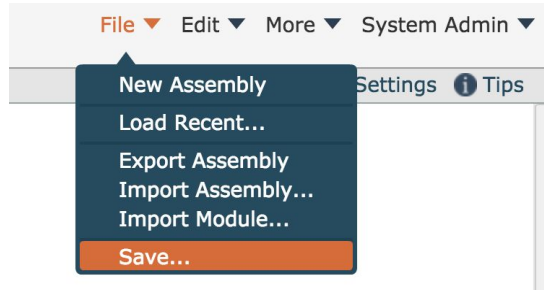
Opening Recently Saved Assemblies

You can quickly access your most recently saved assemblies via the **Load Recent** menu option:

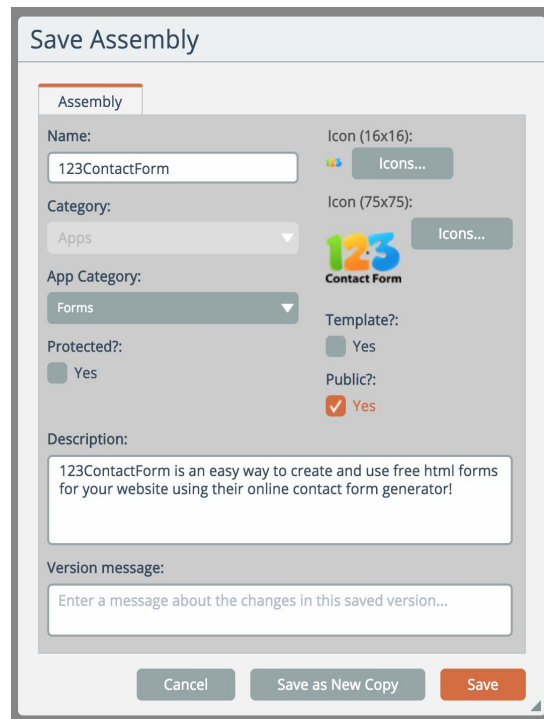


Saving Assemblies

Assemblies can be saved via the **Save** menu:



This opens the Save Assembly dialog:



The contents of the Save Assembly dialog will vary depending upon the type of assembly being saved and your account permissions.

In general the Save Assembly functionality determines the assembly's catalog entry.

If you don't own the assembly, you can only save a copy.

Assemblies can be saved as private or public. Private assemblies are not accessible by others and appear only in your "Mine" tab within your catalogs. Public assemblies can be used by anyone in the system.



Note: When saving a copy of an assembly that is public, it is initially saved as private.



Tip: Assemblies can also be right-clicked in the catalog to make them private or public.

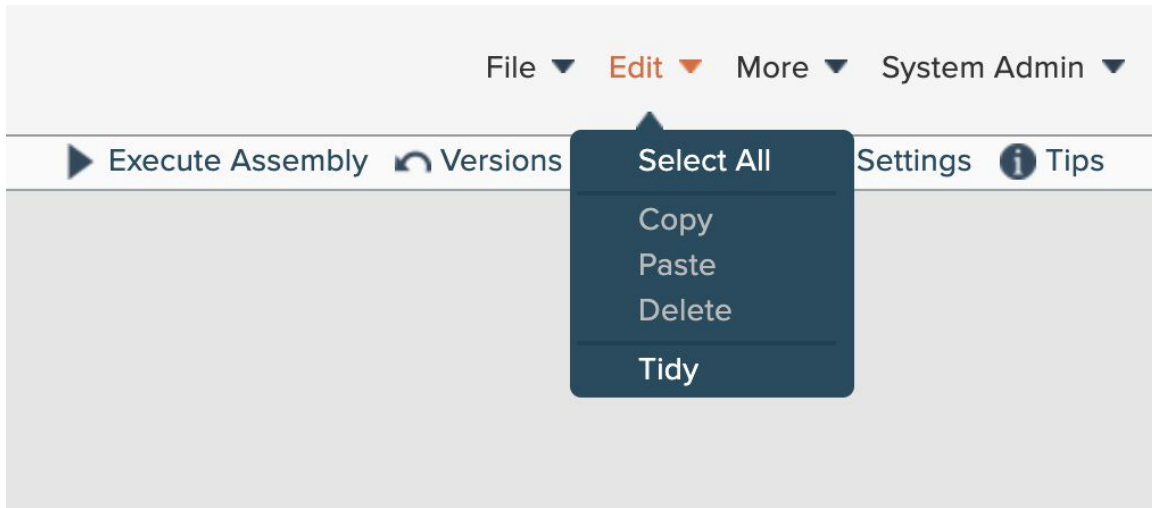
Once saved, any changes made to the assembly's catalog description will be reflected in the Assembly Editor's catalog. App, action, and trigger assemblies also get updated in the Automation Editor catalogs.



Note: After saving an app, action, or trigger assembly, the Automation Editor catalogs are not automatically updated if the Automation Editor is open in another browser tab. Perform searches in them to refresh their contents.

Edit Menu

The **Edit** menu at the top right of the editor provides functionality for copy-paste-delete of modules:



Note: It is usually easier to use the canvas-level right-click menu for edit options. See Chapter 2 section "Editing Assemblies".

Tidying Diagrams

The **Tidy** menu option automatically repositions elements in the diagram so that none overlap.

More Menu

The **More** menu at the top right of the editor provides functionality for the catalog and for managing Batch Jobs and Web Services:



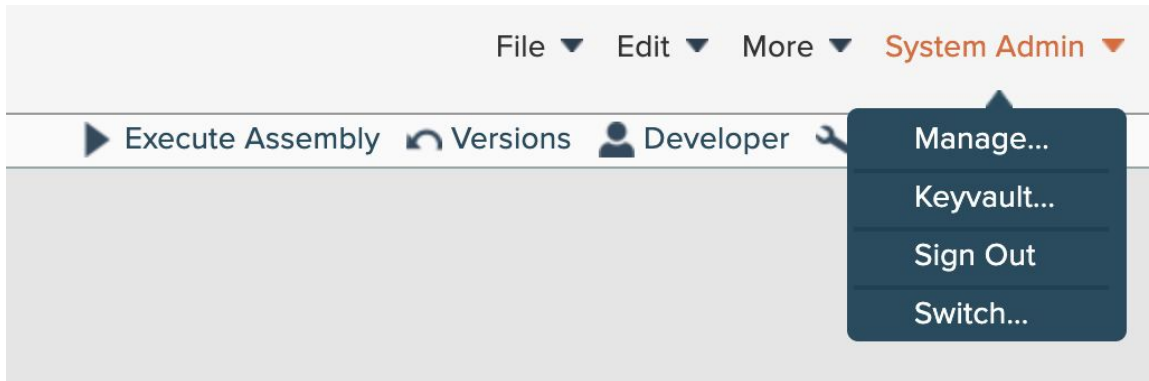
The **Close Catalog** option can be used to close and re-open the catalog, to help provide more editor space in the browser window. The menu option toggles depending on the state of the catalog.

The **Refresh Catalog** option reloads the catalog's contents.

See Chapter 6 for information about managing Batch Jobs and Web Services.

Account Menu

The **Account** menu at the top right of the editor provides functionality for the catalog and for managing Batch Jobs and Web Services:



Account Management

You can manage your account from the **Manage** menu option. The Account Management dialog will appear:

Manage Account

First Name: Last Name:

E-mail address:

Timezone: (GMT-05:00) Eastern Time (US & Canada) ▼

Change password...

Cancel
Update

You can edit your account details and change your password as needed.

Keyvault Management

The **Keyvault** menu option accesses your personal Keyvault. Values stored in the keyvault are accessible from Utility - Keyvault Value modules within assembly diagrams.

See Chapter 5 section "OAuth Integrations" for more about the usage of the keyvault.



Note: Typically your personal keyvault is only used for development and testing within the assembly editor. Finished integrations typically access the system admin's keyvault, accessible in the Admin Console.

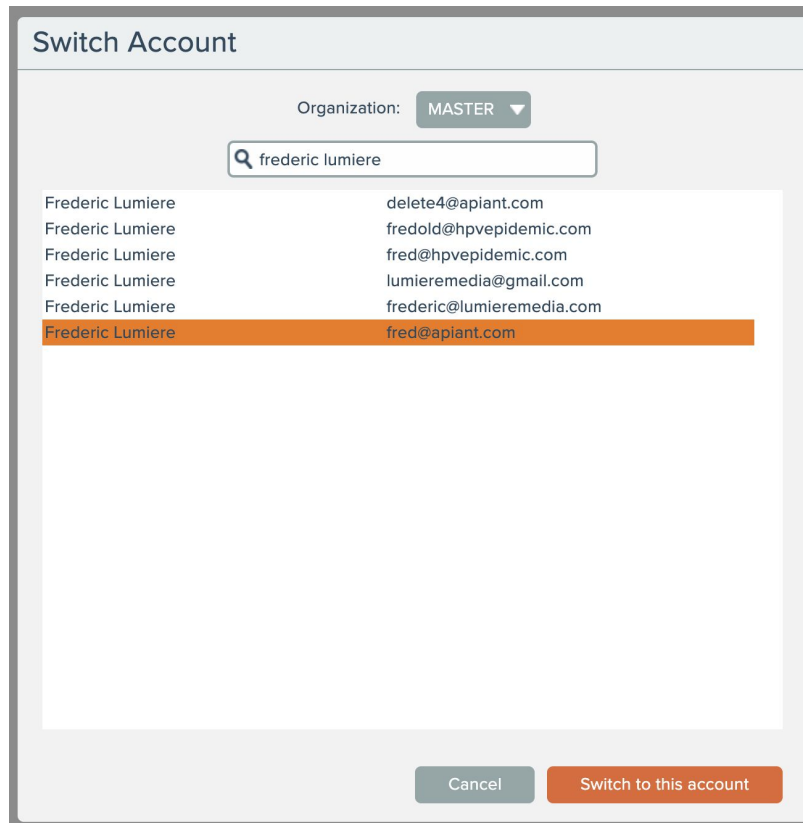
Switch Accounts

The **Switch** menu option is support-level functionality that allows you to impersonate other accounts.



Note: This menu option only appears for accounts having the "Switch Account" permission.

An account selection dialog will appear:



After an account is chosen, the current session will impersonate the selected account.

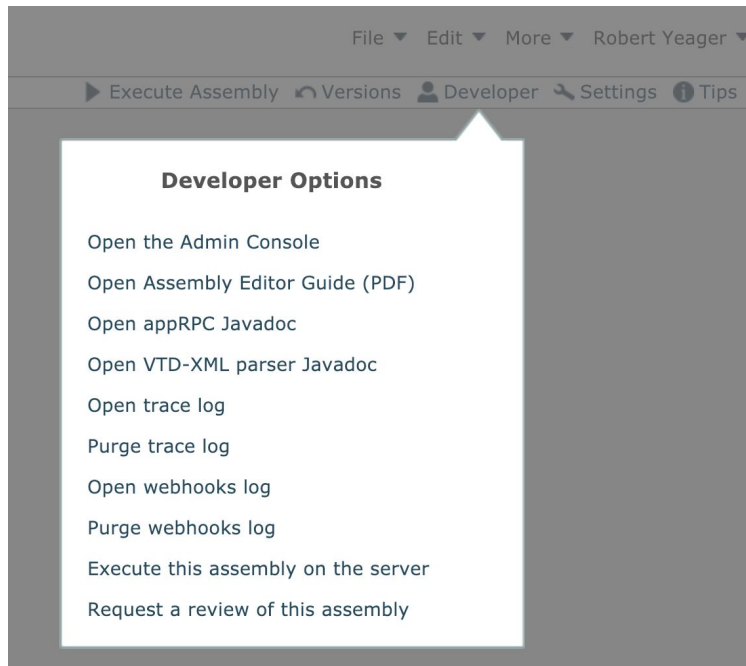


Note: Only switch accounts when no assemblies are being edited!

Assemblies access account information for the account that was active at the time the assembly was opened. So to test an assembly using a certain account, you must first switch to the desired account and then open the assembly.

Developer Menu

The developer menu at the top right of the editor appears for user accounts having the “Assembly Developer” permission:



Mouse over each setting’s hyperlink for help information.

Open Admin Console

Opens the Admin Console in a new browser window.

Open Assembly Editor Guide

Opens this guide in a new browser window.

Open appRPC Javadoc

Opens a new browser window with documentation for the appRPC object used in inlined Java JSP code within Extension modules.

Open VTD-XML Parser Javadoc

Opens a new browser window with documentation for the VTD-XML parser used in inlined Java JSP code within Extension modules.

Open Trace Log

Opens a new browser window with the output from `System.out.println()` when executed in inlined Java JSP code within Extension modules. The trace log is private to the developer's account.

Purge Trace Log

Purges the contents of the trace log. The trace log is private to the developer's account.

Open Webhooks Log

Opens a new browser window with the content of the system's webhooks log. The webhooks log contains raw data received by all incoming webhooks into the system.

Purge Webhooks Log

Purges the contents of the webhooks log. The webhooks log is global to the entire system.

Execute Assembly on Server

Executes the currently opened assembly on the server. Normally assemblies are executed in the browser as they are being developed in the Assembly Editor. Executing assemblies on the server allows them to be troubleshooted in the same manner as they are executed by automations.

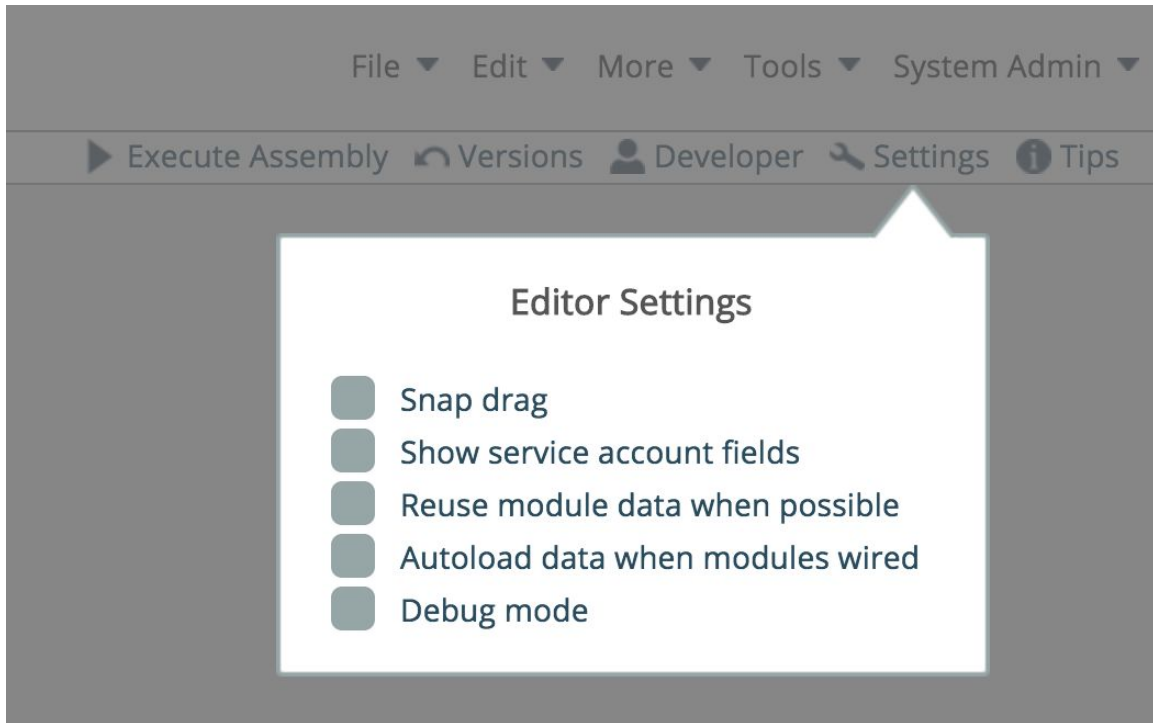
Request a Review of this Assembly

This link only appears if the server is a development server. Sends an email to all accounts in the system having the Assembly Reviewer permission, requesting that someone review the assembly. After clicking the link, you can enter a message for the reviewers to include in the email.

The assembly reviewer may place highlighted annotations in your assembly. The reviewer can send you an email message alerting you when their review is complete.

Settings

The Assembly Editor's settings are available at the top right of the editor:



Editor settings are saved in a browser cookie. If you access the editor in another browser, the settings are not restored with your session.

Mouse over each setting's hyperlink for help information.

Snap drag

When checked, modules will align themselves to an invisible grid when dragged.

Show service account fields

This is an advanced setting not normally used.

A "service account" is synonymous with "app account". It is the name an end user gives an account when an app is connected in the Automation Editor. The first connected account is always named "Default".

By default automation triggers and actions will use the service account configured by end users when automations are constructed. Service account fields within modules are normally left blank to indicate that the system should use the service account configured in the automation. Checking this option will cause service account fields to appear in the Assembly Editor modules, which allows assemblies to be constructed that use multiple service accounts.

Reuse module data when possible

The Assembly Editor's engine executes modules in the editor either when the "Execute Assembly" link at the top right is clicked, or when modules are double-clicked in the editor. (The engine can also be configured to automatically execute if the "Autoload data" setting is checked, described next.)

When the engine executes, if modules have been previously executed the engine can be configured to reuse the previously loaded data by checking the **Reuse module data when possible** option. This setting provides a significant speed enhancement, since the engine can skip modules that are not "dirty". A module is considered dirty if it has not been executed by the engine, or if any control within the module's UI has either received focus or been modified. If the setting is unchecked, all modules within the assembly will be forcibly executed by the Assembly Editor's engine whenever the engine is run.

Autoload data

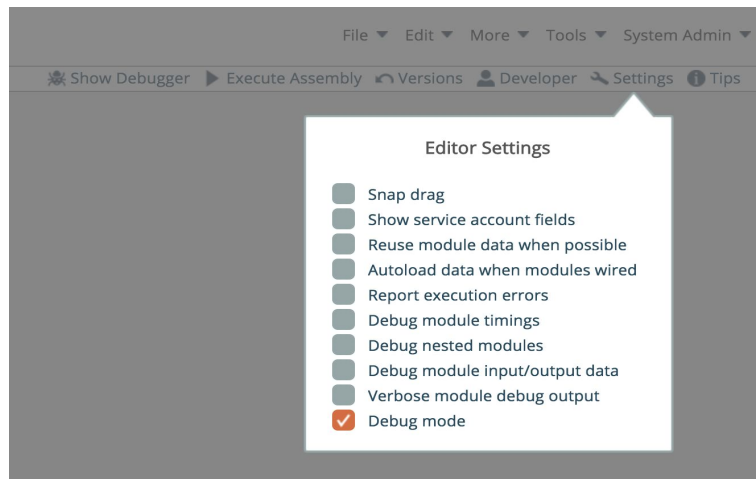
The process of building an assembly consists of wiring modules together and configuring them. Most modules in the system require input data from other modules in order to be configured. The Assembly Editor's engine must be executed to run the assembly and load any available data into the configured modules.

The **Autoload data when modules wired** option controls if the engine will be automatically run whenever modules are wired together. If the option is checked, each time a wire is connected to a module the engine will execute, automatically loading available data into the wired module. If the option is unchecked, the target module must be double-clicked or the Execute Assembly button must be clicked to run the engine to load available data into the module.

Debug mode

Selecting the debug mode option will cause the Assembly Editor to reload itself in debug mode. Debug mode provides access to a debug window where data can be inspected and module implementations can be troubleshooted.

When the editor loads in Debug Mode, additional options appear in the Settings:



Report execution errors

The Assembly Editor's engine traps any module execution errors that may occur as assemblies are executed. These are typically the result of errors in module implementation code created by module developers. As such, the reported error information is most useful for module developers to troubleshoot their developed modules. If the **Report execution errors** option is checked, the engine will display any execution errors in a popup window after the engine completes.

Debug module timings

The Assembly Editor's engine can display timing information for how long each module takes to execute.

Turning on this option may increase the execution time for assemblies in the editor.

Debug nested modules

The Assembly Editor's engine can optionally display nested modules in the debug window as assemblies are executed.

Turning on this option will increase the execution time for assemblies in the editor.

Debug module input/output data

The Assembly Editor's engine can display module input/output data in the debug window as assemblies are executed. This information can help module developers troubleshoot modules.

Turning on this option will **greatly** increase the execution time for assemblies in the editor. Unless modules are being debugged, it is recommended to leave this setting turned off.

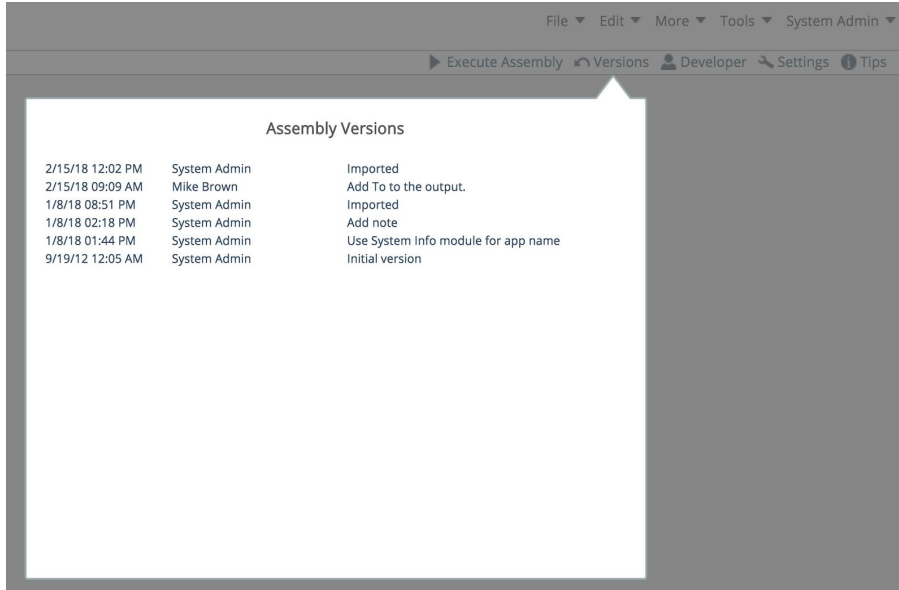
Verbose module debug output

Checking this option will cause modules to output all available debug information.

Turning on this option may **greatly** increase the execution time for assemblies in the editor.

Versions

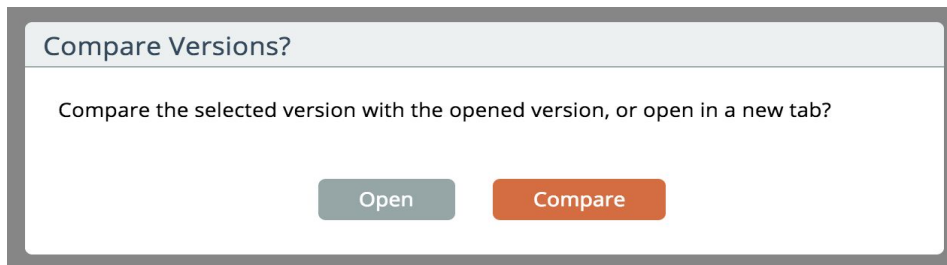
The versions link at the top right of the editor is enabled when an assembly is opened in the editor:



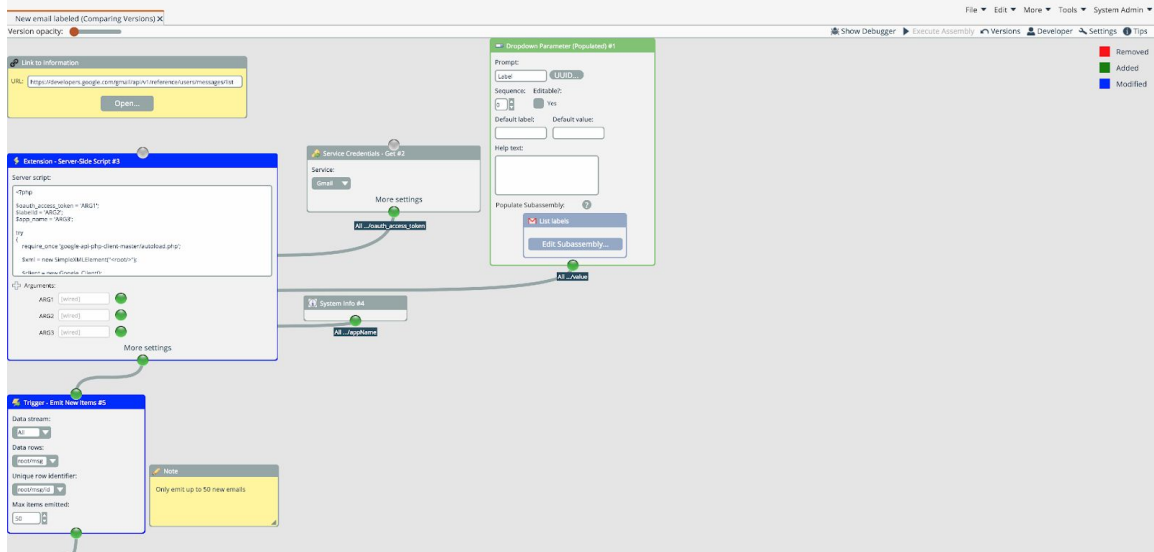
Each time an assembly is saved a new version is created. Clicking an old version from the list of versions will open that older version.

Comparing Assembly Versions

If a version from the assembly versions list is clicked and the diagram currently opened has no unsaved edits, a dialog will appear to either compare the selected version with the currently opened version, or to open the selected version in a new tab:

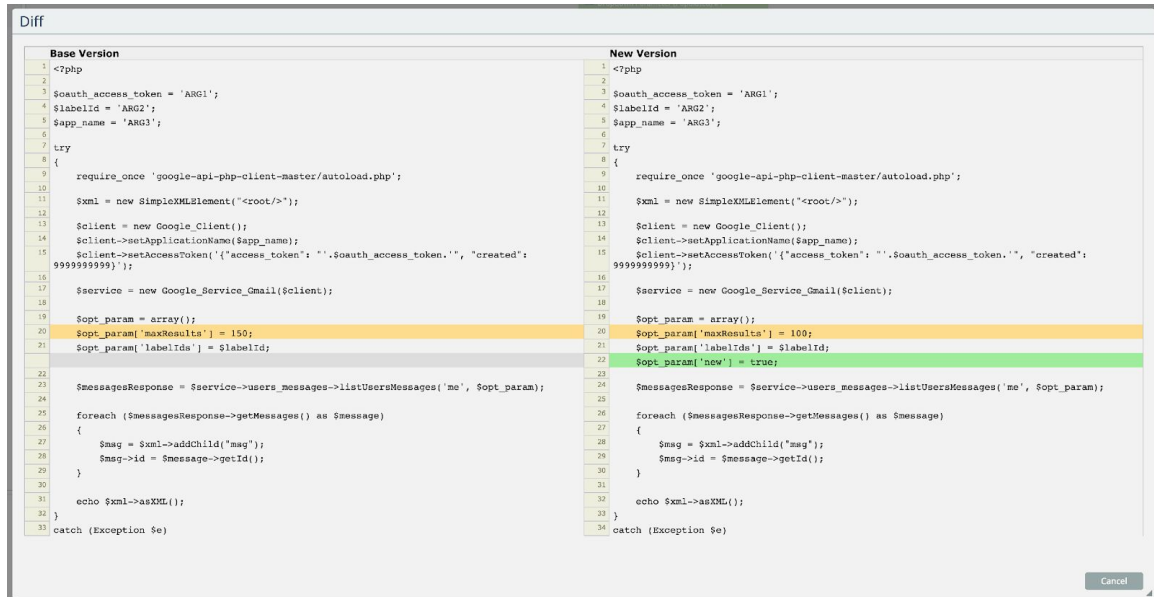


Clicking the **Compare** button will place the currently opened assembly into read-only mode and load the selected version of the diagram such that the two can be compared for differences:



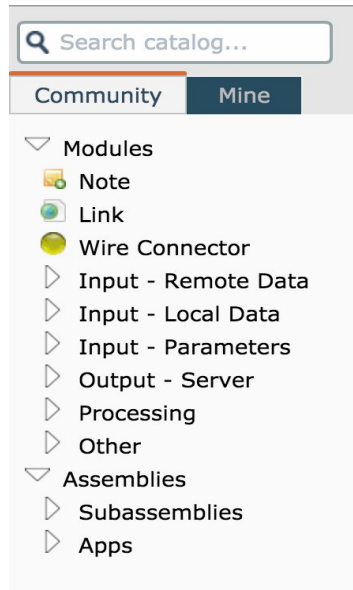
Initially the previously opened diagram is displayed first. Changing the Version Opacity slider at the top left will let either diagram be viewed, allowing changes to be compared. The legend at the top right shows the colors used to highlight removed/added/modified modules.

Clicking the pencil edit icon to edit inline code within Extension modules will display a diff view of the line-by-line code differences:



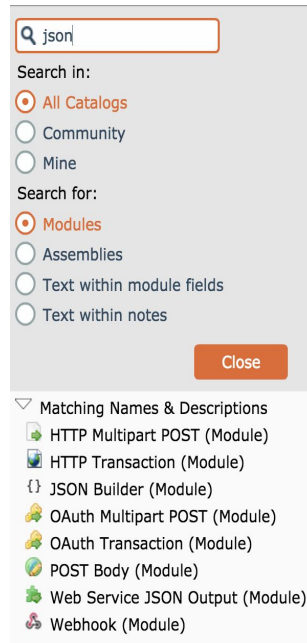
Catalog

The Catalog contains all public, private, and shared software modules, assemblies, and subassemblies:



Items in the Community tab are publicly available to all users in the system. Items in the Mine tab are those items created by you, including any items kept private. The Mine tab also contains items shared by you and items shared with you.

Performing a search opens the search options:



Search in:

All Catalogs

Community

Mine

Search for:

Modules

Assemblies

Text within module fields

Text within notes

Close

Matching Names & Descriptions

- HTTP Multipart POST (Module)
- HTTP Transaction (Module)
- JSON Builder (Module)
- OAuth Multipart POST (Module)
- OAuth Transaction (Module)
- POST Body (Module)
- Web Service JSON Output (Module)
- Webhook (Module)

By default the initial search is for modules matching the query in their names, descriptions, or tags. The initial search is performed within both the Community and Mine catalogs.

Select the appropriate radio button to perform the search again for the selected option.

The **Text within module fields** option searches assemblies with modules having configured fields that contain the value. The matching is performed case-insensitive. When an assembly in the search results is opened, matching module fields will be highlighted.

The **Text within notes** option searches assemblies with notes (yellow stickies) that contain the value. The matching is performed case-insensitive. When an assembly in the search results is opened, matching notes will be highlighted.



Note: Assemblies that are open when searches for module fields or notes are performed will not highlight any matching content.

Only newly opened assemblies from the search results will highlight matching content.

Click the **Close** button to close the search results.

Items in the catalog can be added to the assembly diagram by either clicking on them, or by dragging and dropping them into place within the wiring diagram.

Right-click on items in the module Catalog to view the item's context menu containing actions that may be performed on the item. The context menu options vary depending on which catalog and catalog section is selected, the account's permissions, system settings, etc.

Right-click context menu actions include:

Share, which shares private items with other user accounts

Make public/private, which controls privacy settings for items you own

View Sample Assembly, which opens a sample assembly demonstrating how to use a module

Find, which performs a search for the item where it is contained within other items, e.g. find all assemblies using a module

Tag, which allows tag keywords to be edited for the item. The tags can then be queried in the catalog

Get UUID, which returns the item's unique identifier

Properties, which displays the item's owner and timestamps for when the item was created and last edited

Delete, which deletes items from the system

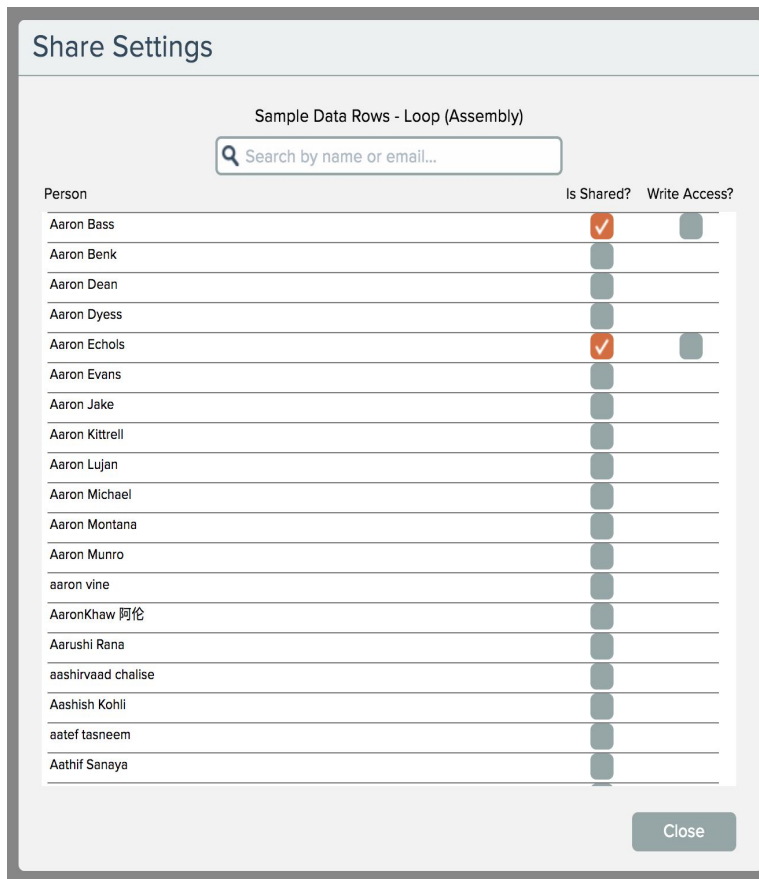
Export, which exports the item as a single export file, which can then be imported into another APIANT system

Publish to production server(s), which automatically performs an export from the sandbox system and imports the item to the production system

Sharing Catalog Content

Non-public content you have created can be shared with other users in the system. To share an item, right-click a non-public item in the catalog and choose the **Share with other accounts** option.

The Share Settings for the item will appear beneath the catalog, listing all user accounts in the system and the item’s access permissions:

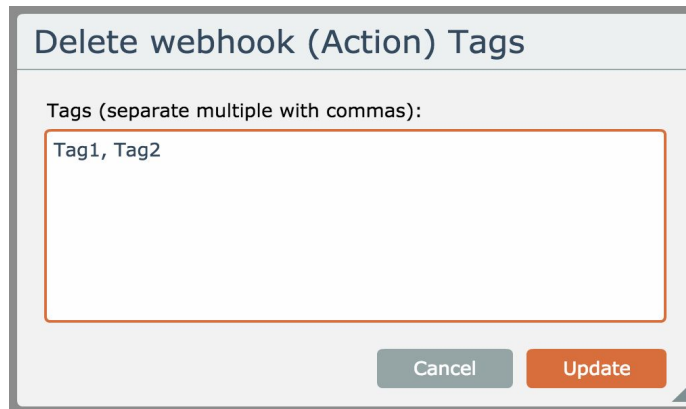


Only one user at a time can have write access to shared content. Users without write access can only save a copy of shared content. Shared modules and assemblies do not mirror updates in real-time when multiple accounts access the items.

Tagging Catalog Content

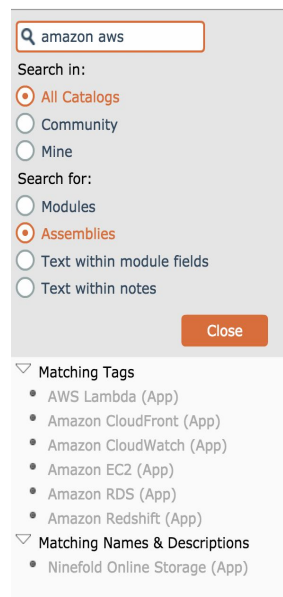
Items in the catalog can be tagged with keywords that can be searched upon. To edit an item’s tags, select the **Tags...** option from a catalog item’s right-click context menu:

A dialog window appears for editing the tags associated with the item:



Separate multiple tag keywords or phrases with commas. Click the **Update** button to save the tags for the item.

When a catalog search is performed, any items with matching tags will appear separately in the search results:



Information Area

The Information Area at the lower left of the editor displays help information and data previews.

Help information is displayed when either an item is moused over in the catalog, or when the question mark icon in module title bars are clicked:

View Sample Assembly

Sends a HTTP transaction to a URL and returns any response.

An XML response will be emitted as a data stream. A JSON response will be converted into XML and emitted as a data stream.

Double-clicking on a module in the editor will cause the engine to execute, if the module either contains no data or if changes to the modules in the diagram have been made. If the engine executes, it will stop execution on the module that was double-clicked. Any module output data will be displayed in the Information Area:

Open Stream Inspector...

Total Data Streams: 1

Stream: rss #1

Stream data:

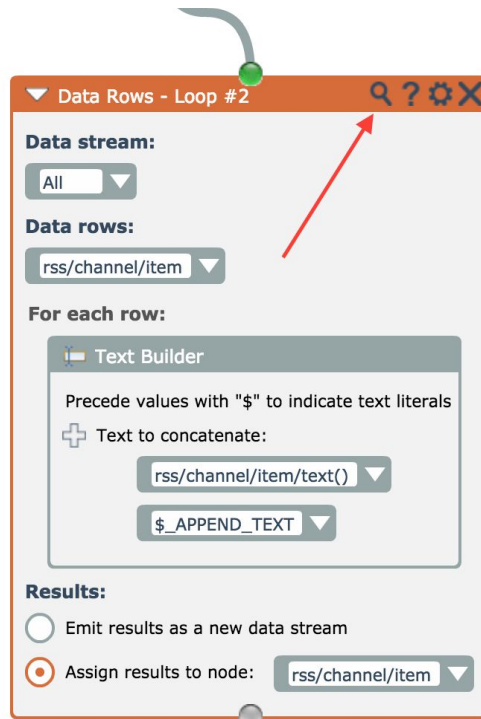
```

<rss>
  <channel>
    <item><![CDATA[1_APPEND_TEXT]]></item>
    <item><![CDATA[2_APPEND_TEXT]]></item>
    <item><![CDATA[3_APPEND_TEXT]]></item>
    <item><![CDATA[4_APPEND_TEXT]]></item>
    <item><![CDATA[5_APPEND_TEXT]]></item>
  </channel>
</rss>

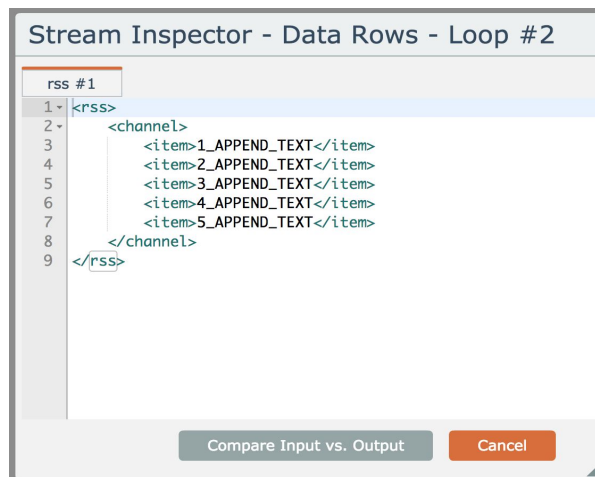
```

If the module emits data streams (XML documents), the **Open Stream Inspector** button will appear. The module will also display a

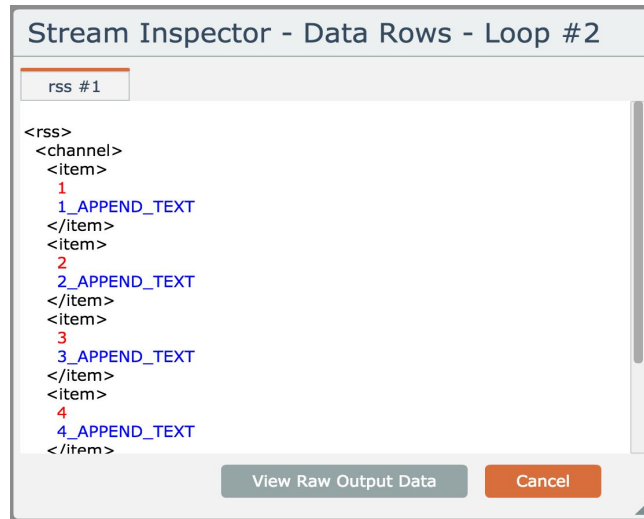
magnify icon in its window title bar to indicate data streams have been loaded into the module and are available for inspection:



Click either the **Open Stream Inspector** button or the magnify icon to open a dialog that will display all the available data streams in a syntax-highlighted viewer:



The Stream Inspector displays the raw data streams output from the module. To view the manipulations performed to the data stream, the input and output data streams can be compared, or “diff’ed”, by clicking the **Compare Input vs. Output** button:



The XML Diff view visually depicts the data modifications modules perform to data streams. The XML Diff view highlights removed data in red, inserted data in blue, and unmodified data in black.



Note: The XML Diff calculations can take a long time for data > 100KB. The diff calculations can be cancelled if you get tired of waiting for the results. Not all XML can be successfully Diff’ed. When this occurs, a message will appear.

Click the **View Raw Output Data** button to return to the raw data stream view.

Quick Picks

Underneath the wiring diagram canvas is a tray where frequently used catalog items can be saved for quick access:



Your quick picks are saved on the server and reloaded every time the Assembly Editor loads.

To add an item to the quick pick tray, drag an item from the catalog and drop it into the tray. The item can then be dragged from the quick pick tray and dropped into a wiring diagram when needed.

	<p>Note: Not all catalog items can be placed into the quick pick tray. Only catalog items that are draggable can be dropped into the tray.</p>
--	---

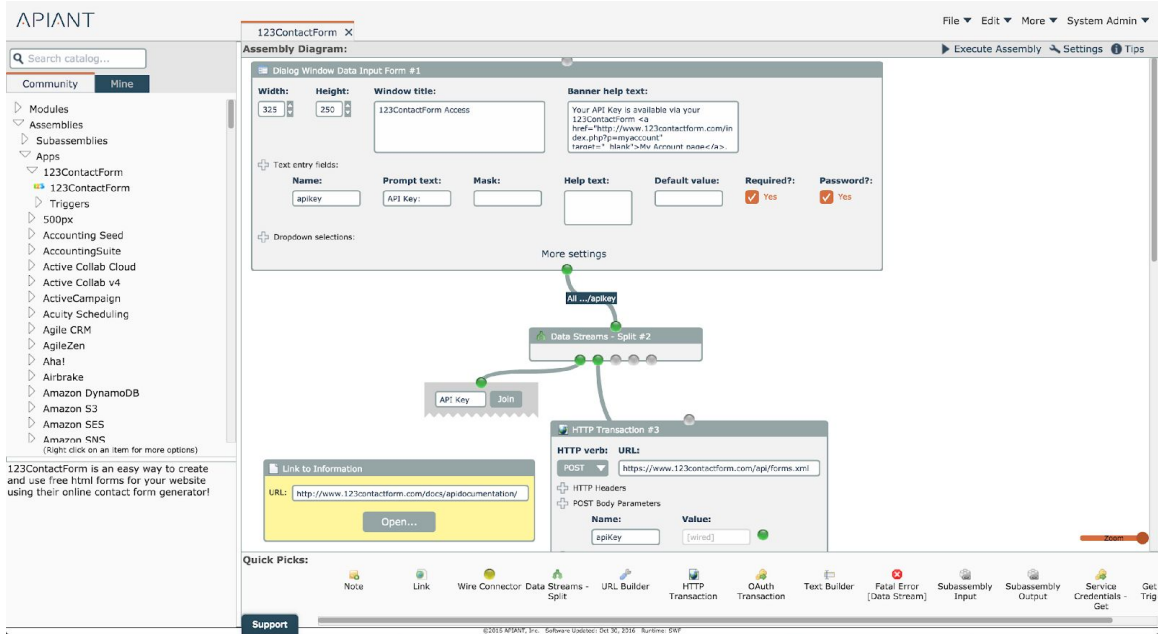
To remove an item from the quick pick tray, first click it to select it, then click the **X** delete icon:



Chapter 2: Working with Modules

Modules

Modules are the basic building blocks of assembly diagrams:

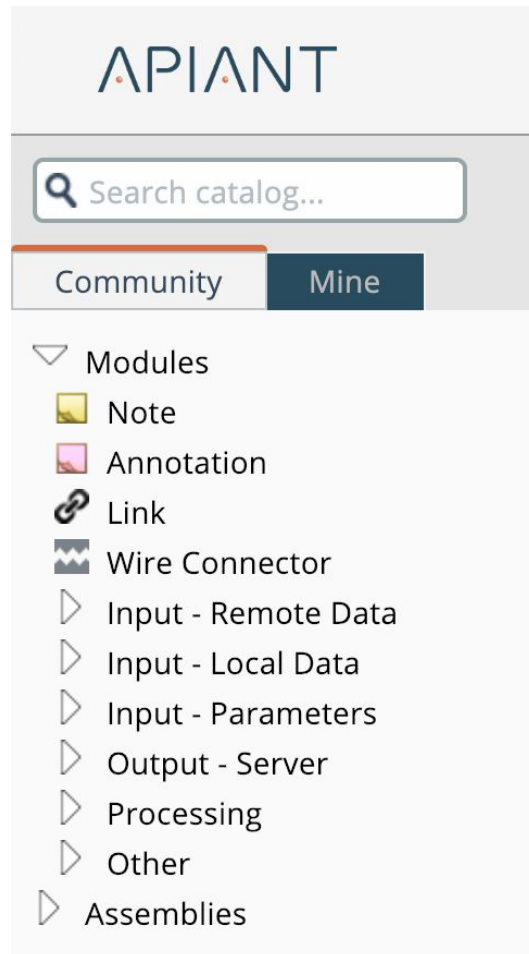


A module is a piece of software that performs specific functionality. Modules are constructed with the Module IDE, built into the Assembly Editor. If you are licensed to use the Module IDE, see the separate Module IDE guide for how to build new modules.

	<p>Note: Generally new modules don't have to be built because the system has Extension modules for inlining Java, PHP, and JavaScript code directly into assembly diagrams, providing an ad-hoc way to extend assembly functionality.</p> <p>The tradeoff is that native modules offer the best possible performance.</p>
--	--

APIANT

The baseline system has almost 200 modules within 6 top-level categories:

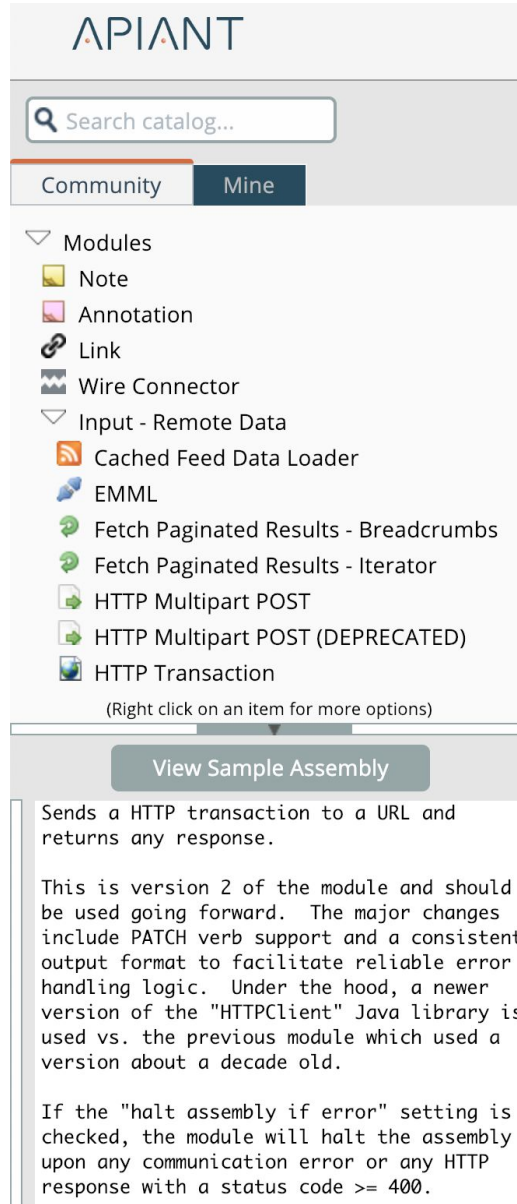


There is currently no standalone documentation that describes each baseline module in detail, but modules have help information and assemblies that demonstrate their usage.

You don't need to know or use all 200 modules for building API integrations. Generally fewer than three dozen modules are commonly used. A list of commonly used modules is at:

<https://intercom.help/apiant/integrators/most-commonly-used-modules-in-the-assembly-editor>

You can view help information for a module by mousing over it in the catalog, or by clicking on the "?" icon in its title bar when it is in a diagram:



The screenshot shows the APIANT interface. At the top is the APIANT logo. Below it is a search bar labeled "Search catalog...". There are two tabs: "Community" and "Mine". A list of modules is shown under a "Modules" heading, including Note, Annotation, Link, Wire Connector, and a sub-section "Input - Remote Data" containing Cached Feed Data Loader, EMMML, Fetch Paginated Results - Breadcrumbs, Fetch Paginated Results - Iterator, HTTP Multipart POST, HTTP Multipart POST (DEPRECATED), and HTTP Transaction. A tooltip is visible over the "HTTP Transaction" module, containing the following text:

(Right click on an item for more options)

View Sample Assembly

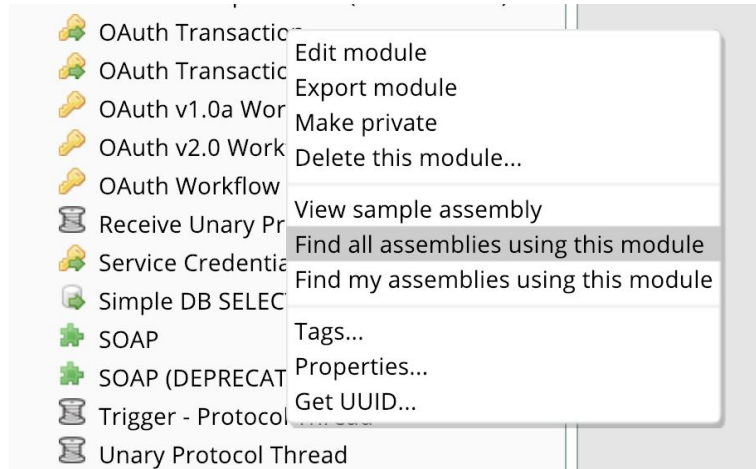
Sends a HTTP transaction to a URL and returns any response.

This is version 2 of the module and should be used going forward. The major changes include PATCH verb support and a consistent output format to facilitate reliable error handling logic. Under the hood, a newer version of the "HttpClient" Java library is used vs. the previous module which used a version about a decade old.

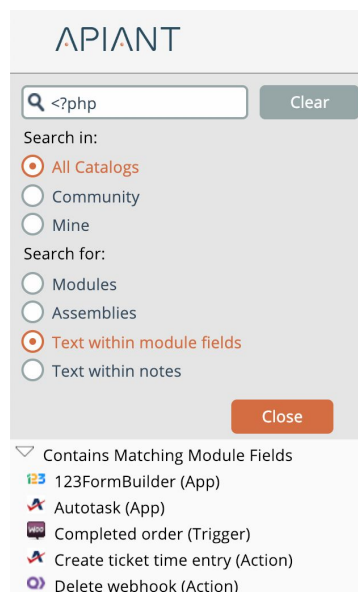
If the "halt assembly if error" setting is checked, the module will halt the assembly upon any communication error or any HTTP response with a status code ≥ 400 .

Click the **View Sample Assembly** button to open a diagram that demonstrates usage of the module.

Another helpful way to learn about a module is to view other assemblies that use it. Right-click on a module in the catalog and use the “find all assemblies using this module” menu option:



When using Extension modules for inlining Java, PHP, and JavaScript code, you can borrow code snippets by searching the catalog for modules containing specified text. For example, to find all assemblies containing any inlined PHP code, perform this search:



Be sure to select the “text within module fields” search option.

Chapter 3: Working with Assemblies

Module Wiring

Wires in the Assembly Editor are used to represent data flowing from module to module.

There are no circular paths within wiring diagrams. Wiring diagrams always flow top-down. To accomplish looping, various Loop Modules can be used in which a nested item (either a Module or Subassembly) can be placed to perform logic over many data rows within XML documents.

The Data Streams - Split module allows a wire to be split into identical outputs, facilitating differing operations upon the data that can later be recombined into a single data stream with the Data Streams - Union module.

To facilitate branching logic, the Conditional Execution modules can be used. These work like Loops, where a nested module or subassembly can be executed based on criteria.

Data in Assembly Editor wires is either:

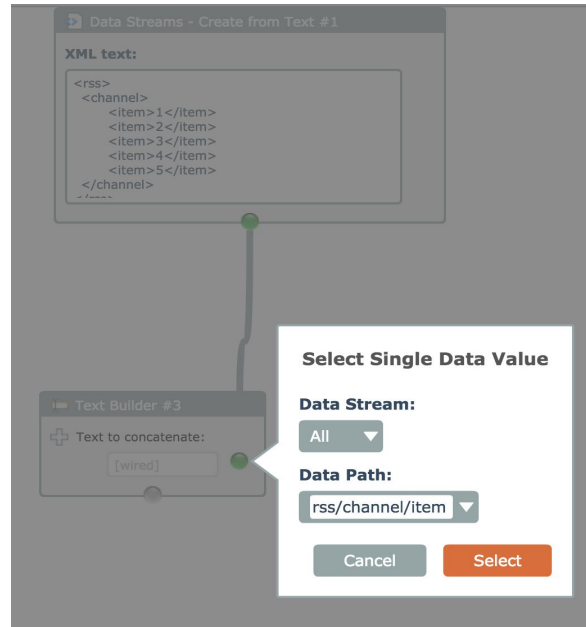
- Literal values like text and numbers
- One or more XML documents called **Data Streams**

Wire nodes at the top of modules represent input data into the module. Wire nodes at the bottom of modules represent output data out of the module. Fields inside of modules can accept input from the output of other modules.

Wire nodes at the top and bottom of modules have a data type that is typically "xml", "text", or "number". The data type can be inspected by mousing over the node.

Wires can only be connected to nodes of a compatible type.

If a wire containing data streams is connected to a text or number node, the editor will prompt to allow a value within a data stream to be used for the text or number value:



If the data path matches more than one node, commas will be placed between the values.

After clicking the Select button, a black label will appear on the wire to indicate what value is being extracted from the data stream:



The black label can be dragged into a different position if needed. Once moved, it will continue to follow its associated wire when the wire is moved.

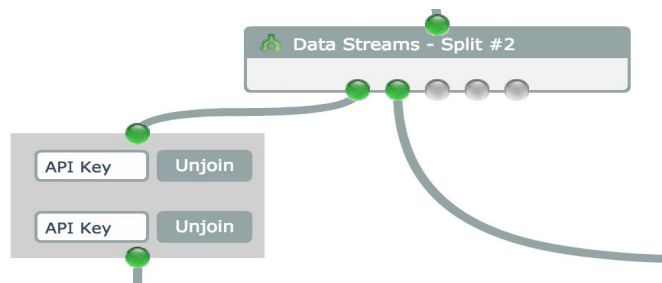
The black label can be clicked to select a different value from the data stream.

APIANT

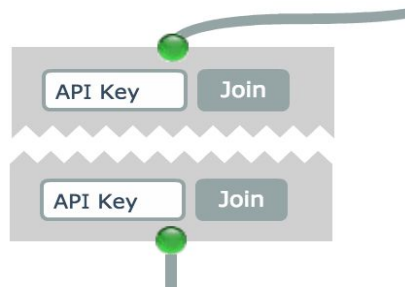
Wire Connectors allow a wire to logically connect from one point in the diagram to another point. The Wire Connector is important, so by default it is in the Quick Picks:




Wire Connectors can connect to any input or output node:



Give the wire connector a useful label so it is obvious what data it references. Then press the **Enter** key or click the **Unjoin** button to separate the connector into two pieces:




Then position the second piece where it is needed in the diagram. The **Join** button can be used if it isn't apparent where the other piece is within the diagram.

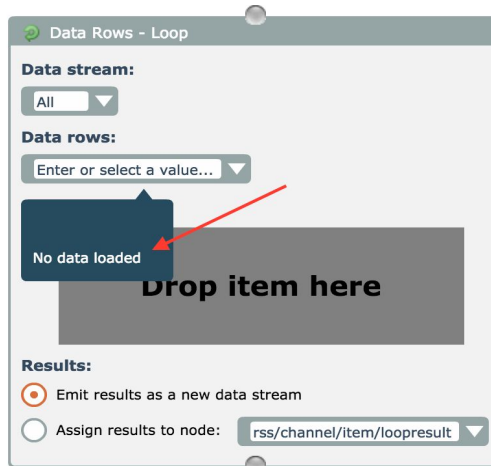
	Tip: Use Wire Connectors to avoid "spaghetti diagrams"!
---	--

Building Assemblies

Modules and other catalog items are added to diagrams by either clicking on them in the catalog, or by dragging and dropping them into place. Modules can be removed from diagrams via the **X** delete icon at the top right of their title bar.

	<p>Note: The numbers that appear in the module title bars do not represent their execution order. Modules are numbered top-down in the diagrams only to facilitate inspection of debug logs, to make it apparent which module is emitting log information.</p>
---	---

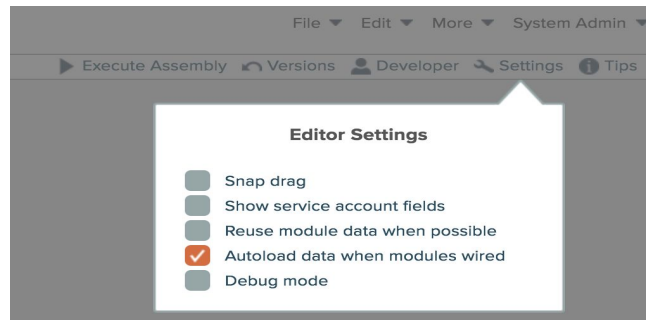
Many modules have fields that operate on data contained within XML data structures. When first added to the diagram, these dropdowns will be empty and show **No data loaded**:



The dropdowns do not become populated until the module is wired with data from another module. To do this, click the yellow output node from the bottom of a data source module and drag the wire to the yellow input node at the top of the destination module. Note that allowed target nodes for the wire will blink within the wiring diagram, as determined by the data type. The data type can be viewed with the mouseover tooltip by placing the mouse cursor over a node.

Once the module has been wired with a connection from a data source module, there are two ways the data fields become active and populated with any available data:

If the Autoload Data setting is checked, any available data from preceding modules in the diagram will be autoloading into the target module when the wire is connected.



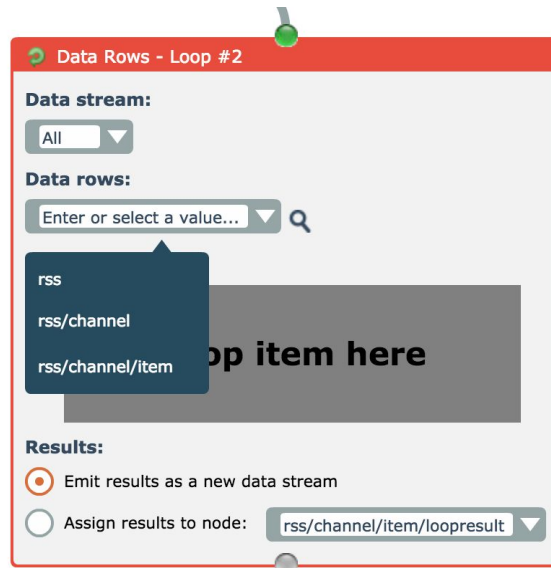
If the Autoload Data option is not checked, then the target module can be double-clicked on its title bar or background to execute the engine and load the module with available data from preceding modules in the diagram. The engine will halt execution on the module that was double-clicked.



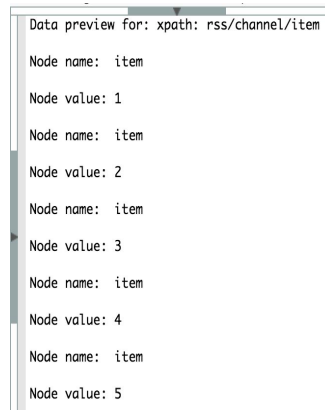
Note: When a module is double-clicked the assembly engine will only run if the module is "dirty", e.g. if any of its fields or settings have changed.

The Execute Assembly link at the top right of the editor can be used to execute the entire diagram.

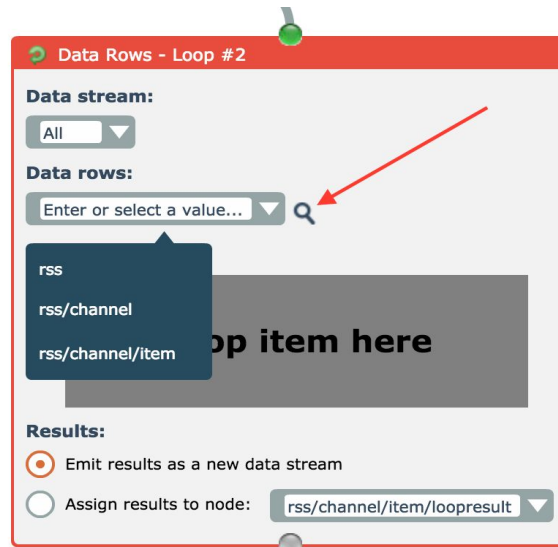
If the input data module returns valid data, the data field dropdown lists will now contain data paths parsed from the XML document:



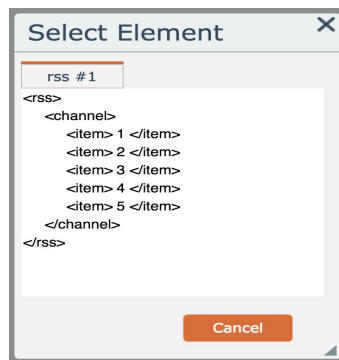
Moving the mouse cursor over data paths in the dropdown list will display matching data in the module Information view at the bottom left of the editor.



Another way to select data paths is to click the magnifying glass icon next to the dropdown lists. The icon only appears after data is loaded into the module:



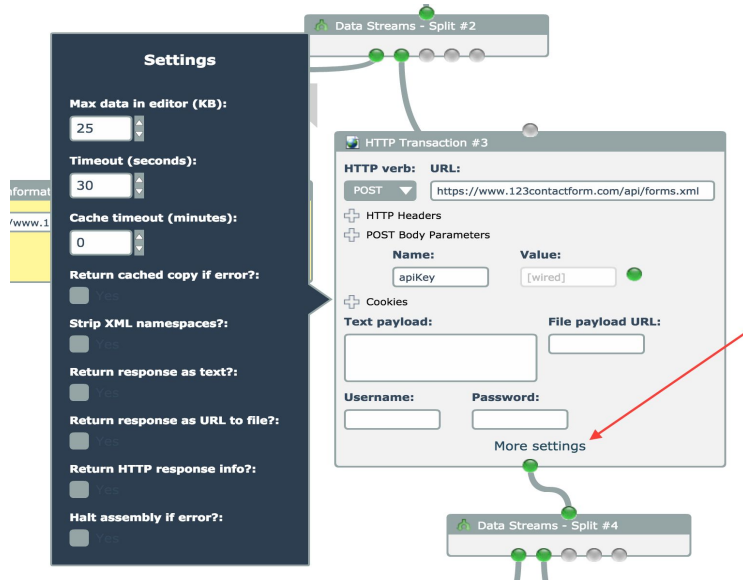
Clicking the icon will open a dialog displaying the available data loaded into the module:



The available data streams loaded into the module will appear as tabs. Select an available tab to view the data for the corresponding data stream.

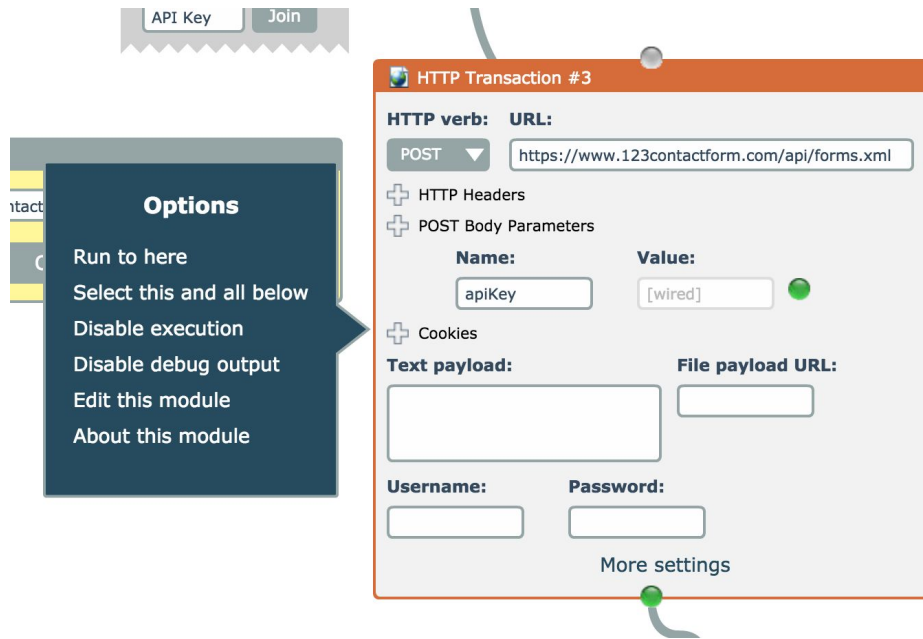
Moving the mouse cursor over various data elements will cause all corresponding elements matching the data path to blink. Clicking on an element that is blinking will close the dialog and populate the dropdown value with the selected data path.

Some modules have additional settings available from a **More Settings** link at their bottom:



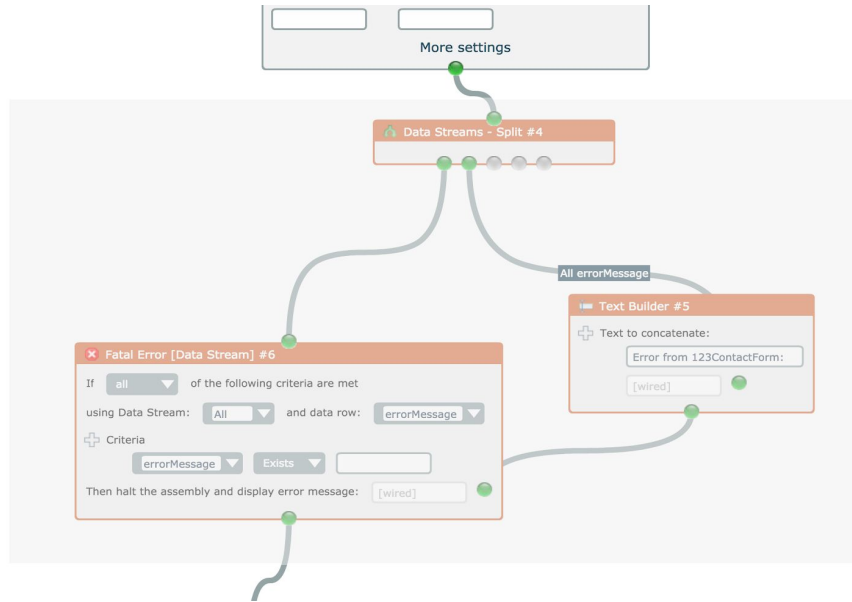
Unlike fields within the module's main body, settings cannot accept any wired values.

Modules can be clicked in their background body area to display options:




Editing Assemblies

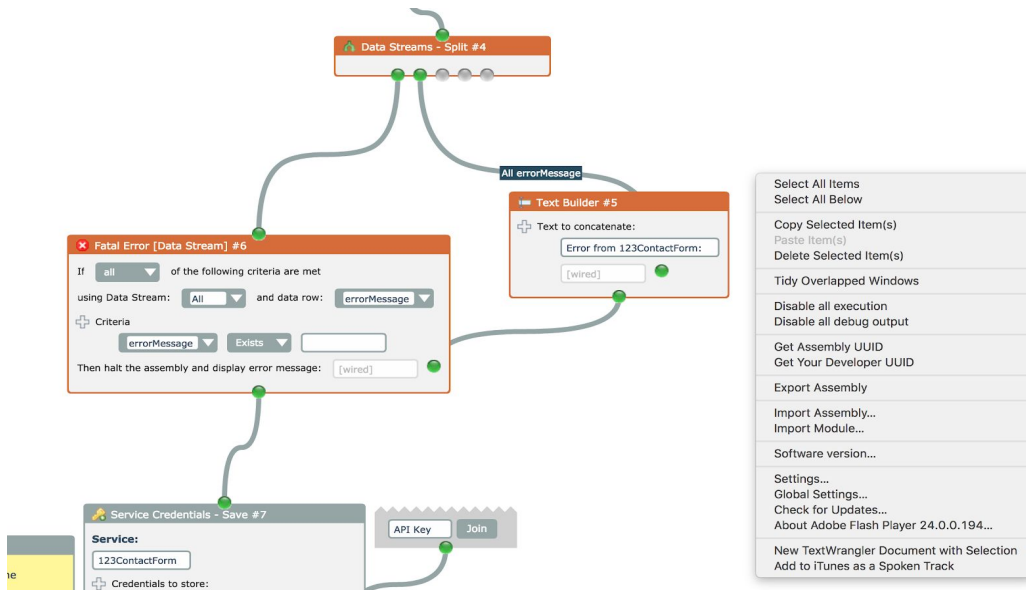
Multiple modules can be moved at once by first dragging a selection area around the modules:



Only modules entirely contained within the selection area will become highlighted. After selecting the desired modules to move, dragging any one of the selected modules will move all selected modules as a group together. Click the editor background to unselect the modules.

	<p>Tip: To easily make space to insert a module into an existing vertical logic chain, right click on the existing module at the insertion point and choose the Select This And All Below option, then drag the selected modules down to make room for the new module.</p>
---	--


When one or more modules are selected, options become available in the editor's right-click context menu to copy, paste, or delete the selected modules. Access the editor context menu by right-clicking on the background:



The **Select All Items** option will select all items in the wiring diagram.

The **Select All Items Below** option will select all items in the wiring diagram below the cursor.

The **Copy Selected Item(s)** option will place the selected items into the editor's clipboard. Afterwards, the Paste Item(s) option becomes enabled, allowing the copied modules to be pasted into the current assembly.

	<p>Tip: Modules can be copied into other assemblies by selecting the target assembly's tab in the editor.</p>
---	--

The **Delete Selected Item(s)** option will delete the currently selected modules from the current assembly. The option is only enabled when one or more modules are selected in the editor.

Cloning modules

Another way to copy a single item in the Assembly Editor diagram is to hold the **Control** key (use the **Command** key on macOS) while dragging the item.

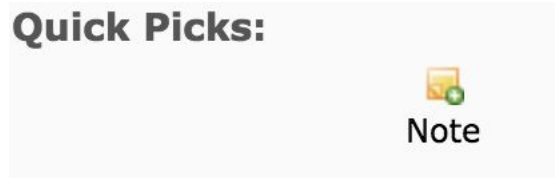
Doing so will create a clone of the item that can be placed into the diagram.



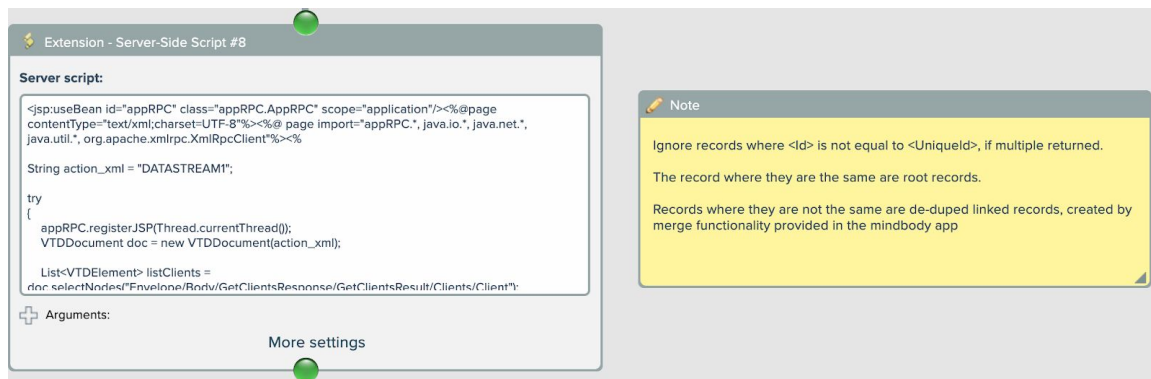
Tip: The main advantage of cloning modules is to be able to copy items into and out of modules that allow nesting, like the Loop and Conditional modules.

Documenting Assemblies

If you intend to share your assemblies with others, consider adding Notes and provide help information to others so they can understand what your assembly does. Notes can be added via the **Note** in the Quick Picks:



The note appears like a yellow sticky you can enter text in:



If an assembly is accessing an API, it can be helpful to include links to that API's documentation. A link can be added to the diagram via the **Link** in the Quick Picks:



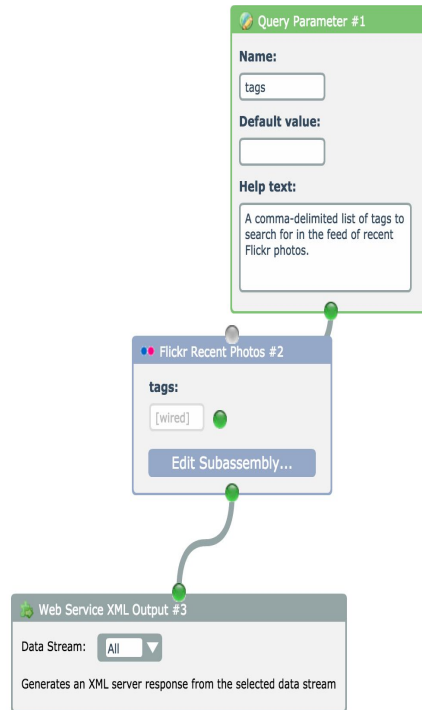
Then enter the URL to the web page that will be opened when the Open button is clicked:



Chapter 4: Subassemblies

Overview

Subassemblies promote the re-use of functionality and reduce the complexity of wiring diagrams:



A subassembly can consist of one or more modules or other subassemblies. Subassemblies can also contain nested subassemblies. Subassemblies appear with a blue window in the editor, to help them stand out from individual modules.

Subassemblies are edited and tested like other standalone assemblies.

Subassemblies are created when an assembly having both a Subassembly Input module and a Subassembly Output module is saved into the **Subassemblies** category:

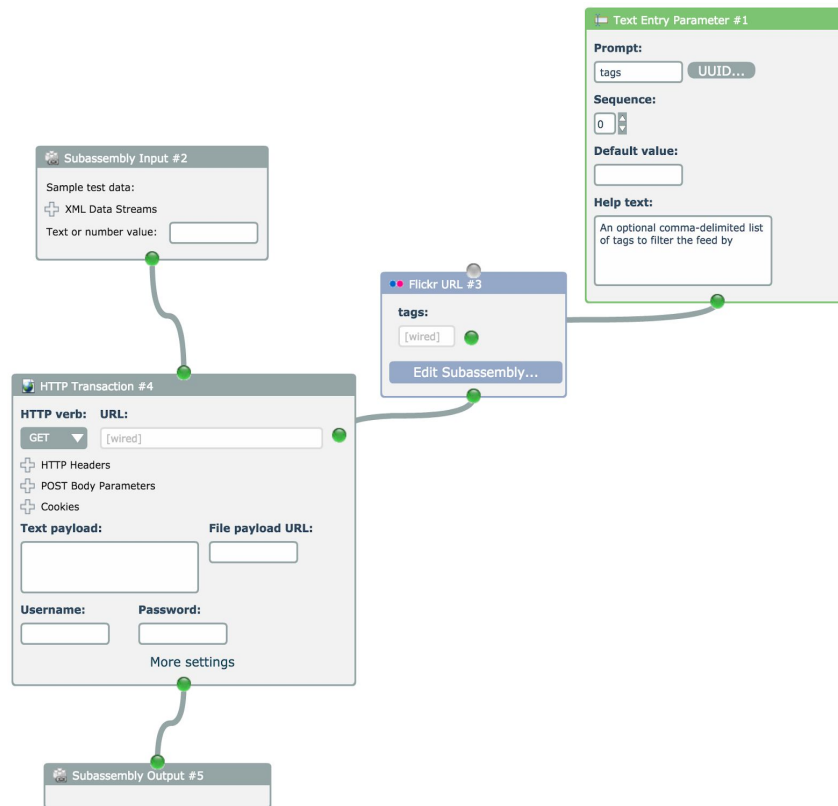
The screenshot shows a 'Save Assembly' dialog box with the following fields and options:

- Name:** Flickr Recent Photos
- Category:** Subassemblies (indicated by a red arrow)
- App:** (empty dropdown)
- Description:** A sample subassembly that returns the most recent public photos uploaded to Flickr.
- Tags (OPTIONAL):** Enter one or more tags separated by commas...
- Version message:** Enter a message about the changes in this saved version...
- Icon (16x16):** Icons...
- Template?:** Yes (checkbox)
- Public?:** Yes (checkbox)

Buttons at the bottom: Cancel, Save as New Copy, Save.

The saved subassembly can then be used within assemblies or other subassemblies.

All subassemblies must contain a Subassembly Input module and a Subassembly Output module, which define the entry point and input data into the subassembly, and the output data from the subassembly:



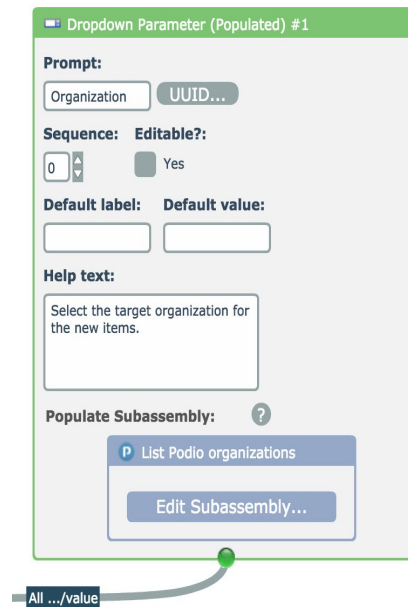
When an outer assembly invokes the subassembly, any data streams wired into the subassembly are emitted from the Subassembly Input module.

The Subassembly Input module can be configured with test data in the form of one or more XML Data Streams, or a text or number value. The test data is only emitted when the subassembly is executed within the editor.

The Subassembly Output module emits whatever is wired into it, which can be one or more data streams, or a text or number value.

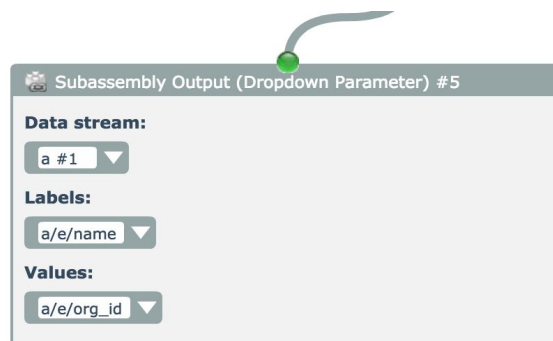
Dropdown Parameter Subassemblies

The Dropdown Parameter (Populated) module is used when configuring triggers or actions. A dropdown list is populated with the contents emitted by a nested subassembly:



The nested subassembly can perform any needed logic, such as fetching data from an API.

The subassembly must use a specialized version of the Subassembly Output module, called **Subassembly Output (Dropdown Parameter)**:

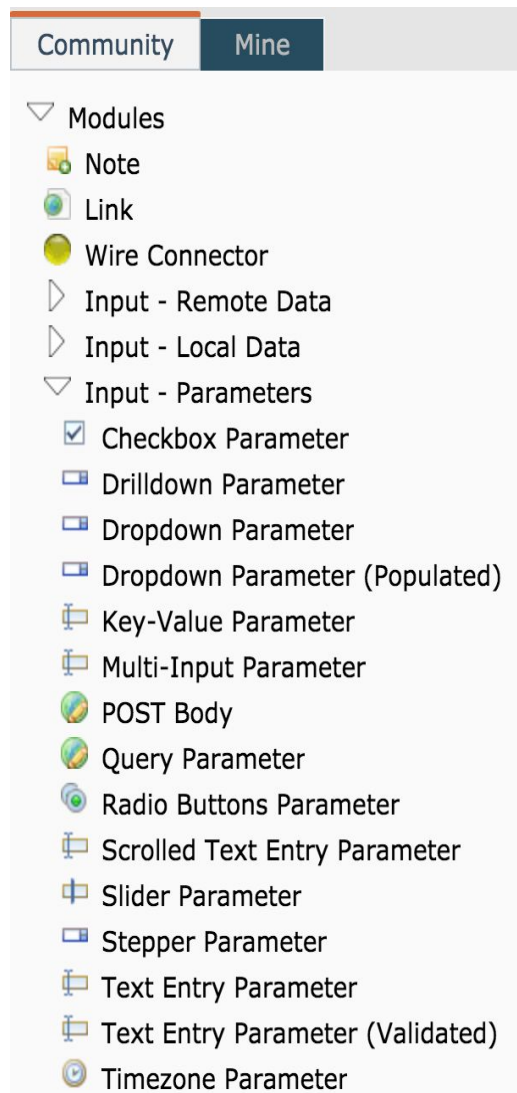


This specialized module allows the labels that appear within the dropdown to be specified, along with values for each label item.

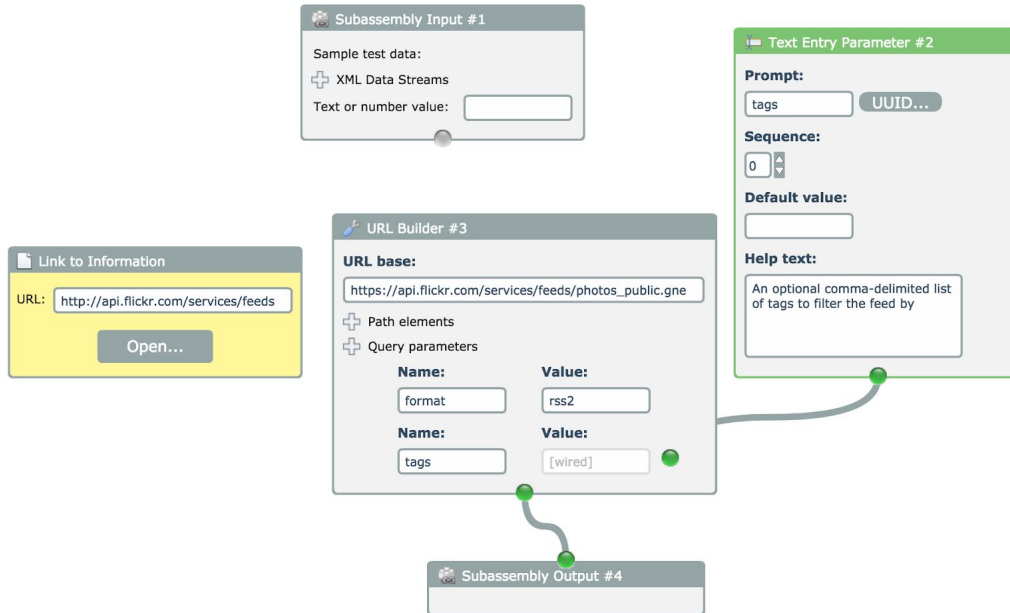
Subassembly Input Parameters

Subassemblies can be parameterized. Subassembly parameters make it possible to customize your subassembly's behavior based on configured values.

The **Parameter** modules contained in the module catalog under the **Input – Parameters** category are used to define subassembly parameters:



In the example below, the Flickr URL subassembly has a parameter **tags** defined:



The **tags** parameter is wired to the Flickr URL such that Flickr will return photos matching the search criteria contained within the **tags** parameter value.

When the Flickr URL subassembly is used, the **tags** parameter becomes exposed in the subassembly window as a configurable field value:



Now, the **tags** parameter can be configured without having to edit the Flickr URL subassembly.

Various types of Parameter modules are available, including checkboxes, sliders, and more. Each parameter value will appear using the UI control specified by the type of Parameter module used.



Note: If one or more parameters are defined and the subassembly is referenced by another assembly, then those existing parameters cannot be removed or altered in order to preserve referential integrity.

Chapter 5: API Integrations

Overview

Apps, triggers, and actions that appear in the Automation Editor are all built as assemblies in the Assembly Editor.

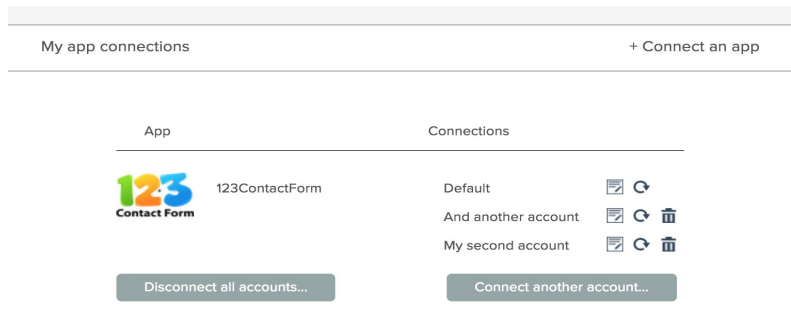


All facets of configuring apps, triggers, and actions in the Automation Editor are controlled by modules within their assemblies.

Automation execution and error handling are also defined via assembly logic.

Service Accounts

End users can connect multiple accounts to apps in the Automation Editor:

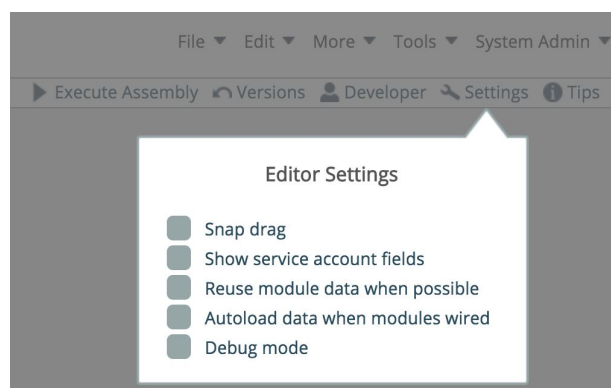


The first connected account is always named **Default**. End users must enter unique names for any additional connected accounts.

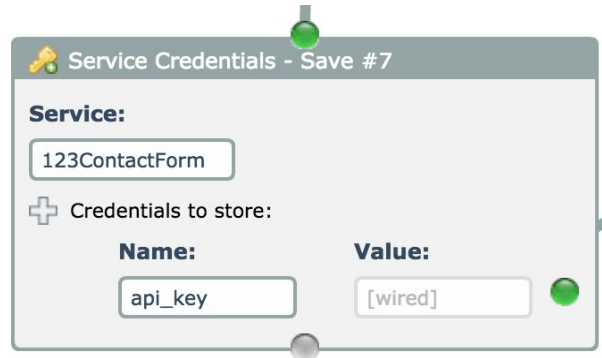
Accounts are associated with credentials needed to access API endpoints on the user's behalf. Certain modules in the Assembly Editor need to reference API credentials. They are:

- OAuth modules
- Service Credential modules

By default the Assembly Editor's settings has the **Show service account fields** setting disabled:

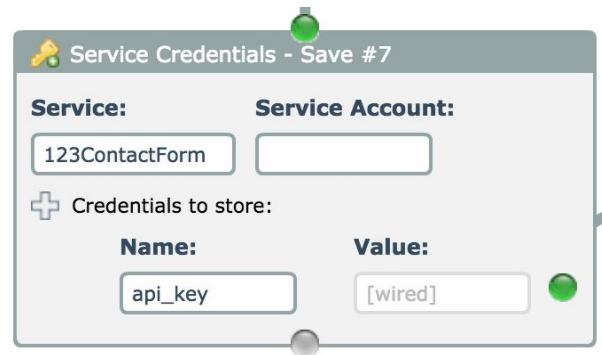


When the **Show service account fields** setting is disabled, modules that have service accounts don't show the service account field.



Typically when building triggers and actions in the Assembly Editor, development and testing is done using a single connected account, the Default account. Service account fields are unused and their presence is distracting since they are always empty.

Checking the **Show service account fields** setting causes them to appear:



When the fields are blank they reference the Default account. Entering any other value will cause the modules to attempt to load credentials for the specified app using that entered name. If the credentials are not found, an error occurs.

Entering service accounts manually in the Assembly Editor can be needed when troubleshooting, in the case where a user is having trouble with a certain connected account.

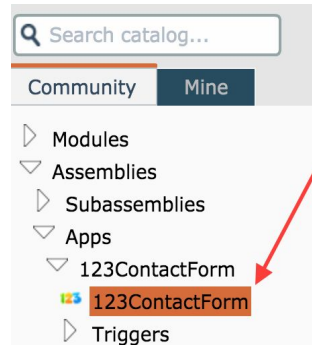
Service account fields also make it possible to build assemblies that reference multiple accounts.



Note: The Automation Editor does not know if a trigger or action references multiple service accounts. It is up to the developer of the trigger/action to inform users that they must connect certain named accounts that the assemblies expect to exist.

App Assemblies

App assemblies appear in the catalog as top-level assemblies for app integrations:



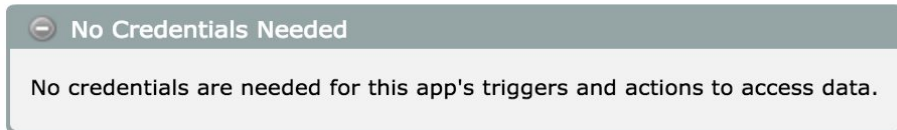
The purpose of app assemblies is to obtain and validate any needed user credentials in order for trigger and action assemblies to make API calls on their behalf.

The basic patterns for obtaining user credentials are:

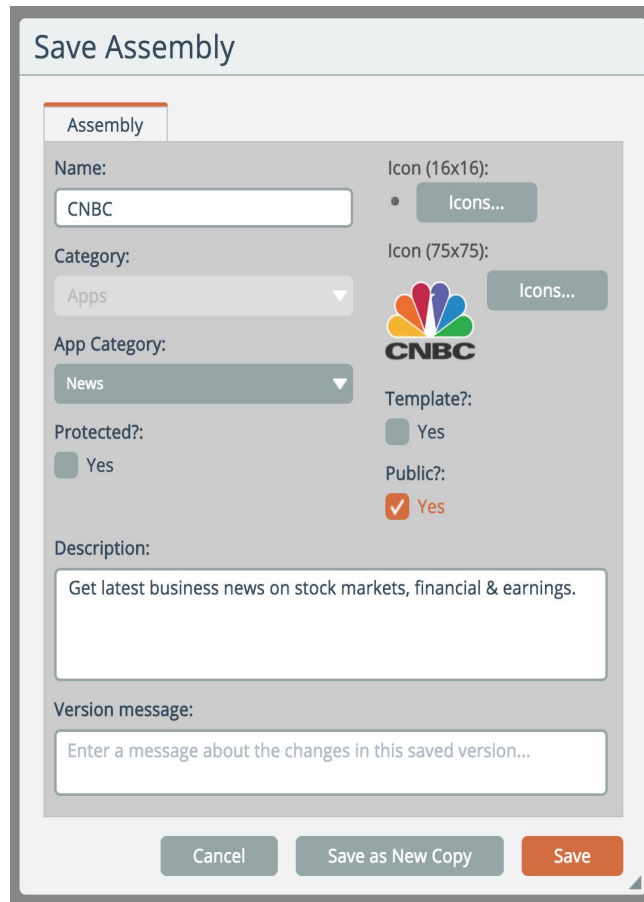
- No credentials needed
- The user needs to enter credentials like a username/password or API key
- The user needs to perform OAuth workflow

No Credentials Needed

Sources of data like RSS news feeds need no authentication to access. In this case, the app assembly just needs a single **No Credentials Needed** module:



The purpose of this module is to just allow the assembly to be saved as an app assembly:



The "Save Assembly" dialog box is shown with the following fields and options:

- Assembly** (tab)
- Name:** Text input field containing "CNBC"
- Category:** Dropdown menu showing "Apps"
- App Category:** Dropdown menu showing "News"
- Protected?:** Yes
- Icon (16x16):**
- Icon (75x75):** (with the CNBC logo selected)
- Template?:** Yes
- Public?:** Yes
- Description:** Text area containing "Get latest business news on stock markets, financial & earnings."
- Version message:** Text area containing "Enter a message about the changes in this saved version..."

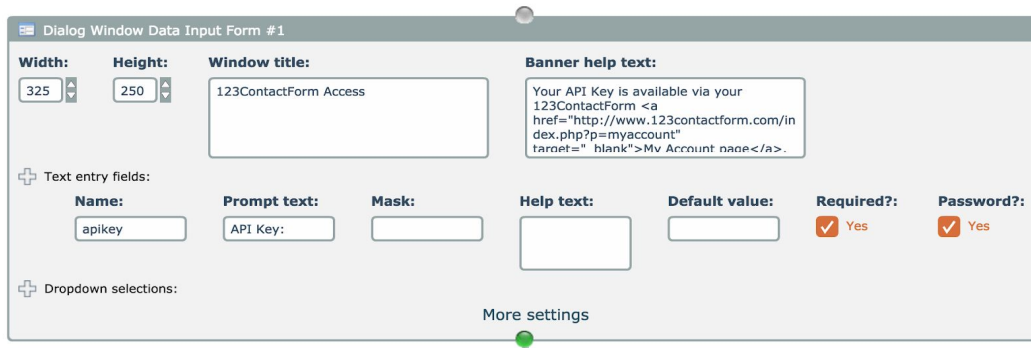
Buttons at the bottom:

User-Entered Credentials

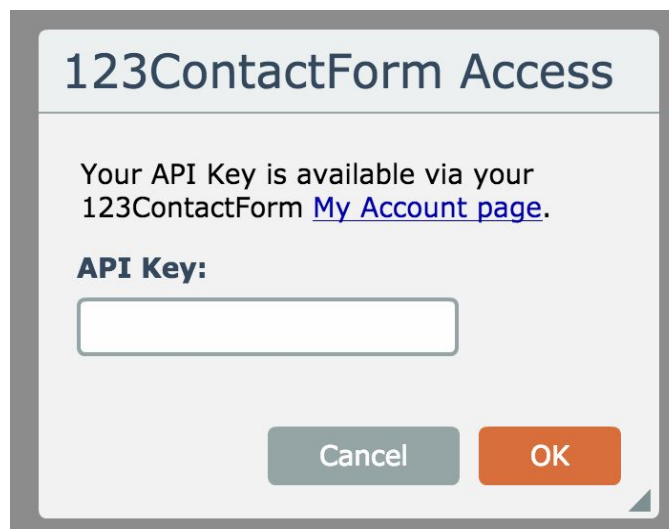
Some API's are accessed with credentials that must be entered by the user, like a username and password, or an API key.

The 123ContactForm assembly is an example that should be copied and modified for these types of apps.

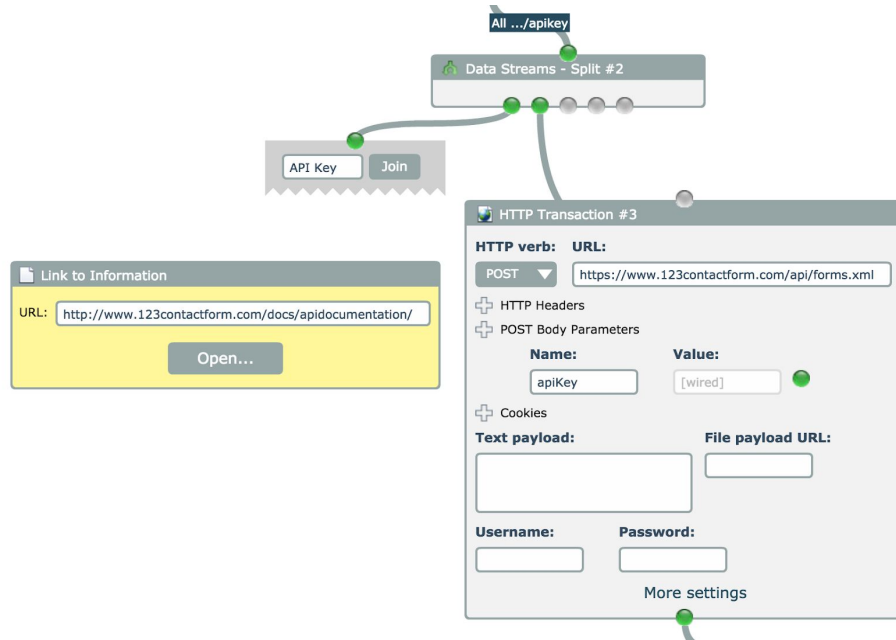
The first module presents a dialog window where the user can enter the needed information:



The module is configured with help information to explain where the user can find needed information. This is how the module appears when executed:

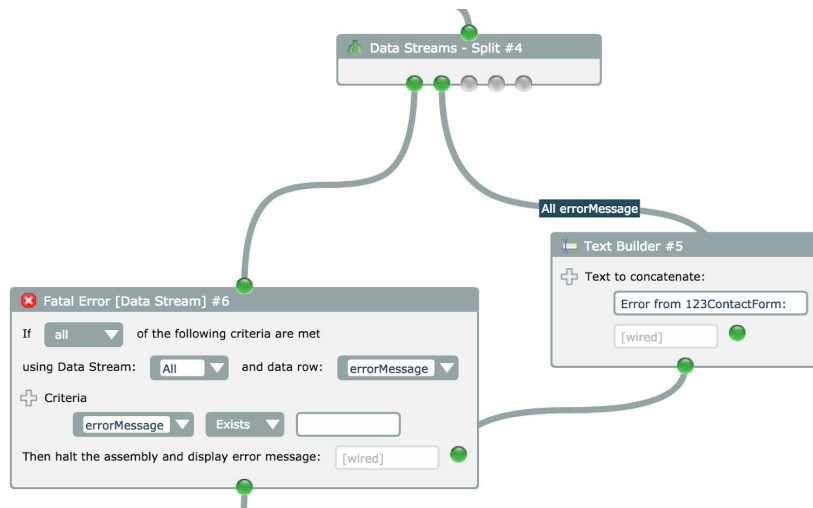


The next task for the 123ContactForm app assembly is to validate the entered API Key by performing an API call with it using the **HTTP Transaction** module:

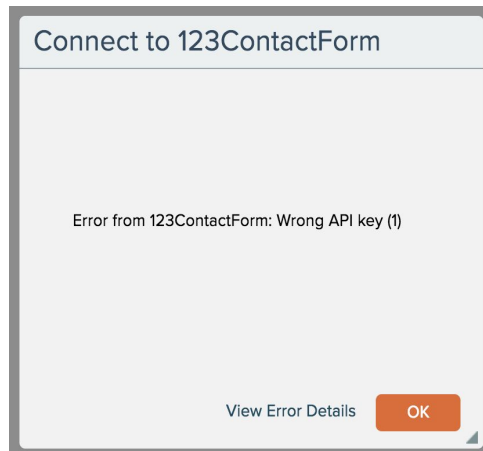


A helpful link to the API documentation is provided that explains how the API expects to receive the API Key.

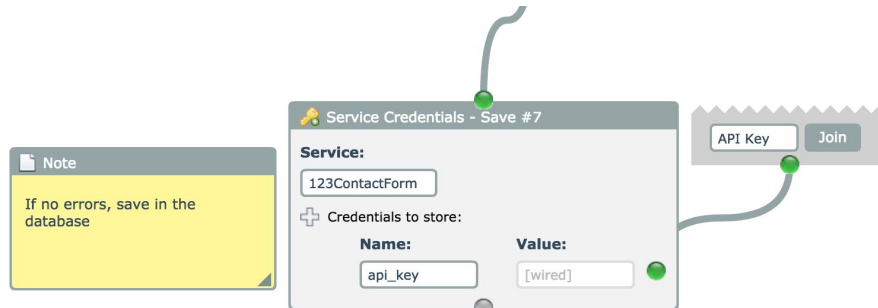
The next section of the app assembly then checks the API response for an error that indicates the entered API key is not valid:



When run in the Automation Editor, this is what appears when an invalid API Key is entered:



The final section of the 123ContactForm app assembly saves the API Key into the system's database via the Service Credentials – Save module:



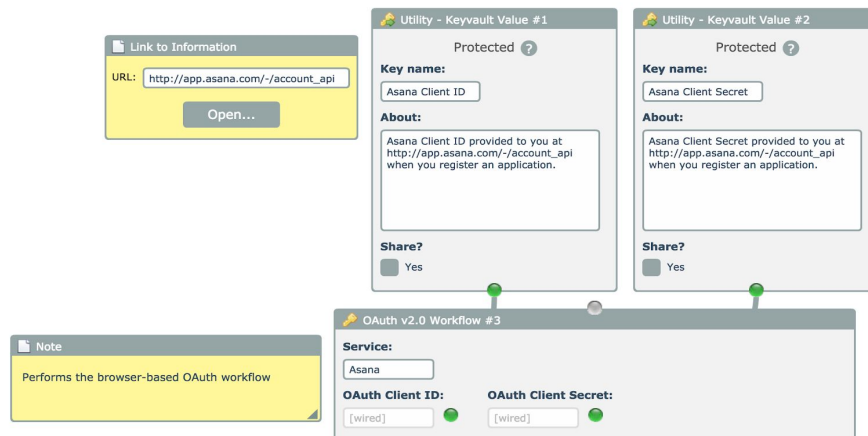
Triggers and actions can retrieve the API Key from the system's database using the **Service Credentials – Get** module.

OAuth Integrations

Most major API's use the OAuth standard for authorizing user access to their API. APIANT currently supports standard implementations of both OAuth v1.0a and v2.0.

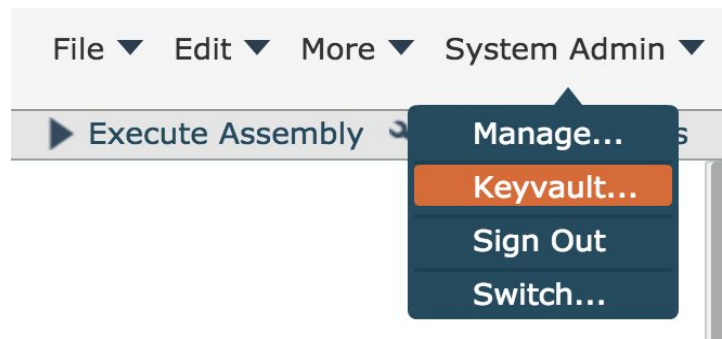
OAuth v1.0a and v2.0 integrations are done essentially the same. The only difference is that a different module is used in the app assembly depending upon which version the API uses.

See the Scoopit app assembly for an example of a standard v1.0a integration. See the Asana app assembly for a standard v2.0 integration.

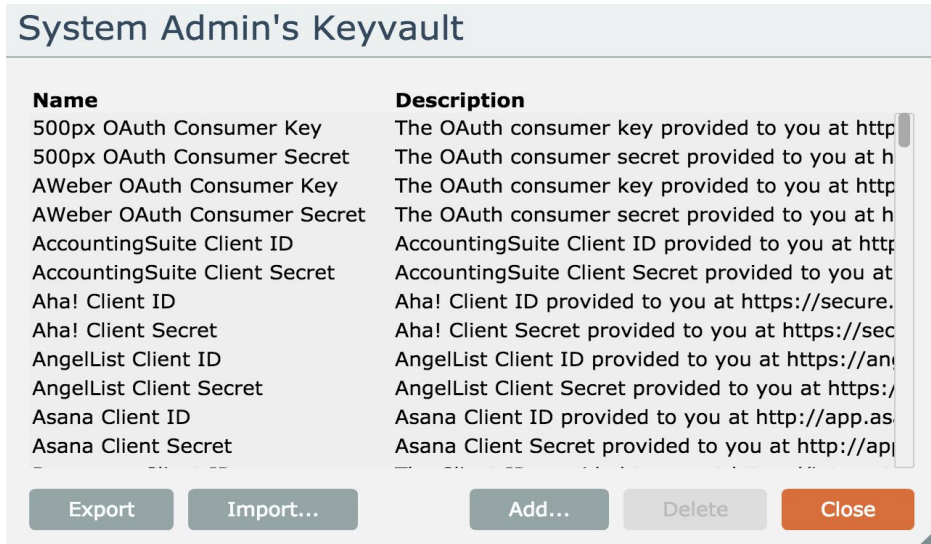


All OAuth app assemblies need to safeguard the API keys provided by the API vendor when you register to access their API. This is done using the **Utility – Keyvault Value** modules as shown above.

Your keyvault is accessible from the account menu:



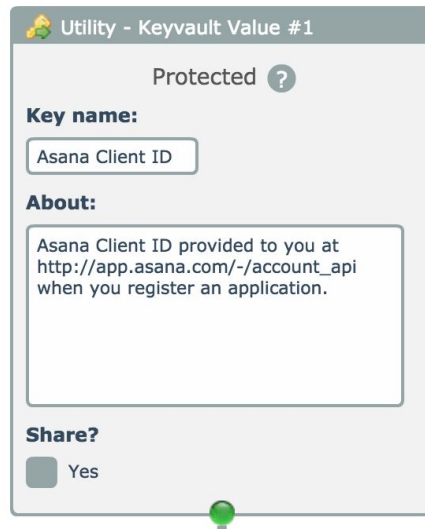
The keyvault is used to store values securely in the system's database:



Name	Description
500px OAuth Consumer Key	The OAuth consumer key provided to you at http
500px OAuth Consumer Secret	The OAuth consumer secret provided to you at h
AWeber OAuth Consumer Key	The OAuth consumer key provided to you at http
AWeber OAuth Consumer Secret	The OAuth consumer secret provided to you at h
AccountingSuite Client ID	AccountingSuite Client ID provided to you at http
AccountingSuite Client Secret	AccountingSuite Client Secret provided to you at
Aha! Client ID	Aha! Client ID provided to you at https://secure.
Aha! Client Secret	Aha! Client Secret provided to you at https://sec
AngelList Client ID	AngelList Client ID provided to you at https://an
AngelList Client Secret	AngelList Client Secret provided to you at https:/
Asana Client ID	Asana Client ID provided to you at http://app.as
Asana Client Secret	Asana Client Secret provided to you at http://ap

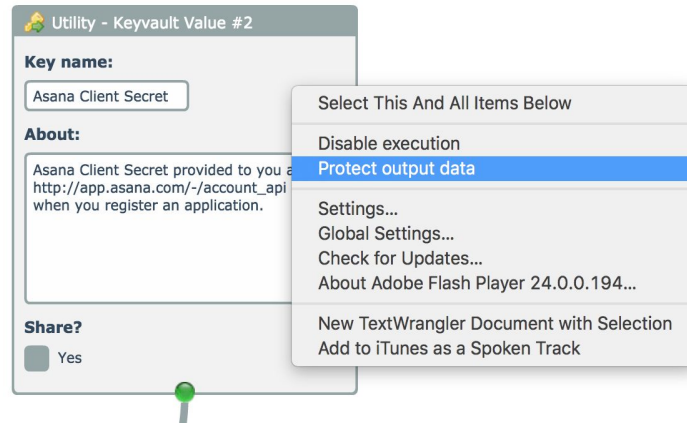
Export Import... Add... Delete Close

The names correspond to the key names entered in the **Utility – Keyvault Value** modules to extract the stored value:





Notice that the top of the module shows **Protected**. This indicates the output from the module cannot be viewed by other accounts when the assembly is executed in the Assembly Editor.

To protect output data for a module, right-click on the module and choose the **Protect Output Data** option:



When protected, other user accounts cannot view the output data from the module when executing the assembly in the Assembly Editor.

	Tip: Any module can have its output data protected. Consider using this whenever sensitive information needs to be hidden from other accounts in the system.
--	---

	Tip: If a value from a protected module is passed to other modules before it reaches a destination field, then those other modules should also have their output data protected.
---	---

OAuth v2.0 Token Refresh

Some OAuth v2.0 services require that the access token be refreshed. They will provide a refresh token URL for this, which must be configured within the **OAuth v2.0 Workflow** module:

Refresh token URL:	HTTP verb:	Use HTTP body params?:
<input type="text" value="https://app.asana.com/-/oauth_token"/>	<input type="button" value="POST"/>	<input checked="" type="checkbox"/> Yes

Follow their documentation to know which HTTP verb is needed (POST is used most often) and if the OAuth parameters need to be sent as HTTP body parameters or not.

OAuth Access Token Expiration

OAuth access tokens can expire due to these circumstances:

- The user chooses to revoke access by performing an action in the API provider's app.
- The OAuth v2.0 API provider requires that the access token be refreshed but the OAuth v2.0 Workflow module's settings for refreshing the token are not correct.

Whenever an access token becomes invalid, usually the API will return a 401 HTTP status code. If an automation encounters a 401 HTTP status code when executing a trigger or action that uses OAuth, the system performs these steps:

- The user's connected account is disconnected by removing their credentials from the database
- All of the user's automations that use the connected account are turned off

Trigger and Action Commonality

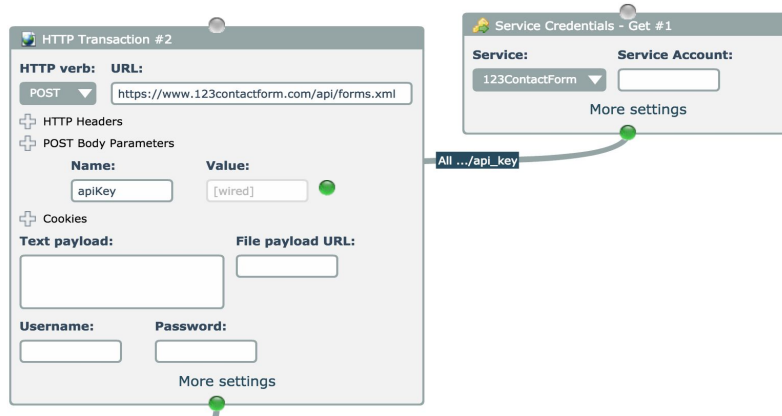
Trigger and Action assemblies share some common features and behaviors.

API Credentials

Most triggers and actions invoke API's that require either user-entered credentials or OAuth credentials.

The OAuth Transaction module and its variants automatically fetch the user's credentials.

If an HTTP Transaction module or its variants is used, the **Service Credentials – Get** module is needed to fetch API credentials saved by the App Assembly:



In both cases, the Service Account fields should be left empty so that the system will automatically use the account selected by the end user when building automations in the Automation Editor.

Error Handling

When a Trigger or Action assembly returns an error, the following occurs:

- The automation stops its processing immediately
- If the automation's owner has configured it to report any error, an email alert is sent to the automation's owner
- An email alert is sent to the system admin

Errors can be triggered in the following ways:

- When a Fatal Error module's criteria is met
- When a HTTP Transaction module or OAuth Transaction module, or any of their variants, has the "halt assembly if error" setting enabled and the API either returns an HTTP status code ≥ 400 or no data can be returned due to network-level errors
- When any module returns an error due to misconfiguration, missing required data, or processing errors

The **Turn Off Automation** module can be placed within a **Conditional** module if a permanent error is detected such that it makes no sense to keep the automation running. An example is an action that writes rows to a spreadsheet. If the spreadsheet no longer exists, the automation should be halted.

Automatic Error Retries

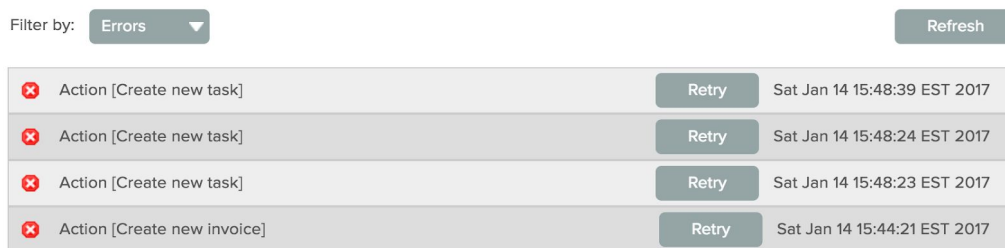
The system can be configured to recognize some errors as being retryable, where the system will automatically retry the trigger data row. By default, the system is configured retryable errors include:

- When a HTTP Transaction module or OAuth Transaction module, or any of their variants, has the "halt assembly if error" setting enabled and the API returns an HTTP status code ≥ 500
- When a HTTP Transaction module or OAuth Transaction module, or any of their variants, has the "halt assembly if error" setting enabled and a communication-level error occurs before the API can send a response

Generally, only transient errors are configured to be retryable, where it makes sense that the failed transaction might succeed after waiting for a short period of time.

When the system performs an automatic retry, the trigger data row will be retried after 5 minutes, then after 10 minutes, then after 15 minutes.

If after three automatic attempts the data row still fails, the data row can be manually retried from the Automation Editor via the automation's history screen:



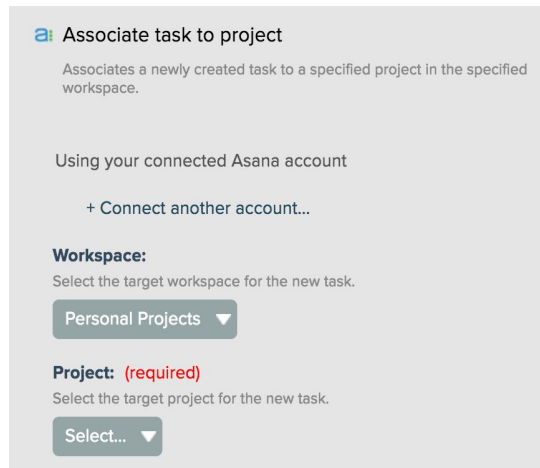
The screenshot shows a table with a filter dropdown set to 'Errors' and a 'Refresh' button. The table contains four rows, each representing a failed action. Each row has a red 'x' icon, the action name, a 'Retry' button, and the timestamp.

Filter by:	Errors	Refresh	
✘	Action [Create new task]	Retry	Sat Jan 14 15:48:39 EST 2017
✘	Action [Create new task]	Retry	Sat Jan 14 15:48:24 EST 2017
✘	Action [Create new task]	Retry	Sat Jan 14 15:48:23 EST 2017
✘	Action [Create new invoice]	Retry	Sat Jan 14 15:44:21 EST 2017

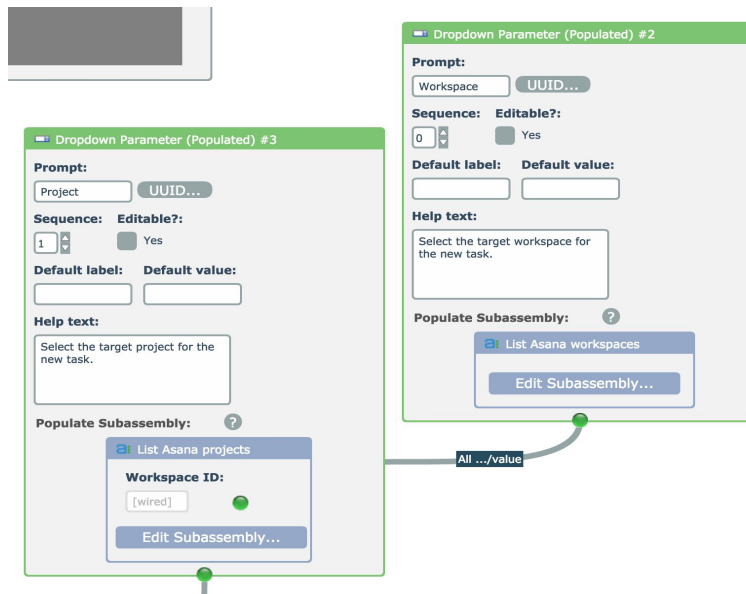
The system retries failed transactions at the point of failure in the automation's action logic. If the failed action succeeds, the remainder of the automation's logic is executed. This is for both automatic and manual retries.

Configuration Settings

Triggers and Actions can display configuration settings in the Automation Editor:



Configuration settings are displayed for the corresponding Parameter modules placed into the trigger and action assemblies:



In this example, the end user must first choose a workspace and then select a project within the workspace. The List Asana Projects subassembly uses the selected workspace's id to fetch the projects for

the selected workspace. The **Sequence** number determines the display ordering of Parameter modules in the Automation Editor.

Trigger Assemblies

Trigger assemblies cause the actions in an automation to be executed when their criteria is met. Typically triggers will fire upon new or updated data.

The main categories of triggers are:

- Webhook triggers that receive new or updated data pushed from an external system. These are also called Instant Triggers.
- Protocol thread triggers that maintain an always-on connection for sending and receiving data by running in threads on the server. Generally a SDK is used that under the hood will maintain a socket connection and communicate with a protocol over the socket. An example is the XMPP protocol for Jabber. Another example is a message queue listener. Protocol triggers are also Instant Triggers.
- Polling triggers that periodically check an API for new or updated data, for example every 15 minutes.
- Date/Time triggers that fire at a certain date and time, or on a periodic time schedule like once an hour.
- Gated triggers that fire only once each time a threshold criteria is met, like a weather trigger that fires when a temperature goes below a certain value.

Webhook Triggers

Webhook triggers, also called Instant Triggers, are the most efficient type of triggers. 3rd-party systems push data via a webhook URL, which immediately results in an automation being executed.

Webhooks come in two variations:

- Webhooks that end users must manually configure in the 3rd-party app
- Webhooks that can be registered via the app's API

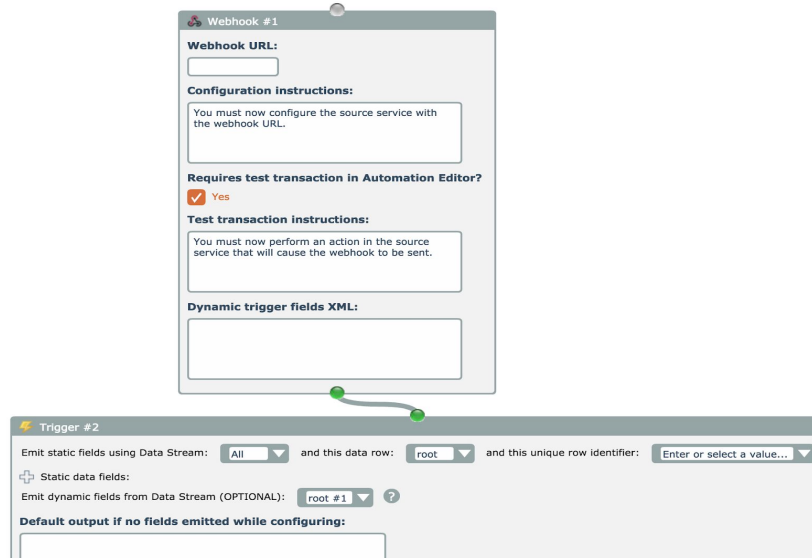
There are further variations within each of those:

- Webhooks with one event type per webhook URL
- Webhooks with multiple event types sent to a single webhook URL per customer account
- Webhooks with multiple event types and multiple customer accounts all delivered to a single webhook URL

All webhook triggers contain the **Webhook** module.

Manually Configured Webhooks

Some apps only allow webhooks to be manually configured by end users. For these apps, the Webhook modules **Webhook URL** field must be left blank:



When the **Webhook URL** field is blank, the module will generate a webhook URL when the assembly is either executed in the Assembly Editor (for testing) or in the automation editor. Configuration instructions should be entered instructing the end user how to configure the webhook in the app.

The Webhook module should have the **Requires test transaction** checkbox checked if the data schema for the webhook's payload is unknown or not always the same.

If the Webhook always returns the same data schema, then uncheck the **Requires test transaction** checkbox and enter the output XML from the Webhook module into the **Dynamic trigger fields XML** field after running the assembly in the Assembly Editor first. This field XML defines the output from the Trigger module.



Note: You must add CDATA escaping to the Raw POST Payload 'value' and 'helptext' nodes when entering the Webhook module's output into the **Dynamic trigger fields XML** field.

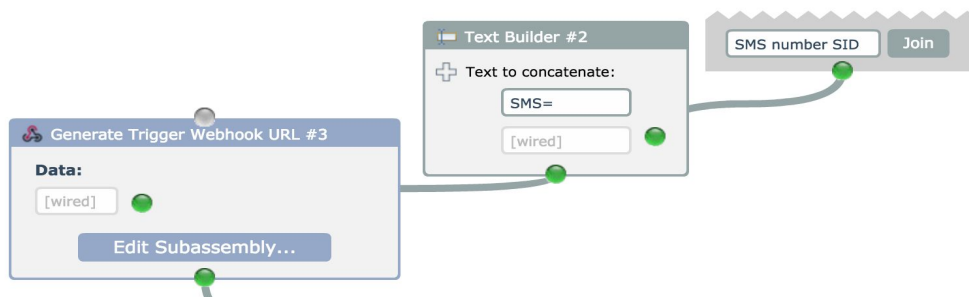
The Webhook module parses the received webhook and emits data fields in a manner that can be directly consumed by the Trigger module. The Trigger module's **unique row identifier** field should be left empty for all webhooks.

API-Registered Webhooks

Some API's provide endpoints for the registration and management of webhooks. API-Registered webhooks follow this basic pattern:

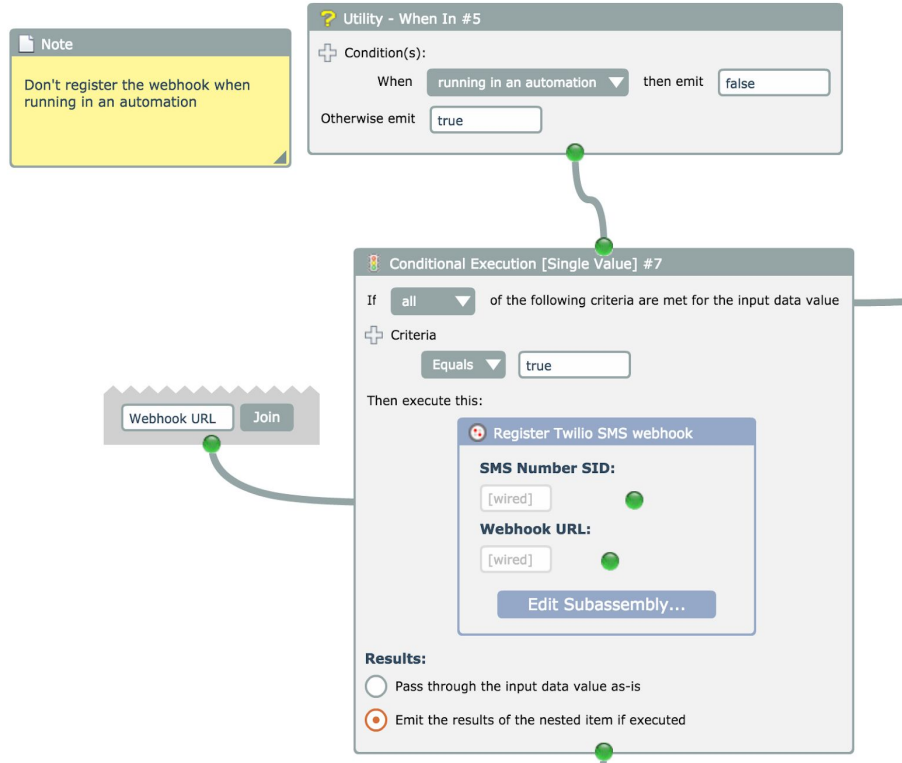
- When the webhook assembly is run in either the assembly or automation editor, the Generate Trigger Webhook URL or Generate Service Webhook URL subassemblies are used to create the webhook URL that will be fed into the Webhook module.
- When the webhook assembly is run in either the assembly or automation editor, the API endpoint to register the webhook URL with the app will be called.
- If the registered webhook is dangling because it was generated in the Assembly Editor or an automation was deleted or not completely built, the app's Delete Webhook assembly will be invoked by the system to unregister the webhook from the app.

The first step for registering a webhook URL via an API is to generate the webhook URL. This can only be done by either using the **Generate Trigger Webhook URL** or **Generate Service Webhook URL** subassemblies:



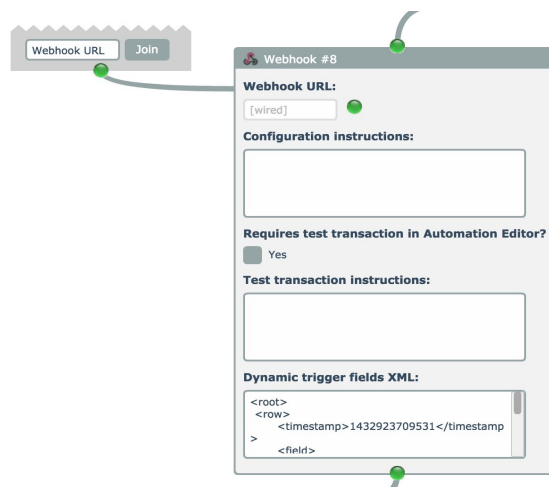
Both subassemblies accept a data parameter that can be supplied to facilitate the needs of the **Delete Webhook** assembly, to be discussed in the *Dangling Webhooks* section further below.

The next step is to conditionally register the webhook URL via the API:



The API endpoint to register the webhook URL should only be invoked when the assembly is being executed in the assembly or automation editor. The **Utility – When In** module and **Conditional** module above are used to determine when to invoke the subassembly to register the webhook URL.

Finally, the generated webhook URL is wired into the **Webhook** module's field:



The Configuration Instructions are blank because the webhook is registered automatically via the API and the end user doesn't need to perform any configuration.

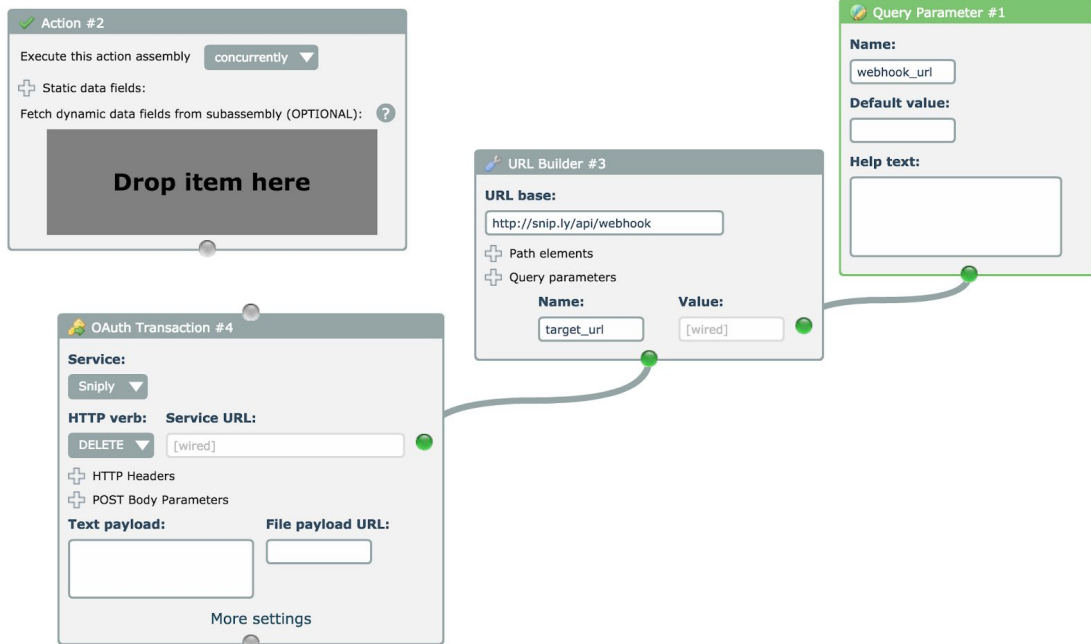
Dangling Webhooks

Webhooks that are registered via API have the risk of being left dangling. A dangling webhook is one that is still registered with an app, but no automation is available to receive and process the webhook. So the app keeps sending the webhook needlessly.

Dangling webhooks can result from the following:

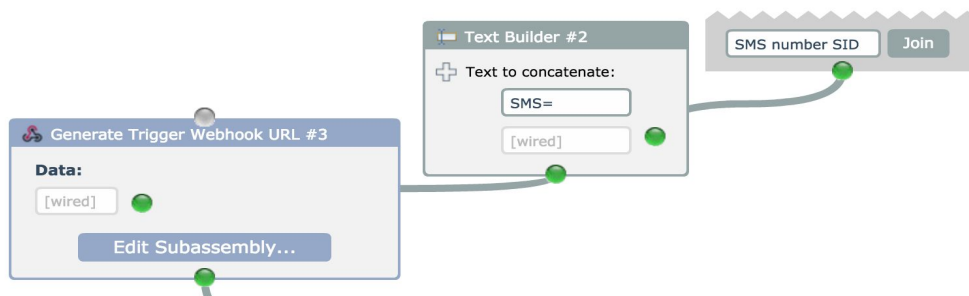
- Test webhook URLs are generated for use in the Assembly Editor to facilitate building and testing of the webhook trigger assembly
- An end user starts to build an automation with the webhook trigger but never saves the automation
- The automation is deleted

Developers of API-registered webhook triggers must build a **Delete Webhook** assembly that will unregister dangling webhooks via the app's API. The assembly must be named exactly "Delete webhook". It is a type of action assembly. It should be made private, because it is not an action to be used in the automation editor. It is an action just so it appears in the Assembly Editor's catalog underneath the app.



Because it is an action, it must have an Action module which is unused. The assembly can receive two query parameters named "webhook_url" and "data", both of which can be received via the **Query Parameter** module.

The value of "webhook_url" will be the webhook URL that was registered with the API. The value of "data" will be whatever data was supplied to either the Generate Trigger Webhook URL or Generate Service Webhook URL subassemblies:



The "data" value can be anything needed to facilitate deletion of the webhook.

The system will periodically scan the database for dangling webhooks, ones that are not associated with any automations. The system invokes the **Delete Webhook** assembly for any dangling webhooks it finds.



Note: Don't forget to save your delete webhook assemblies as PRIVATE.

Webhooks with one event type

Most apps send one event type per webhook URL. For example, a “new contact” event would be sent to webhook URL #1 and an “updated contact” event would be sent to a different webhook URL #2.

In this case, each event type maps to a separate webhook trigger assembly.

The **Generate Trigger Webhook URL** subassembly must be used when working with an app that sends one event type per webhook URL, for the case where they provide an API for registering webhooks.

The generated webhook URL will be of the form **{SERVER_URL}/webhook/automation_uid**, which is to say that each event type is processed by a single automation.

Webhooks with multiple event types per account

Some apps send multiple event types to a single webhook URL for a given customer account. For example, both “new contact” and “updated contact” and other events would all be delivered to a single webhook URL. This type of webhook is called a **Service Webhook**.

In this case, the webhook URL maps to all the automations for the user that contain any webhook trigger. The webhook triggers have to inspect the webhook payload and only emit data if the payload contains data for the given trigger. For example, a “new contact” trigger would inspect the payload and only emit data if the received event was a new contact.

The **Generate Service Webhook URL** subassembly must be used when working with an app that sends multiple event types to a single webhook URL.

The generated webhook URL will be of the form **{SERVER_URL}/webhook/app_assembly_uid-person_uid**, which is to say that the webhook is associated to the app and to the end user. When the webhook is received the URL is parsed to determine the app and user account. Then all active automation for the user with triggers for the app are executed.



Tip: The best way to understand how to go about integrating webhooks is to view existing examples already in the system. Find the Webhook module, the Generate Trigger Webhook URL subassembly, and the Generate Service Webhook URL subassembly in the catalog, right-click on them, and find assemblies where they are used to examine existing implementations.

Webhooks with multiple events and multiple accounts

Some apps may use a single webhook for sending all activity in their entire system, for all customer accounts. MINDBODY and Clover Network are two apps that do this.

Use a Unary Protocol Thread to build this type of webhook integration. See the next section.

Protocol Thread Triggers

Another form of Instant Triggers are Protocol Threads. A Protocol Thread trigger maintains an always-running thread on the server that will typically use an SDK that maintains a socket connection. Protocol Thread triggers come in two variations:

- Unary Protocol Threads, where a given app will have a single thread that handles all inbound and outbound data for all automations. It is also possible for Unary Protocol Threads to respond to payloads delivered to a single webhook URL.
- Per-Trigger Protocol Threads, where each trigger will have its own thread. If 5 active automations use the trigger, 5 threads will be running on the server.

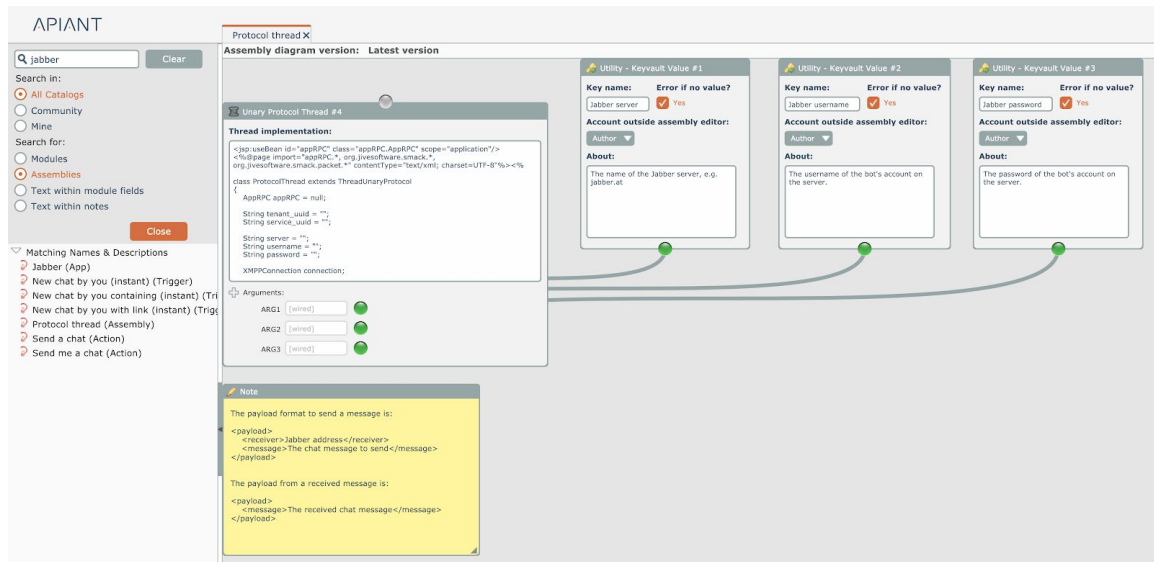
Unary Protocol Threads

Two types of Unary Protocol Threads can be built: freeform threads that send/receive data in any manner possible, and webhook listeners that receive payloads delivered to a single URL.

Freeform Protocol Threads

Jabber is an example of a freeform Unary Protocol Thread, where a SDK that wraps the XMPP protocol is used to send/receive messages to/from the server's bot account.

Search for "Jabber" in the Assembly Editor catalog and load its Protocol Thread assembly:



Account information for the Jabber bot is stored in the keyvault. The **Utility - Keyvault Value** modules load the needed credentials from the keyvault and supply the value to the Java JSP code within the **Unary Protocol Thread** module.

The bottom section of Java JSP code within the **Unary Protocol Thread** module validates that the required server address and account are not empty in the keyvault.

```

173 String xml = "DATASTREAM1";
174 String tenant_uuid = "TENANT_UUID"; // The system will replace this with the tenant's uuid
175 String service_uuid = "SERVICE_UUID"; // The system will replace this with the service's uuid
176
177 String server = "ARG1";
178 String username = "ARG2";
179 String password = "ARG3";
180
181 ProtocolThread thread = null;
182
183 try
184 {
185     if (server.length() > 0 && username.length() > 0 && password.length() > 0)
186     {
187         thread = new ProtocolThread(appRPC, tenant_uuid, service_uuid, server, username, password);
188         appRPC.registerUnaryProtocolThread(tenant_uuid, service_uuid, thread);
189         thread.start();
190         thread.join();
191     }
192     else
193     {
194         System.out.println("Configuration error: your keyvault needs to contain entries for 'Jabber server', 'Jabber username', and 'Jabber password'.");
195     }
196 }
197 catch (Exception e)
198 {
199     // Stackdump will be written to the unary protocol thread's log accessible from the Admin Console
200     System.out.println(e);
201 }
202 finally
203 {
204     if (thread != null && !thread.isInterrupted())
205     {
206         thread.interrupt();
207     }
208 }
209 appRPC.unregisterUnaryProtocolThread(tenant_uuid, service_uuid, thread);
210 }
211 }
212 }
213 %>

```

If the supplied server address and username/password are not empty then the protocol thread is started in lines 187-192 above.

The protocol thread implementation starts at the top of the code:

```

1 <jsp:useBean id="appRPC" class="appRPC.AppRPC" scope="application"/>
2 <%@page import="appRPC.*, org.jivesoftware.smack.*, org.jivesoftware.smack.packet.*" contentType="text/xml; charset=UTF-8"%><%
3
4 class ProtocolThread extends ThreadUnaryProtocol
5 {
6     AppRPC appRPC = null;
7
8     String tenant_uuid = "";
9     String service_uuid = "";
10
11     String server = "";
12     String username = "";
13     String password = "";
14
15     XMPPConnection connection;
16
17     public ProtocolThread(AppRPC appRPC, String tenant_uuid, String service_uuid, String server, String username, String password)
18     {
19         this.appRPC = appRPC;
20
21         this.tenant_uuid = tenant_uuid;
22         this.service_uuid = service_uuid;
23
24         this.server = server;
25         this.username = username;
26         this.password = password;
27     }

```

Note that this code imports the Smack Java library which handles XMPP communication for Jabber. Java libraries needed by protocol threads must be installed on the server in order to be used.

The protocol thread's run() method should have an outer infinite loop with an inner try block:

```
29 public void run()
30 {
31     while (true)
32     {
33         try
34         {
35             this.connection = new XMPPConnection(this.server);
36
37             this.connection.connect();
38             System.out.println("Connected to server " + connection.getHost());
39
40             if (this.username.contains("@"))
41             {
42                 this.username = this.username.substring(0, this.username.indexOf("@"));
43             }
44
45             this.connection.login(this.username, this.password);
46             System.out.println("Logged in as " + connection.getUser());
47
48             Presence presence = new Presence(Presence.Type.available);
49             this.connection.sendPacket(presence);
50
```

Here the code is signing into the Jabber server and setting the bot's presence to "available".

The thread's run() method is the listening for inbound messages. This section of code implements the callback receiver of messages:

```

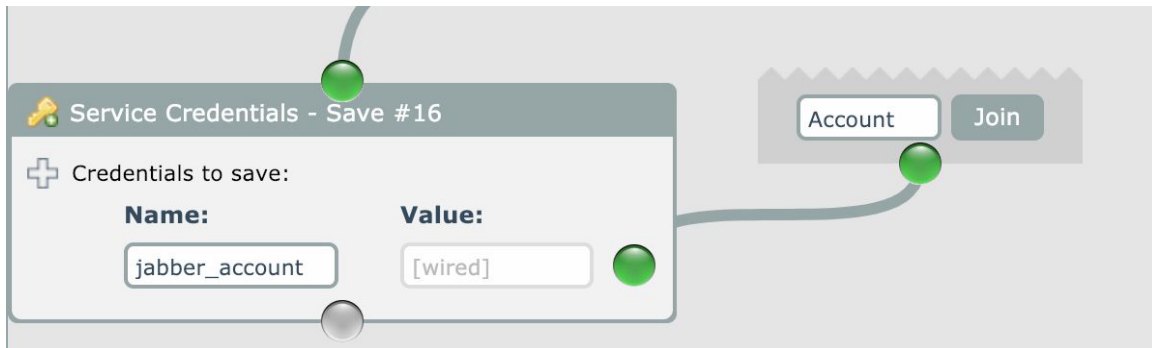
51 PacketListener pl = new PacketListener()
52 {
53     public void processPacket(Packet p)
54     {
55         try
56         {
57             if (p instanceof Message)
58             {
59                 Message msg = (Message) p;
60
61                 String received_chat = msg.getBody();
62                 if (msg.getType().equals(Message.Type.chat) && received_chat != null && received_chat.length() > 0)
63                 {
64                     String from_account = msg.getFrom().substring(0, msg.getFrom().indexOf("/"));
65
66                     System.out.println("Received message ["+received_chat+"] from ["+from_account+"]");
67
68                     VTDDocument doc = new VTDDocument("<payload/>");
69
70                     doc.getRootElement().addElement("message", received_chat);
71
72                     // "jabber_account" is used in the Service Credentials - Save module in the Jabber app assembly
73                     String result = appRPC.handleReceivedUnaryProtocolPayload(tenant_uuid, service_uuid, "jabber_account",
74                                     from_account, doc.commit().asXML());
75                     System.out.println(result);
76                 }
77             }
78         }
79         catch (Exception e)
80         {
81             appRPC.error(e);
82         }
83     }
84 };
85
86 this.connection.addPacketListener(pl, null);
87
88 while (true)
89 {
90     Thread.sleep(10000);
91 }

```

After receiving the message, **appRPC.handleReceivedUnaryProtocolPayload()** is invoked which is the heart of any Unary Protocol Thread. This method is what routes data to active automations.

The mechanism by which the system knows which automations to invoke is the 3rd and 4th parameters. In this case, "jabber_account" is used to identify connected accounts in the system.

Open the "Jabber" app assembly and find this at the bottom:



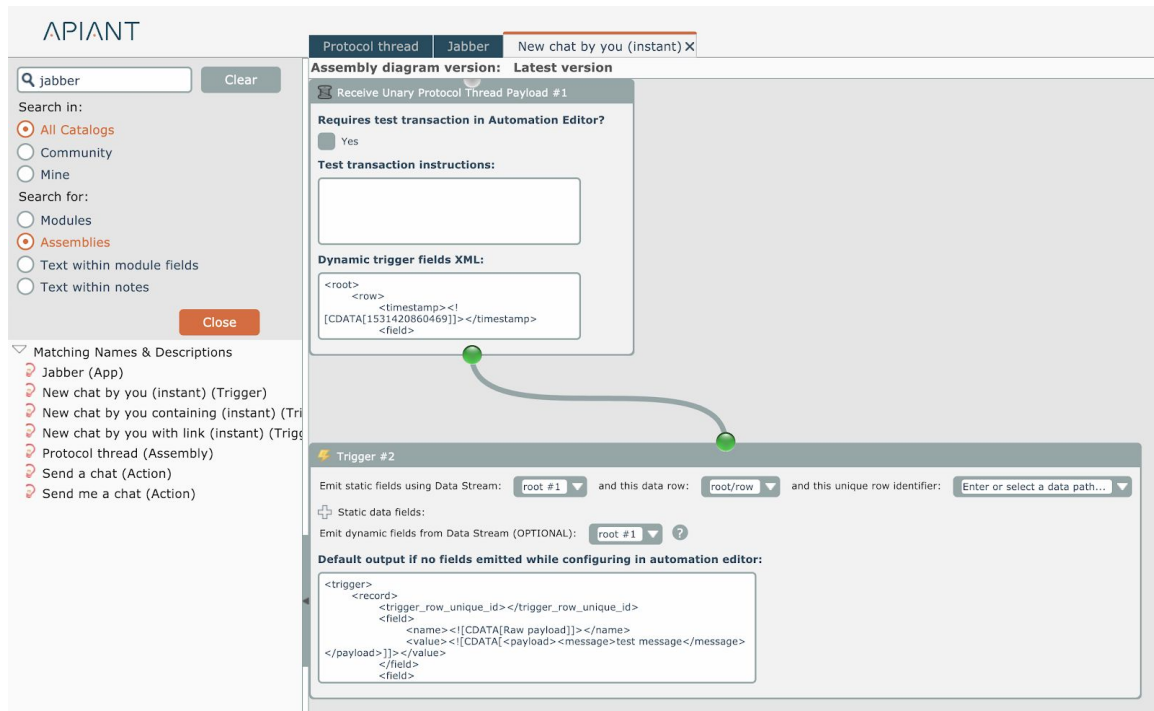
So when a user connects a Jabber account their Jabber account id (for Jabber it is like an email address) is saved in the system's database with the name "jabber_account". When the user builds automations having a Jabber trigger the account they selected will be associated with the automation. This is how

appRPC.handleReceivedUnaryProtocolPayload() knows which automations to invoke and deliver the payload.

The last parameter to

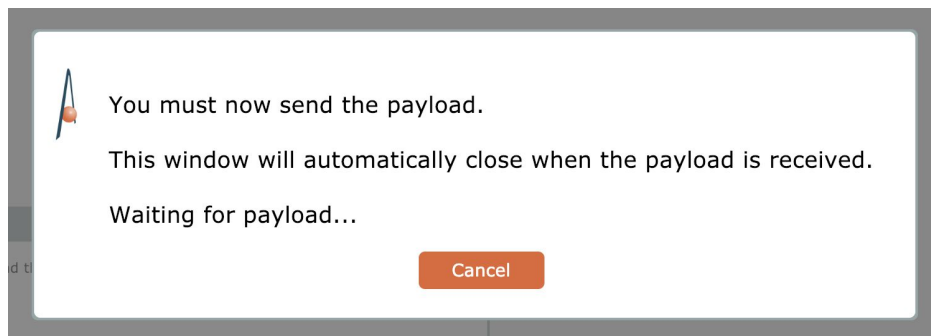
appRPC.handleReceivedUnaryProtocolPayload() must be an XML string value consisting of the payload to send. In the case of the Jabber implementation, lines 68 and 70 above create a simple XML document to use as the payload. The Jabber triggers will receive this XML payload from the protocol thread.

Search for the Jabber “new chat by you (instant)” trigger in the Assembly Editor’s catalog and open it.



The **Receive Unary Protocol Thread Payload** module receives the XML payload and extracts data fields from it.

To initially build the trigger, the “**dynamic trigger fields XML**” value at the bottom of the **Receive Unary Protocol Thread Payload** module will be empty. Run the assembly in the Assembly Editor and this message appears:



The **Receive Unary Protocol Thread Payload** module is waiting to receive a payload. You would send your bot a Jabber message. The waiting message will disappear when the payload has been received ok

and the **Receive Unary Protocol Thread Payload** module will emit the parsed data fields. That XML is then pasted into the “**dynamic trigger fields XML**” field at the bottom of the module to use as the default payload when the trigger is configured in the automation editor.



Note: Your protocol thread must be running in order for it to receive data and send a payload to the waiting module. Protocol threads are started from the Admin Console, which is described in the next section.

If your protocol thread emits payloads of varying schema, then you would leave the “**dynamic trigger fields XML**” field empty, check the “**requires test transaction in automation editor**” checkbox and provide instructions on how to send a payload when configuring the trigger.

Back to the end of the run() method in the Java JSP code:

```

91         catch (InterruptedException e)
92         {
93             // Stackdump will be written to the unary protocol thread's log accessible from the Admin Console
94             System.out.println(e);
95
96             break;
97         }
98         catch (Exception e)
99         {
100            // Stackdump will be written to the unary protocol thread's log accessible from the Admin Console
101            System.out.println(e);
102        }
103        finally
104        {
105            if (this.connection != null && this.connection.isConnected())
106            {
107                this.connection.disconnect();
108            }
109        }
110    }
111 }

```

The try block should break the outer while(true) infinite loop when an InterruptedException is received, meaning the system is asking the thread to be terminated.

All other exceptions are being logged via System.out.println().

Finally the Jabber connection is being disconnected when the thread terminates.



Tip: To troubleshoot your code, use System.out.println(). It gets written to your developer trace log, accessible via your Developer menu at the top right of the Assembly Editor.

To send data, protocol threads must implement the **sendUnaryProtocolPayload()** method:

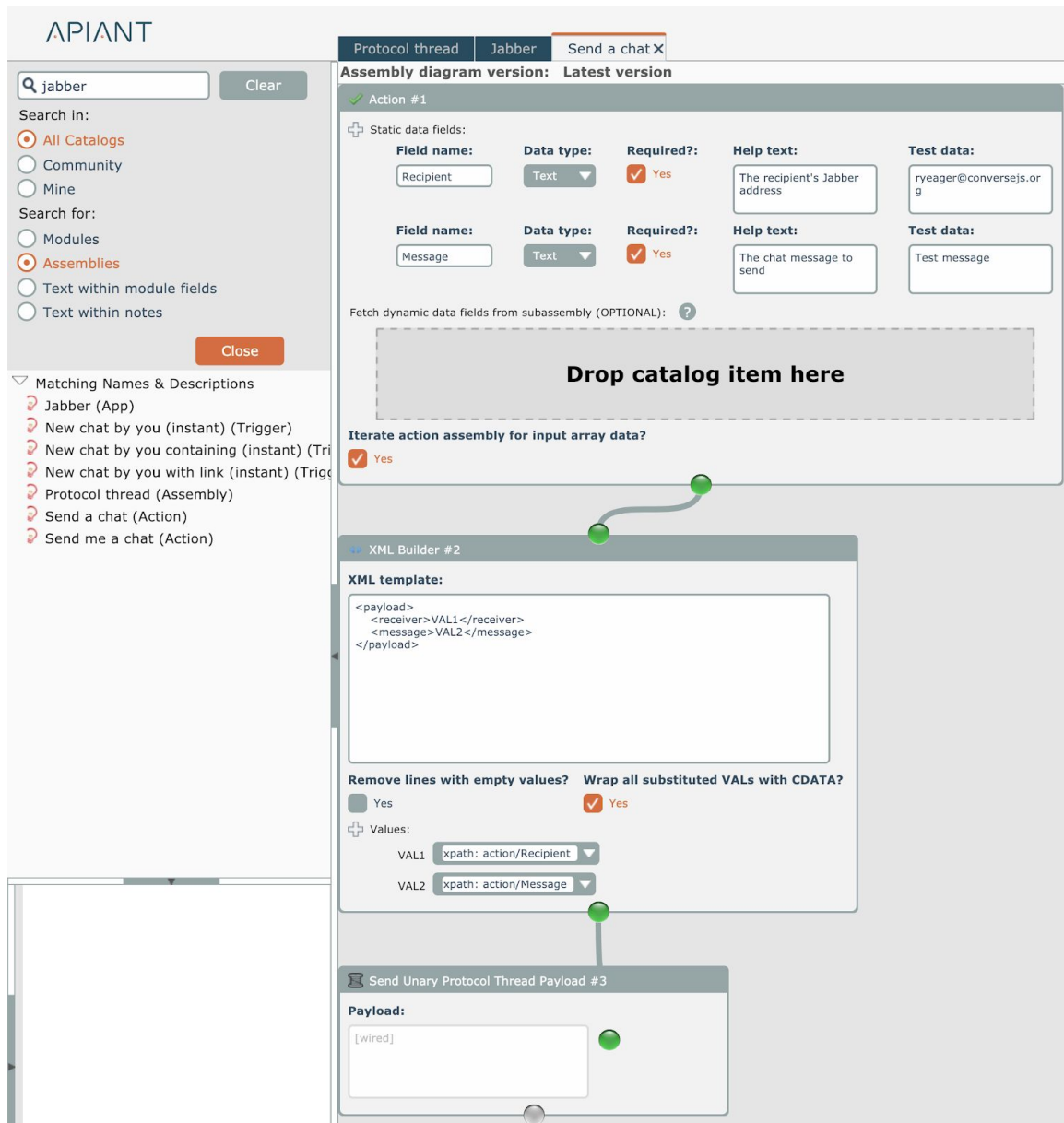
```

119 public String sendUnaryProtocolPayload(String payload)
120 {
121     String error = "";
122     try
123     {
124         VTDDocument doc = new VTDDocument(payload);
125
126         String receiver = doc.selectSingleNode("payload/receiver") == null ? "" : doc.selectSingleNode("payload/receiver").getText();
127         String message = doc.selectSingleNode("payload/message") == null ? "" : doc.selectSingleNode("payload/message").getText();
128
129         if (receiver.length() > 0 && message.length() > 0)
130         {
131             if (!this.connection.isConnected())
132             {
133                 this.connection = new XMPPConnection(this.server);
134
135                 this.connection.connect();
136                 System.out.println("Connected to server " + connection.getHost());
137
138                 if (this.username.contains("@"))
139                 {
140                     this.username = this.username.substring(0, this.username.indexOf("@"));
141                 }
142
143                 this.connection.login(this.username, this.password);
144                 System.out.println("Logged in as " + connection.getUser());
145
146                 Presence presence = new Presence(Presence.Type.available);
147                 this.connection.sendPacket(presence);
148             }
149
150             System.out.println("Sending ["+message+"] to ["+receiver+"]");
151
152             Message msg = new Message(receiver, Message.Type.chat);
153             msg.setBody(message);
154             this.connection.sendPacket(msg);
155         }
156         else if (receiver.length() == 0)
157         {
158             error = "No receiver specified.";
159         }
160     }
161     catch (Exception e)
162     {
163         // Stackdump will be written to the unary protocol thread's log accessible from the Admin Console
164         System.out.println(e);
165
166         error = appRPC.getStackTrace(e);
167     }
168
169     return error;

```

The method receives a single argument which is an XML payload. That payload comes from an action.

Search for the Jabber “send a chat” action in the Assembly Editor’s catalog and open it.



See that an XML document is constructed with the input values. This is the XML payload that is being sent to the protocol thread via the **Send Unary Protocol Thread Payload** module.

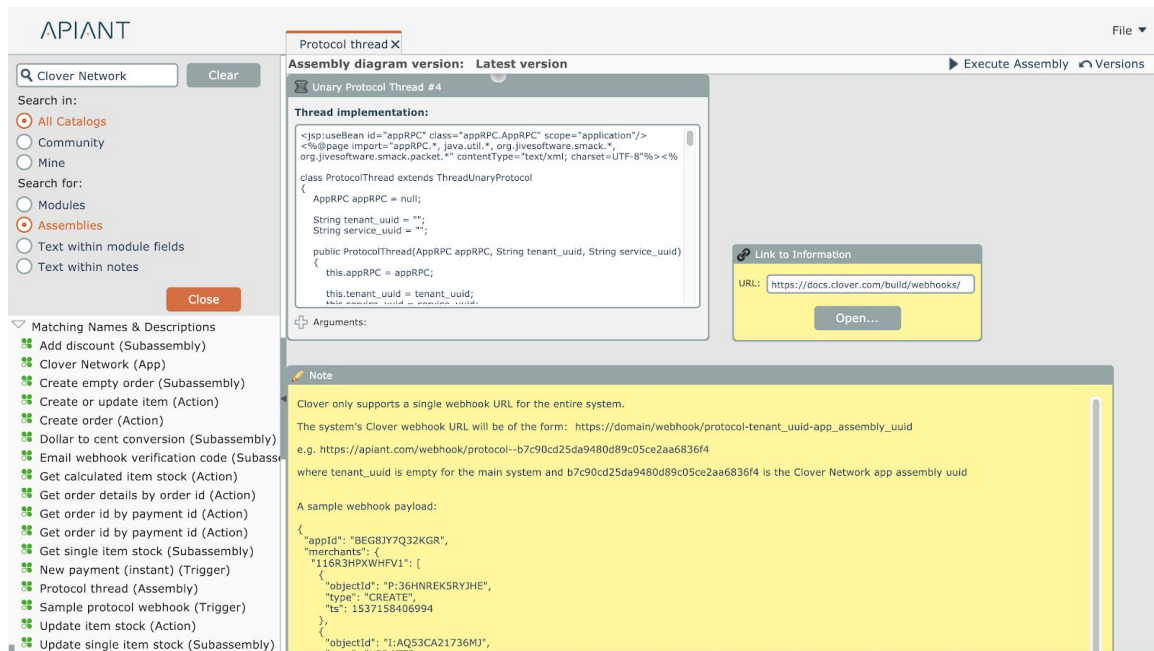
Back to the above code, observe that this XML payload is parsed and the Smack library is used to send the message via Jabber XMPP. Any errors that occur must be returned as a string error message from the **sendUnaryProtocolPayload()** method. The error message gets

logged into the automation's logs and may also be emailed to the automation's owner and system admin, depending on settings.

Webhook Listener Threads

Some apps such as MINDBODY and Clover Network use a single webhook to send all data for all customers. This type of webhook can be supported with a Unary Protocol Thread.

Search for "Clover Network" in the Assembly Editor catalog and open the "Protocol Thread" assembly:



The URL for Unary Protocol Thread webhooks has the following structure:

https://domain/webhook/protocol-tenant_uuid-app_assembly_uuid

Where "**tenant_uuid**" is the uuid of the tenant system. The tenant uuid is empty for the master system.

And where "**app_assembly_uuid**" is the uuid of the app assembly.

For Clover Network on APIANT's development server, the webhook URL is:

<https://apiant.com/webhook/protocol--b7c90cd25da9480d89c05ce2aa6836f4>

The bottom section of Java JSP code within the **Unary Protocol Thread** module just starts the protocol thread. Webhook protocol threads generally don't need any other data.

```

94 String xml = "DATASTREAM1";
95 String tenant_uuid = "TENANT_UUID"; // The system will replace this with the tenant's uuid
96 String service_uuid = "SERVICE_UUID"; // The system will replace this with the service's uuid
97
98 ProtocolThread thread = null;
99
100 try
101 {
102     thread = new ProtocolThread(appRPC, tenant_uuid, service_uuid);
103
104     appRPC.registerUnaryProtocolThread(tenant_uuid, service_uuid, thread);
105
106     thread.start();
107     thread.join();
108 }
109 catch (Exception e)
110 {
111     // Stackdump will be written to the unary protocol thread's log accessible from the Admin Console
112     System.out.println(e);
113 }
114 finally
115 {
116     if (thread != null && !thread.interrupted())
117     {
118         thread.interrupt();
119     }
120
121     appRPC.unregisterUnaryProtocolThread(tenant_uuid, service_uuid, thread);
122 }

```

The thread's **run()** method needs to just keep the thread running:

```

18     public void run()
19     {
20         // Not used, but needs to do this to keep the thread always running
21         try
22         {
23             while (true)
24             {
25                 Thread.sleep(10000);
26             }
27         }
28         catch (Exception e)
29         {
30         }
31     }

```

When the system receives payloads to the webhook URL, the system invokes the **handleReceivedWebhook()** method:

```

33     public String handleReceivedWebhook(String queryParameters, String payload)
34     {
35         // String sample_payload_json = "{\"appId\":\"BEG8JY7Q32KGR\",\"merchants\"
36         // String sample_payload_xml = "<o><appId>BEG8JY7Q32KGR</appId><merchants><
37
38         try
39         {
40             // Convert Clover JSON payload to XML
41             String payload_xml = appRPC.json_to_xml(payload);

```

The received payload depends on the sending system and may be JSON, XML, or some other format. In this case for Clover Network, JSON is received and the **appRPC.json_to_xml()** method is used to convert it to XML.

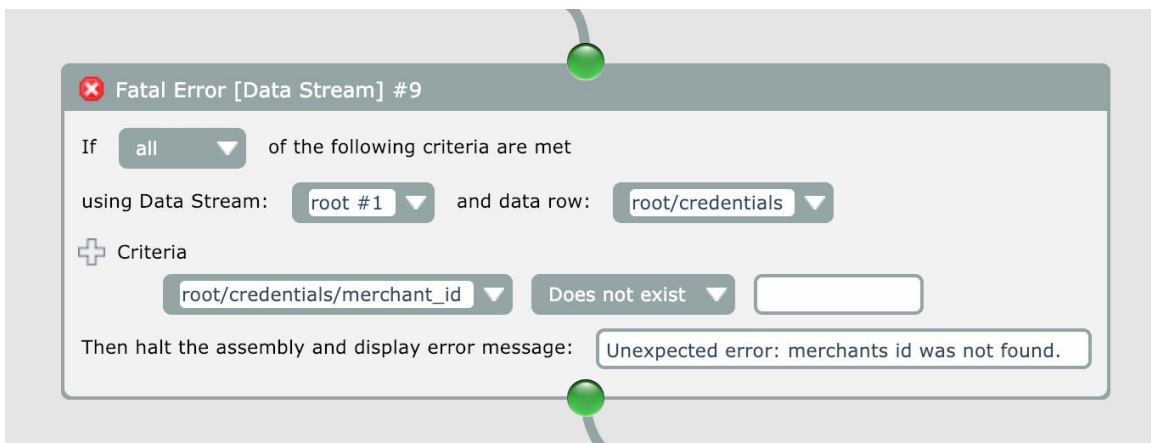
As with a non-webhook Unary Protocol Thread, the **appRPC.handleReceivedUnaryProtocolPayload()** method is used to send an XML payload to all active automations:

```

56         VTDDocument doc_out = new VTDDocument("<root/>");
57
58         List<VTDElement> listTransactions = elMerchant.selectNodes("*");
59         for (VTDElement elTransaction : listTransactions)
60         {
61             doc_out.getRootElement().add(elTransaction);
62         }
63
64         String xml_out = doc_out.commit().asXML();
65
66         String result = appRPC.handleReceivedUnaryProtocolPayload(this.tenant_uuid, this.service_uuid,
67                                                                 "merchant_id", merchant_id, xml_out);
68
69         if (result.length() == 0)
70         {
71             System.out.println("Sent payload to automations for merchant ["+merchant_id+"], payload = "+xml_out);
72         }
73         else
74         {
75             System.out.println("Error ["+result+"] sending payload to automations for merchant ["+merchant_id+"],
76
77         }
78     }

```

It is imperative that the webhook payload contain a piece of data that uniquely identifies the connected account. For Clover Network, their webhook payload contains a merchant ID. The Clover Network app assembly obtains the merchant ID value during the OAuth workflow and checks to make sure the value exists when validating the connected account:



Webhook trigger assemblies are built in the same manner as described for freeform Unary Protocol Thread triggers, using the **Receive Unary Protocol Thread Payload** module. Usually the triggers will have to

do filtering so that the triggers only process payloads of a specific type.

See the previous section on how to use the **Receive Unary Protocol Thread Payload** module to build webhook trigger assemblies.

Starting/Stopping Unary Protocol Threads

When you run a Unary Protocol Thread assembly in the Assembly Editor, it doesn't keep the thread running. It just runs for a second and then terminates, just as a way for the thread developer to perform a compilation check of the Java JSP code.

Unary Protocol Threads are started/stopped from the Admin Console's **Protocol Threads** screen:

The screenshot displays the 'Admin Console [Master]' interface for 'APIANT'. On the left is a sidebar with navigation items: Activity, Automations, Batch Jobs, Web Services, Protocol Threads, Native Work Queue, Memory Usage, Tenants, User Accounts, User Roles, App Catalog, Catalog Categories, System Compile, System Export, System Keyvault, System Settings, and System Upgrade. The main content area is titled 'Protocol Threads' and contains three sections:

- Active Protocol Threads:** None active. A 'View Log...' button is on the right. Total active protocol threads: 0.
- Active Unary Protocol Threads:** Clover Network Protocol thread, Jabber Protocol thread, MINDBODY Protocol thread. 'View Log...' and 'Deactivate' buttons are on the right. Total active unary protocol threads: 3.
- Inactive Unary Protocol Threads:** None. An 'Activate' button is on the right. Total inactive unary protocol threads: 0.

A 'Refresh All' button is located at the bottom center of the main content area.

Select an inactive Unary Protocol Thread from the list at the bottom and click the **Activate** button to start a thread. Once activated, threads will be automatically restarted whenever the system boots.

You can view a protocol thread's log by selecting it and clicking the **View Log** button. The Java JSP code can write to the log with `System.out.println()`.

Stop an active thread by selecting it and clicking the **Deactivate** button. Once deactivated, the thread will not be restarted whenever the system boots.

Per-Trigger Protocol Threads

Per-trigger Protocol Threads are suitable when individual automations need to keep a connection to a source of data. An example is a message queue. They are built using the **Trigger - Protocol Thread** module in a trigger assembly:

Trigger - Protocol Thread

Requires test transaction in Automation Editor?

Yes

Test transaction instructions:

You must now go to APP NAME and PERFORM THIS ACTION.

Dynamic trigger fields XML:

Thread implementation:

```
<jsp:useBean id="appRPC" class="appRPC.AppRPC"
scope="application"/><%@page import="appRPC.*"
contentType="text/xml; charset=UTF-8"%><%
class ProtocolThread extends ThreadProtocol
{
    AppRPC appRPC = null;

    String tenant uuid = "";
```

Arguments:

This module can be thought of as a combination of the **Webhook** module and the **Unary Protocol Thread** module. For each active automation, the system will execute the Java JSP thread on the server. So if 5 automations are active using protocol thread triggers, 5 threads will be running.

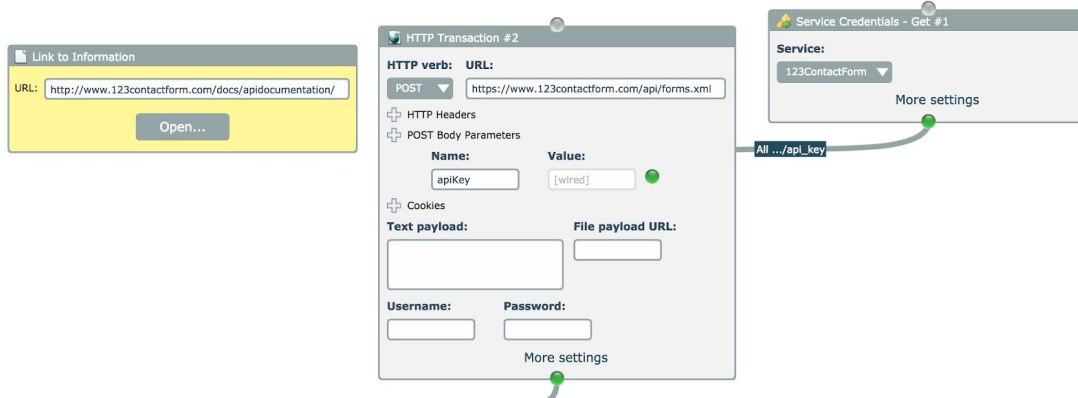
The Java JSP code implementation is essentially the same as a Unary Protocol Thread. See the previous section on Unary Protocol Thread development for information on developing the Java JSP code.

Polling Triggers


Polling triggers periodically check an API for new or updated data, for example every 15 minutes. The polling schedule and frequency is configurable by end users in the Automation Editor's dashboard.

Polling triggers should only be used when equivalent webhooks are not made available by the API provider. Polling triggers are less efficient, consume more system resources (as well as for the API itself), and can potentially miss new/updated data when many data rows are added/updated in the API.

Polling triggers begin by fetching a list of data from the API:



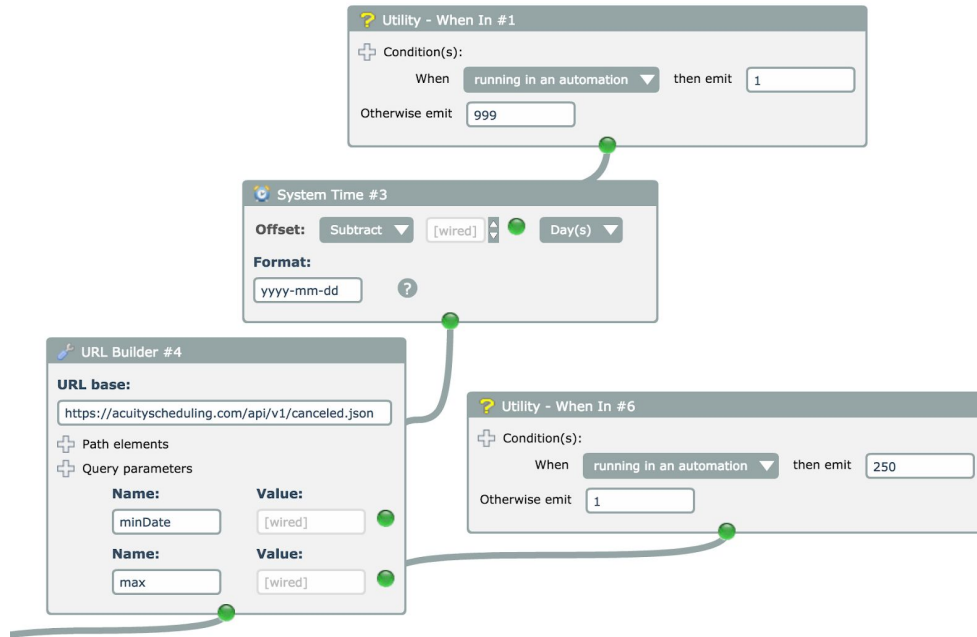
The example above is a polling trigger for new forms. Polling triggers that monitor for new data must receive the data sorted by newest items first. Polling triggers that monitor for updated data must receive the data sorted by most recently updated first.

	<p>Warning! Polling triggers will not operate correctly if the data is not sorted appropriately from the API.</p>
---	--

Most API's return a fixed amount of data rows. Polling trigger developers need to be aware that if more items are added/updated in the app than are retrieved from the API, those items will be missed and never processed by the automation. For example, if a polling trigger fetches 100 data rows from the API and the automation runs every 15 minutes, then if the end user adds or updates 150 items in the app within 15 minutes 50 items will not be processed.

Running the automation at a faster polling speed can be one way to reduce the possibility of missed items. Retrieving more data rows can be another.

If an API supports fetching new or update items that have been added or updated since a given timestamp, it is usually best to take advantage of that as a way to avoid possibly missed data rows:

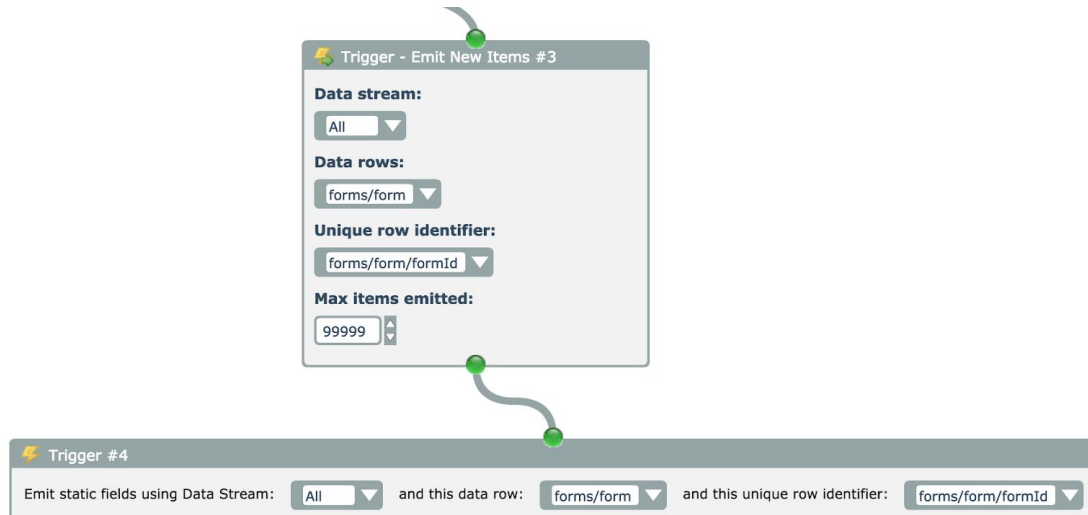


In this example the API has a **minDate** parameter that is being configured to fetch canceled appointments. The **Utility – When In** modules are being used to configure different values depending on whether the assembly is being run in automation rather than in the assembly or automation editors.

When running in an automation it is desirable to fetch a manageable amount of data rows such that the automation won't hit its timeout limits. In the above example up to 250 items are fetched for the previous 24 hours.

When running in the automation editor and assembly editor, just a single data row is needed. If the Trigger module is configured to emit dynamic data fields like is done for the above example, then the trigger must try to fetch at least one data row so its data fields can be parsed and emitted by the Trigger module to facilitate data field mapping in the automation editor. So the example above searches for one data row up to 999 days old, just to make an attempt to find at least one data row to facilitate the parsing of dynamic data fields.

After fetching data, polling triggers should then use the **Trigger – Emit New Items** module to determine which data rows are new or updated:

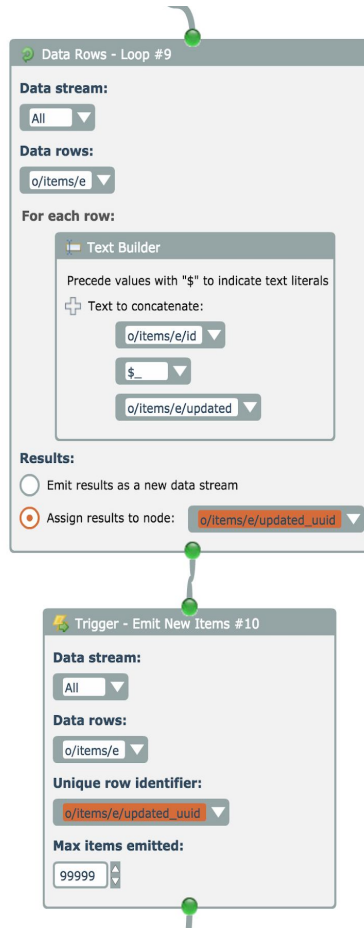


The example above is for a new form trigger. The Trigger – Emit New Items module is configured to examine all form ids. The module stores the form ids in the database. Each time the trigger is executed the ids stored in the database are compared against the form ids received from the API. Only data rows with form ids not currently in the database are emitted, representing new forms.




Note: Note that the selected unique row identifier must be the same in the Trigger – Emit New Items module and the Trigger module. If they aren't the same the Assembly Editor will display a warning when the assembly is saved.

Triggers for updated data need logic to create a unique identifier that is a combination of the item id plus its last updated timestamp for use with the Trigger – Emit New Items module. In this manner, the Trigger – Emit New Items module will know when an item has been updated when its last updated timestamp changes:



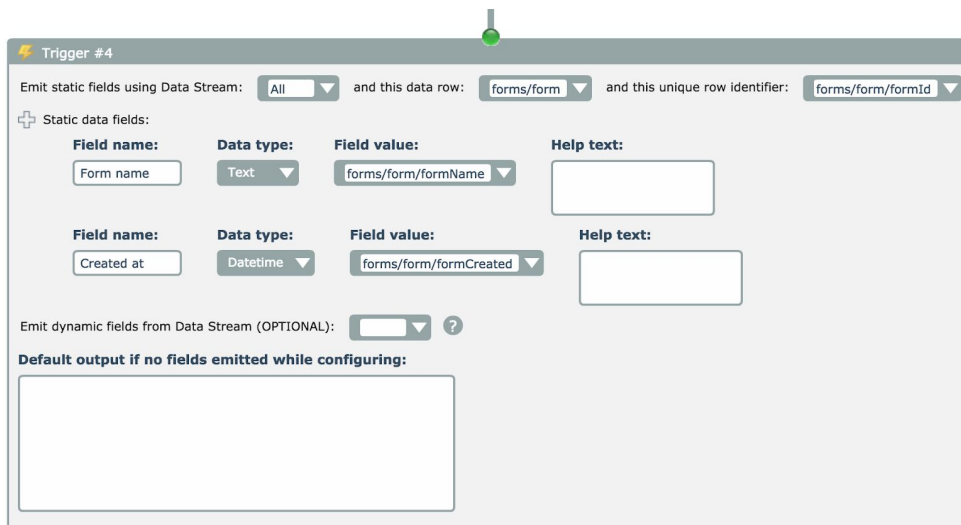
The above example is the general pattern to use to build unique identifiers for updated data items. Use a Loop module with a nested Text Builder to concatenate the item ids with the last updated timestamp, then configure the Trigger – Emit New Items module to use the generated update identifier.

Don't forget that the Trigger module also needs to use the generated update identifier, too.



Tip: Any processing that a polling trigger needs to perform on data rows, like transforming data or making additional API calls to fetch additional details for each data row, should be performed AFTER the Trigger – Emit New Items module, since it emits only the data rows being emitted by the trigger. It would be wasteful processing to perform functionality on data rows that later get discarded and not emitted by the trigger.

All polling trigger assemblies must have a Trigger module at the end:



Trigger #4

Emit static fields using Data Stream: and this data row: and this unique row identifier:

Static data fields:

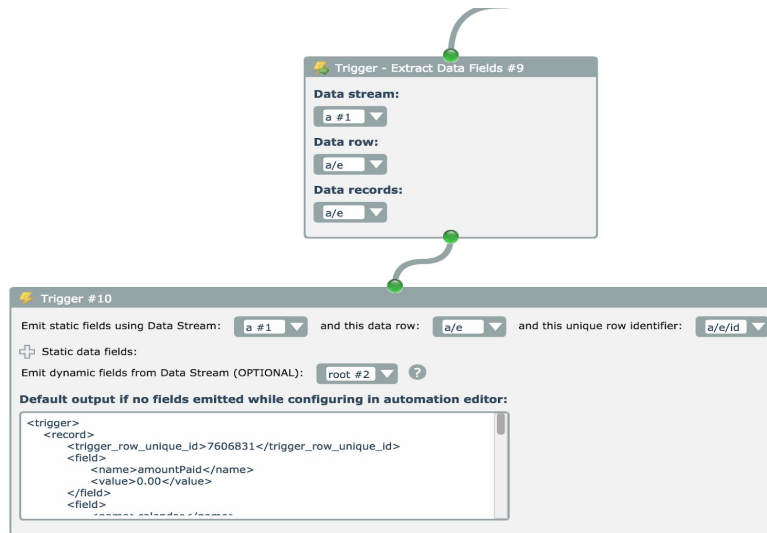
Field name:	Data type:	Field value:	Help text:
<input type="text" value="Form name"/>	<input type="text" value="Text"/>	<input type="text" value="forms/form/formName"/>	<input type="text"/>
<input type="text" value="Created at"/>	<input type="text" value="Datetime"/>	<input type="text" value="forms/form/formCreated"/>	<input type="text"/>

Emit dynamic fields from Data Stream (OPTIONAL): ?

Default output if no fields emitted while configuring:

The Trigger module emits data fields for each data row that can be mapped to actions in automations. The example above shows static data fields being keyed in manually.

Data fields can also be parsed and extracted dynamically from the data:



The **Trigger - Extract Data Fields** module is able to parse out all data fields from most API's. Its output data stream can then be selected in the Trigger module for emitting the dynamically parsed data fields.

When a trigger emitting dynamic data fields is configured in the Automation Editor, the trigger is executed in order to fetch a record and parse its fields. The trigger's API call should use the **Utility - When In** module to configure the API parameters to try to fetch a single record, so end users don't have to wait any longer than needed when configuring the automation. The Trigger module can be configured with default data fields if no item can be fetched from the API when configuring the trigger in the automation editor.

To configure default data fields, run the trigger in the Assembly Editor so that at least one trigger record is emitted. Open the Stream Inspector for the Trigger module, copy its output, and paste it into the bottom of the Trigger module.

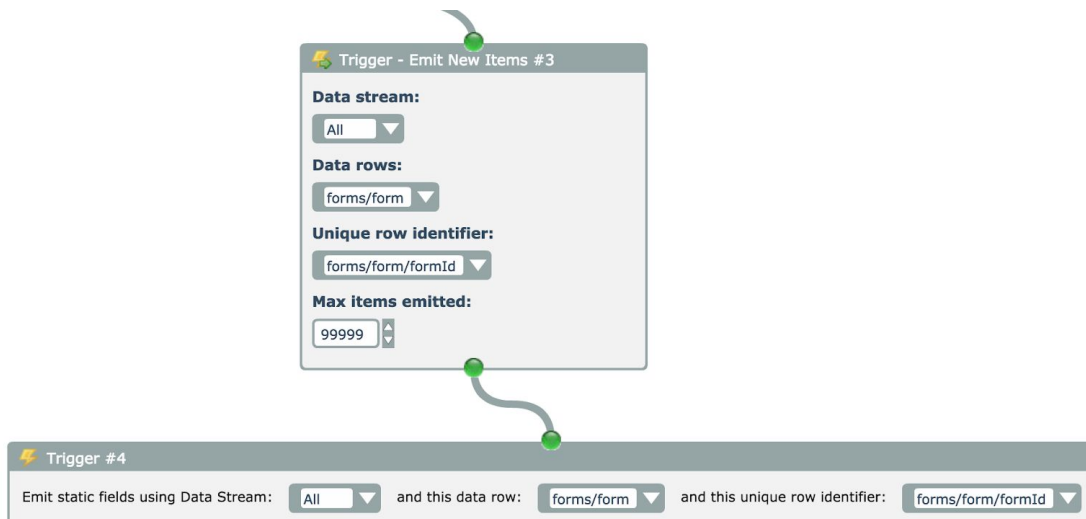


Note: Default output fields should only be defined for triggers that always emit the same fields, e.g. not for triggers that emit custom fields.

Data Row Identifier Storage

Polling triggers store data row identifiers in the database in order to avoid reprocessing records that have already been processed.

The Trigger - Emit New Items module reads row ids from the database. Each data row having an id already in the database is filtered from its input data stream. Only data rows with ids not in the database are emitted:



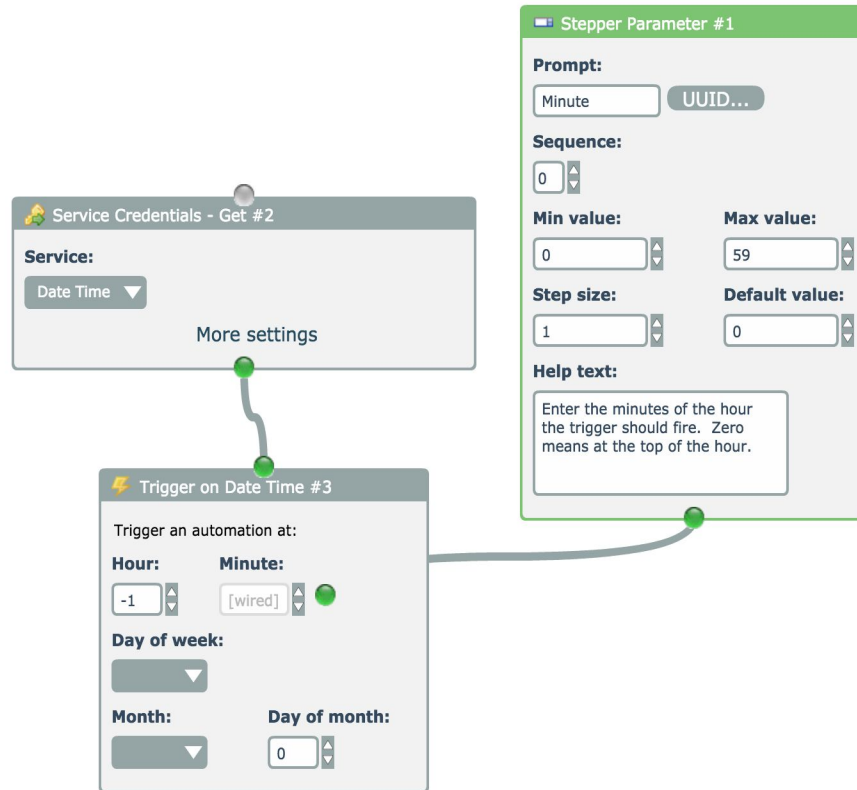
Data row ids are stored in the database by the automation execution engine when the first action completes its execution, regardless if the action succeeded or failed.

Consider if the trigger emits 5 data rows to be processed. The automation execution engine processes each data row serially, one after another. As the first data row begins to process, after the first action completes the data row's id is stored in the database. As each action completes its execution the state of the engine is updated in the database so that if a failure occurs a retry can be performed from the point of failure.

Now consider if something interrupts the system while the second data row's actions are being processed. The data row ids for the remaining third, fourth, and fifth data rows are not yet stored in the database. So the next time the polling trigger is executed, they can be emitted by the Trigger - Emit New Items module, assuming they are present in the API's response of fetched data.

Date/Time Triggers

Date/Time triggers fire at a certain date and time, or on a periodic time schedule like once an hour. The **Trigger on Date Time** module is used to define the schedule:

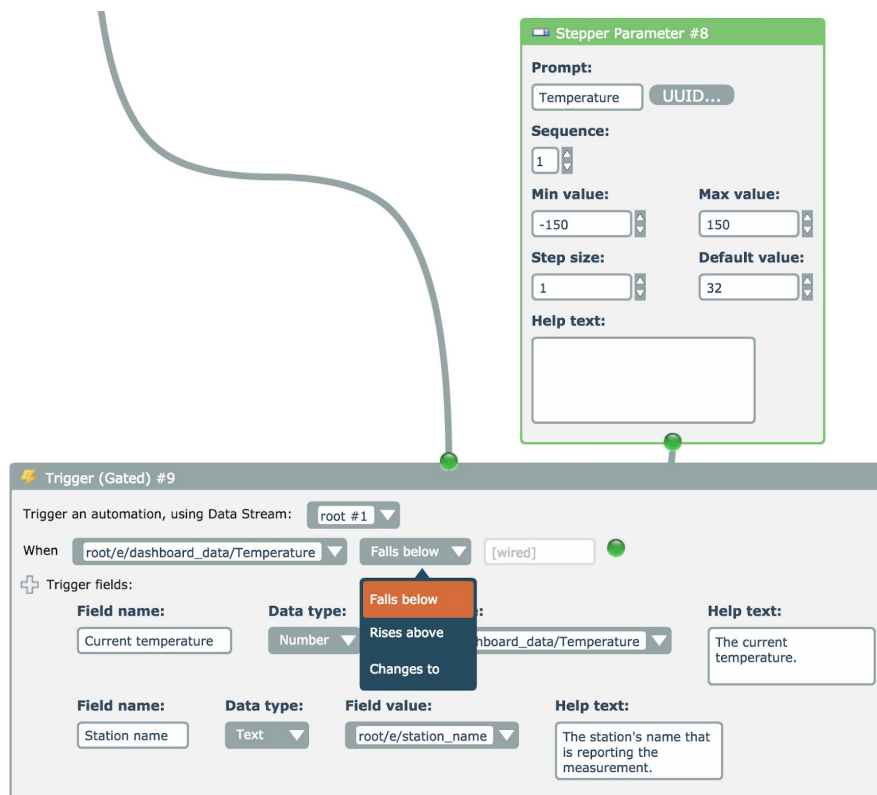


Typically Parameter modules are used to allow the end user to configure the schedule, as shown.

Gated Triggers

Gated triggers fire only once each time a threshold criteria is met. An example is a weather trigger that fires when a temperature goes below a certain value. The trigger fires once when the temperature falls below the threshold, but doesn't fire again until the temperature rises above the threshold and then falls below the threshold again.

Gated triggers simple need to make an API call to fetch the data, then configure the threshold criteria and the event using the **Trigger (Gated)** module:



Typically a Parameter module is used to allow the end user to configure the threshold value, as shown.

Export Triggers

An export trigger is one that can be used for one-time bulk exports of data from one system, typically so it can be loaded into another system.

Export triggers are denoted by using "Export" as the first word in the trigger's name.

Export triggers are constructed in a manner where all API data is fetched, typically via pagination. Export triggers do not use the Trigger – Emit New Items module.

The screenshot displays the APIANT interface for configuring an export trigger. The main workspace shows an assembly diagram with the following components:

- URL:** `https://developers.google.com/gmail/api/v1/reference/users/messages/list`
- Server script:**

```

<?php
$oauth_access_token = ARG1;
$pageId = ARG2;
$pageToken = NULL;

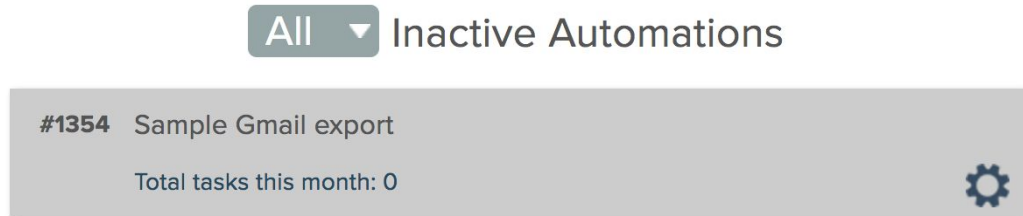
try {
    require_once 'google-api-php-client-master/autoload.php';
    $xml = new SimpleXMLElement("<root/>");

```
- Service Credentials - Get #2:** Configured for the 'Gmail' service.
- Dropdown Parameter (Populated) #1:** Prompt: 'Label', Sequence: 'UID...', Editable?: Yes.
- Trigger #5:**
 - Static data fields configuration:

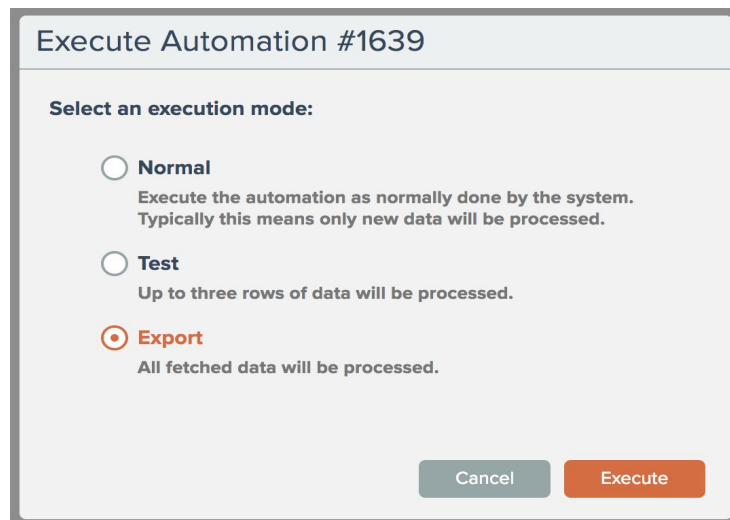
Field name	Data type	Field value	Help text
From	Text	root/msg/from	The sender of the email.
Subject	Text	root/msg/subject	The email's subject.
Body Summary	Text	root/msg/body_summary	A summary of the email's body.

On the left, a sidebar lists various triggers, with 'Export all email addresses labeled' selected. A note at the bottom left states: "Exports all email addresses with the specified label. NOTE: this trigger is intended to only be used for one-time exports by running the automation manually in 'Import Mode' from the dashboard."

Export automations that have an export trigger appear differently in the automation editor's dashboard. They are never in an active state and can only be executed manually:



They can only be executed in export mode:




Executing the automation in export mode results in any timeout settings to be ignored. The automation will run until it processes all data, or until manually halted.

	<p>Warning! Automations process all data in-memory. It is possible for an export automation to be halted by the system if it consumes too much memory, depending upon how much data is processed and the system's available memory.</p>
--	--

Action Assemblies

All Action assemblies begin with an **Action** module that defines input data fields for the action to process:

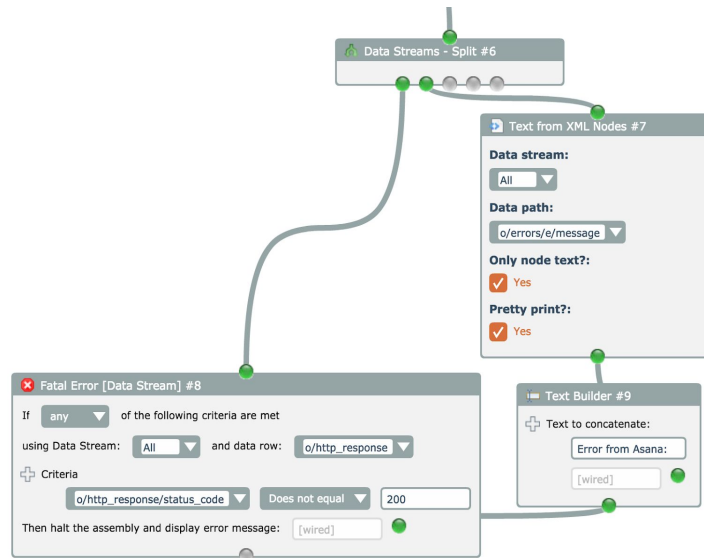
Static input data fields can be typed into the module and configured manually. Dynamic data fields can be loaded from a nested subassembly. As an example of when dynamic fields must be loaded, consider a spreadsheet action to write a new row. The action doesn't know which columns are available until the sheet is inspected via the API and its columns are determined.

	<p>Tip: Rather than keying in and configuring a lot of static data fields by hand, you can create a subassembly and use the Data Streams – Create From Text module to configure the fields from static XML data, which can be easier to create.</p>
---	---

The outputs from the Action module are then typically formatted for sending to the API. The **URL Builder**, **Text Builder**, **JSON Builder** and **XML Builder** modules can be helpful for formatting data. Complicated data formatting usually requires custom coding via the **Extension – Server-Side Script** module.

Once data is formatted, the API is invoked via the HTTP or OAuth Transaction modules, or one of their variants.

After invoking an API call, the action should check for any error conditions and return an error if the API call was unsuccessful:

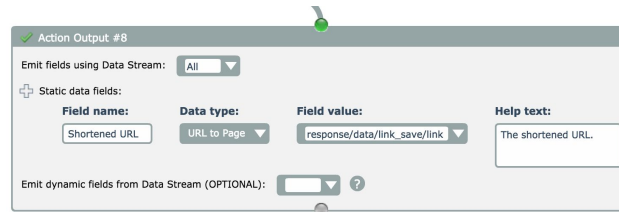


Warning! Actions must catch and throw all error conditions in order for the transaction to be flagged as an error in the automation’s history and for alert processing to be performed.

The proper way to do error checking in an Action is to test for the absence of a successful result from an API call. For example, if you are adding a new contact to a CRM then check the API’s response to ensure that the new contact ID is returned.

Some actions need to indicate to the system that the trigger data row should be skipped rather than retried. For example, an action that checks for existing duplicate data needs to skip the trigger data row if a duplicate exists. The **Action - Skip Data Row** module can be used within a **Conditional** module for this.

Finally, actions can emit output data fields via the **Action Output** module.



The output data fields can be used as inputs to subsequent actions in multi-action automations.

Find Actions

A Find Action is one that finds a single item and returns its information using the Action Output module. An example is a CRM “find contact” action that looks up a contact by email address.

The best example of a general-purpose find action is the Salesforce “find object record” action. When configuring the action in the automation editor this action emits all available output fields without performing a search, by fetching the object’s schema of available fields.



Warning! If an Action Output module emits dynamic fields, the action assembly is executed in the Automation Editor when configuring the automation in order to determine the mappable fields. The action must not perform any API call that creates data!

Use the Utility - When In module and Conditional modules to control processing done when configuring the action in the automation editor vs. its behavior when running in an automation.

Two-Way Sync

Two-way sync (or bidirectional sync) is when changes are mirrored between two systems. Traditional triggers and actions cannot accomplish two-way sync because they would result in an infinite loop. For example, when a new item is added in System A, automation #1 would then add the item in System B. But then an automation #2 that monitors System B for new items would be triggered and the same item would be added to System A, which again triggers automation #1 watching System A for new items.

The system makes it possible to build two-way sync automations that avoid infinite loops:



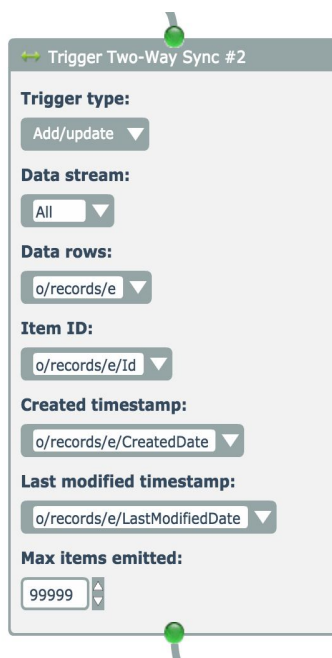
Each action maps data fields from the other app's trigger. When needed, the Transform Text app's **Lookup value in key-value table** action can be used to map values from app to app.

The system's two-way sync implementation currently has these limitations:

- Only "flat" records can be synced currently. Arrays of data associated with the main records won't be synced. So in the case of billing orders, line item changes won't be synced. It is possible to initially create the order line items in the other system. It's just that any subsequent changes to the line items can't be mirrored.
- Only items added to either system after the automation is built will be synced.
- If a synced record in both systems is updated at the same time when the automation runs, only one change is made. There is no merging of conflicts when a synced record is modified in both systems before the automation runs.

Two-Way Sync Triggers

Two-way sync triggers must use the **Trigger Two-Way Sync** module instead of the Trigger – Emit New Items module:

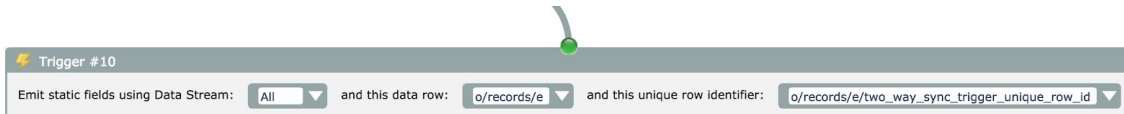


The image shows a configuration window for a 'Trigger Two-Way Sync #2' module. The window has a title bar with a double-headed arrow icon and the text 'Trigger Two-Way Sync #2'. Below the title bar, there are several sections with dropdown menus and a text input field:


- Trigger type:** A dropdown menu with 'Add/update' selected.
- Data stream:** A dropdown menu with 'All' selected.
- Data rows:** A dropdown menu with 'o/records/e' selected.
- Item ID:** A dropdown menu with 'o/records/e/Id' selected.
- Created timestamp:** A dropdown menu with 'o/records/e/CreateDate' selected.
- Last modified timestamp:** A dropdown menu with 'o/records/e/LastModifiedDate' selected.
- Max items emitted:** A text input field containing the number '99999'.

This module can be used in both polling triggers and in webhook triggers. Be sure to set the **Trigger Type** at the top of the module appropriately.

The Trigger Two-Way Sync module adds a node to each emitted data time named **two_way_sync_trigger_unique_row_id**. This data node must be selected as the unique row identifier in the Trigger module:

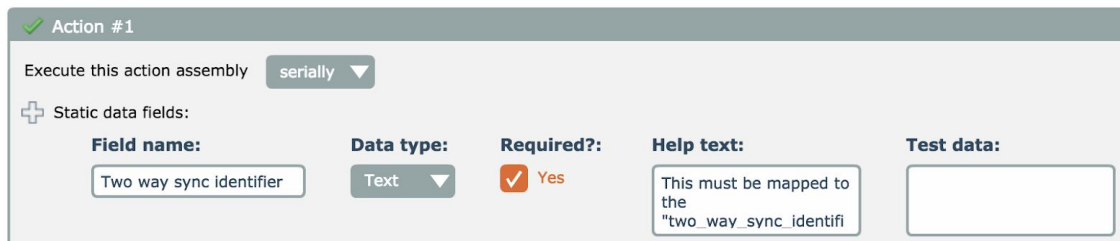


The Trigger module will emit a field named **two_way_sync_identifier** that must be mapped in the automation editor to the field of the same name in the action's field mappings.

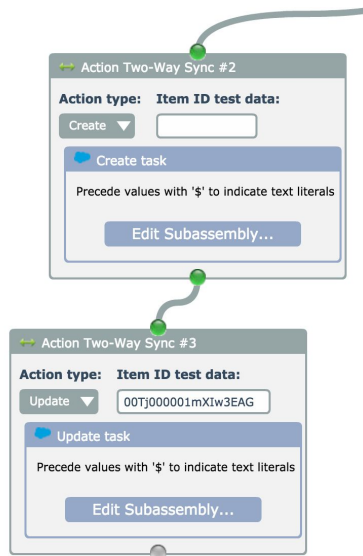
	<p>Note: Two-way sync triggers must have a name starting with "Two-way sync". The system currently uses the trigger name to determine which triggers are two-way sync.</p>
---	---

Two-Way Sync Actions

Two-way sync actions must define a field named **Two way sync identifier** as their first static data field. The field must be **required**. It is best to copy its help text from another two-way sync action assembly in the system.



Two-way sync actions must use the **Action Two-Way Sync** module with nested subassemblies to perform the configured actions:



The **Action type** selection determines what action the subassembly should perform. The Action Two-Way Sync modules must be chained together in the order of Create => Update => Delete.

The Trigger Two-Way Sync and Action Two-Way Sync modules both store item identifiers and timestamps in the system's database that are used to determine which action type needs to be performed.



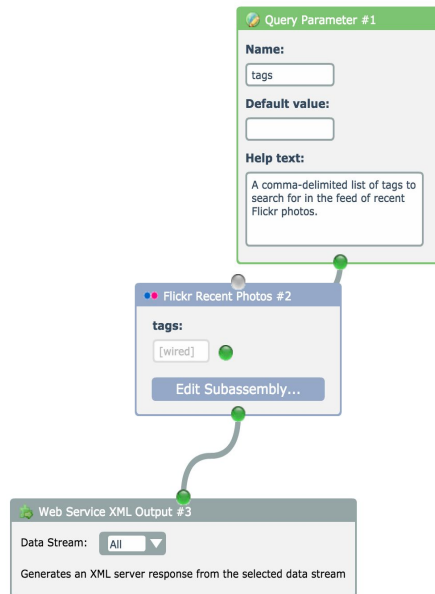
Note: Two-way sync actions must have a name starting with "Two-way sync". The system currently uses the action name to determine which actions are two-way sync.

Chapter 6: Other Assembly Types

Web Services

Web Services are assemblies that can be invoked via a URL. Web services make it possible to build API endpoints in the form of assembly diagrams that can be invoked and return data to other systems and programming tools.

All web service assemblies must have a Web Service Output module that emits data:

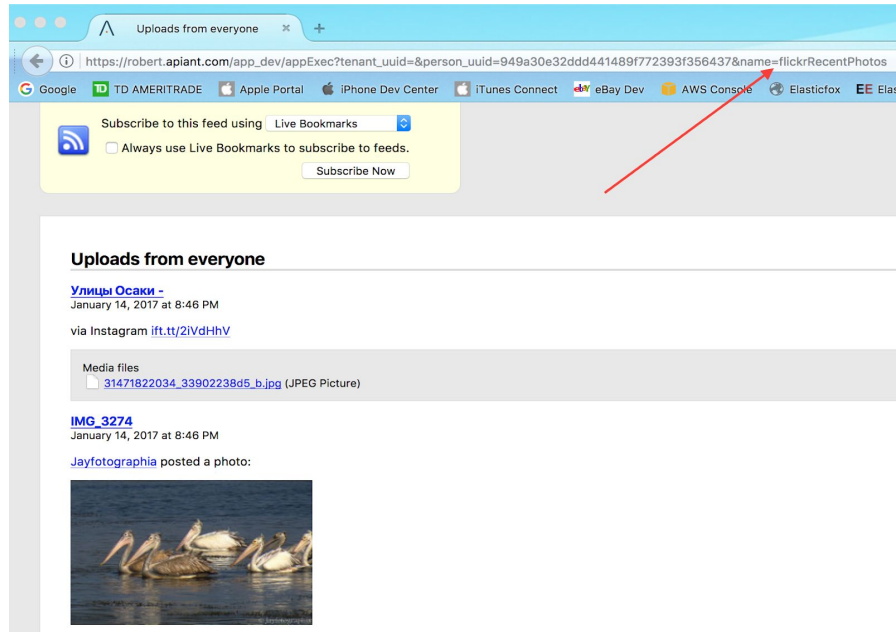


Three versions of the Web Service Output module are currently available: XML, JSON, and CSV.

Input into web services can be received as either one or more query parameters on the URL that are received by **Query Parameter** modules, and/or as a POST payload received with the **Post Body** module.

When the **Execute Assembly** link at the top right of the editor is clicked for web services, an option will appear to run the web service in a new browser tab. This can be used to obtain the URL for invoking the web service.

When an assembly containing the Web Service Output module is saved, a system-wide unique web service name must be entered. This unique name can also be used to invoke the web service:



In the example above, the **flickrRecentPhotos** web service is invoked by name with a URL, by passing a **name** parameter.

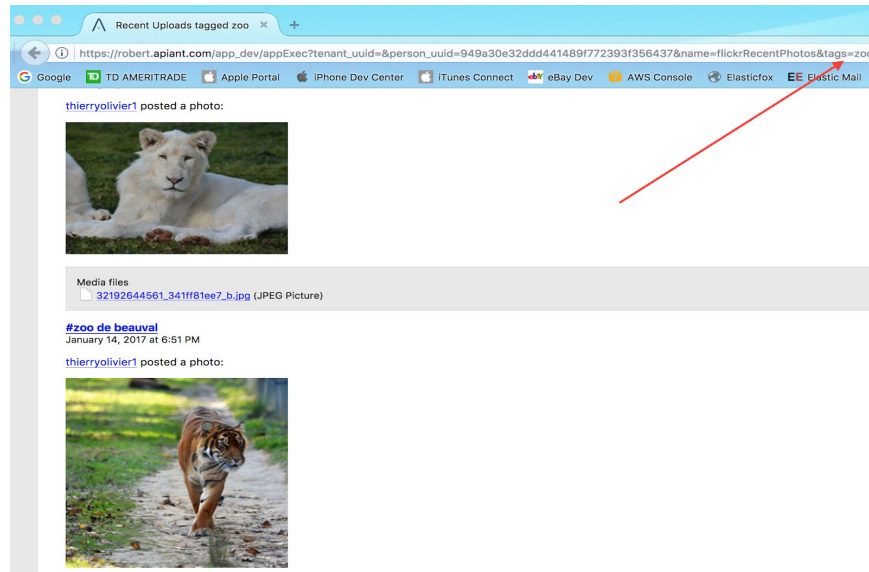
Web services are easily consumed within other assemblies in APIANT by dragging web services out of the catalog:



The example above shows the **flickrRecentPhotos** web service as it appears when placed within an assembly.

Notice that the **flickrRecentPhotos** web service has a query parameter, "tags". This comes from a top-level Query Parameter module embedded within the web service assembly.

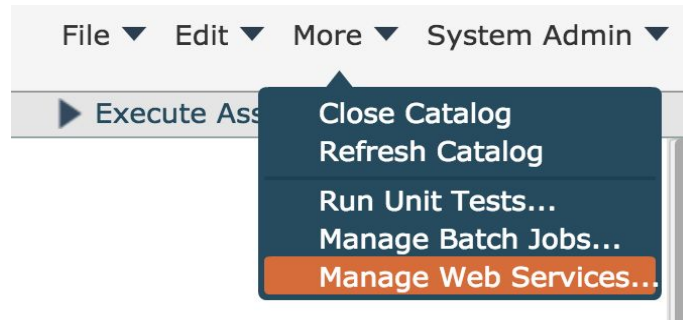
Query parameters are passed on the URL to the web service. The following shows how that is done:



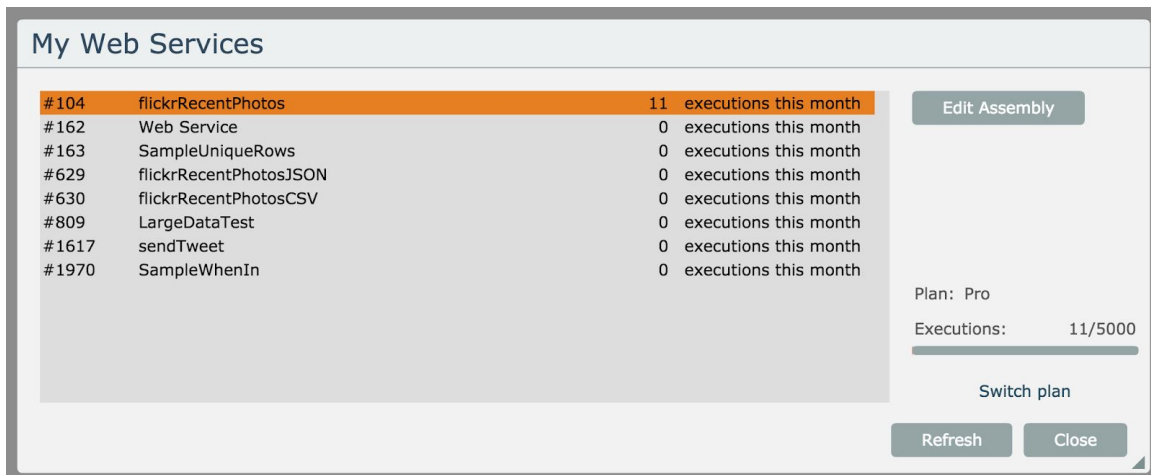
All top-level Query Parameter modules within the web service assembly can be specified in this manner on the URL used to invoke the web service.

Managing Web Services

Web services can be managed via the More => Manage Web Services menu:



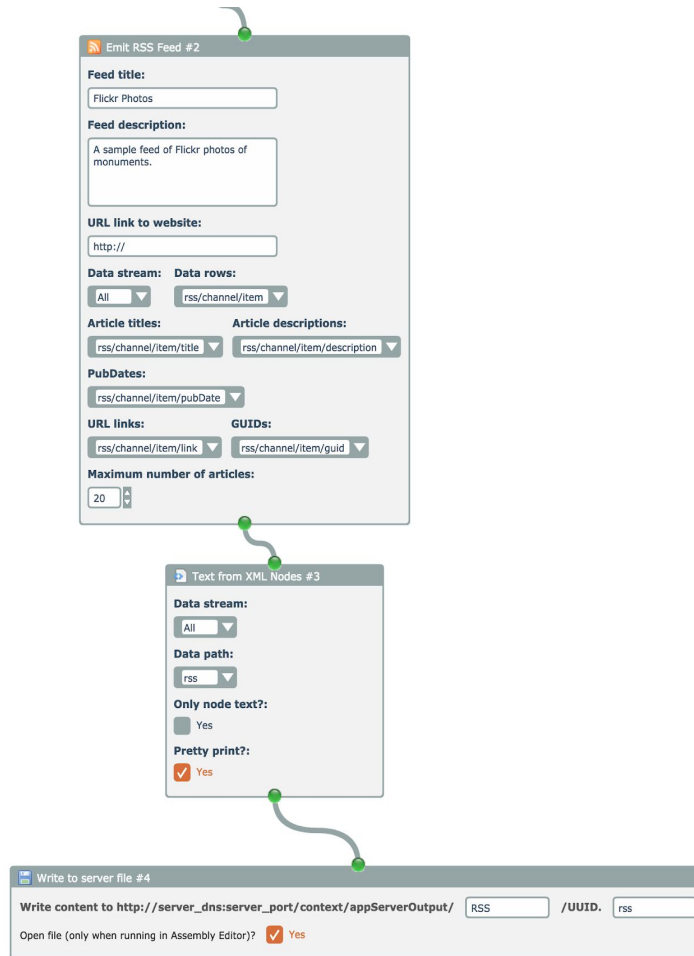
A dialog will open showing all your web services:




User accounts having a subscription plan will be able to see how many times their web services have been invoked for the current billing period.

RSS Feeds

An RSS feed is an XML file with a standardized structure, typically used for blog or news articles. Assemblies can be built that generate RSS feed files:



In this example, the **Emit RSS Feed** module is used to format the feed output. The **Text from XML Nodes** module then extracts the RSS feed contents as a string and saves it to a file on the server using the **Write to server file** module.

	<p>Tip: You typically want the RSS feed to be refreshed periodically. That can be done by making the RSS feed assembly a Batch Job. See the next section.</p>
---	---

Batch Jobs

Batch Jobs can be created to execute assemblies at a scheduled interval. They can be useful for background tasks that need to occur at a regular interval, like refreshing RSS feeds.

Batch Job modules define a batch job:

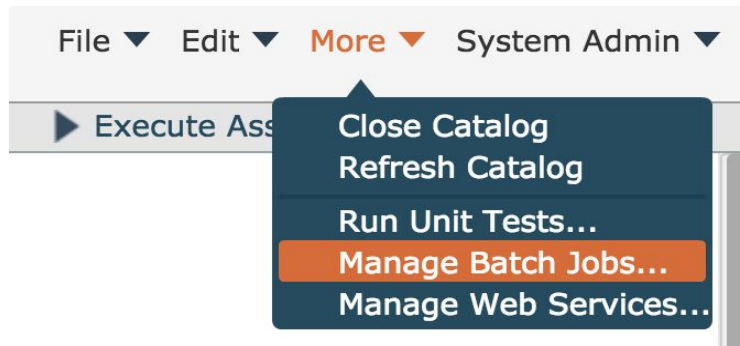
The screenshot displays the 'Assembly Diagram' section of the APIANT interface. On the left, a search bar contains 'batch job'. Below it, search filters are shown: 'All Catalogs' (selected), 'Community', 'Mine', 'Search for: Modules' (selected), 'Assemblies', 'Text within module fields', and 'Text within notes'. A 'Close' button is visible. Below the filters, a list of 'Matching Names & Descriptions' includes 'Batch Job (Cron) (Module)', 'Batch Job (Daily) (Module)', and 'Batch Job (Hourly) (Module)'. The main 'Assembly Diagram' area shows three interconnected modules: 'Batch Job (Cron) #2' with a 'Cron schedule:' field containing '*****'; 'Flickr Recent Photos #1' with a 'tags:' field containing 'monument' and an 'Edit Subassembly...' button; and 'Emit RSS Feed #3' with a 'Feed title:' field containing 'Flickr Photos' and a 'Feed description:' field containing 'A sample feed of Flickr photos of monuments.' Green lines connect the modules, indicating their relationships.

Three variations of Batch Job modules are currently available. The Cron version lets a raw cron schedule string be configured. The Daily and Hourly versions are simpler, where the system will randomly choose a schedule that runs at a daily or hourly rate.

Batch Jobs are not executed until they are activated. See the next section Managing Batch Jobs.

Managing Batch Jobs

Batch jobs can be managed via the More => Manage Batch Jobs menu:



A dialog will open showing all your batch jobs:



User accounts having a subscription plan will be able to see how many times their batch jobs have been invoked for the current billing period.

Note: Batch jobs only execute if the **Is Active** checkbox is checked.

Chapter 7: Other Functionality

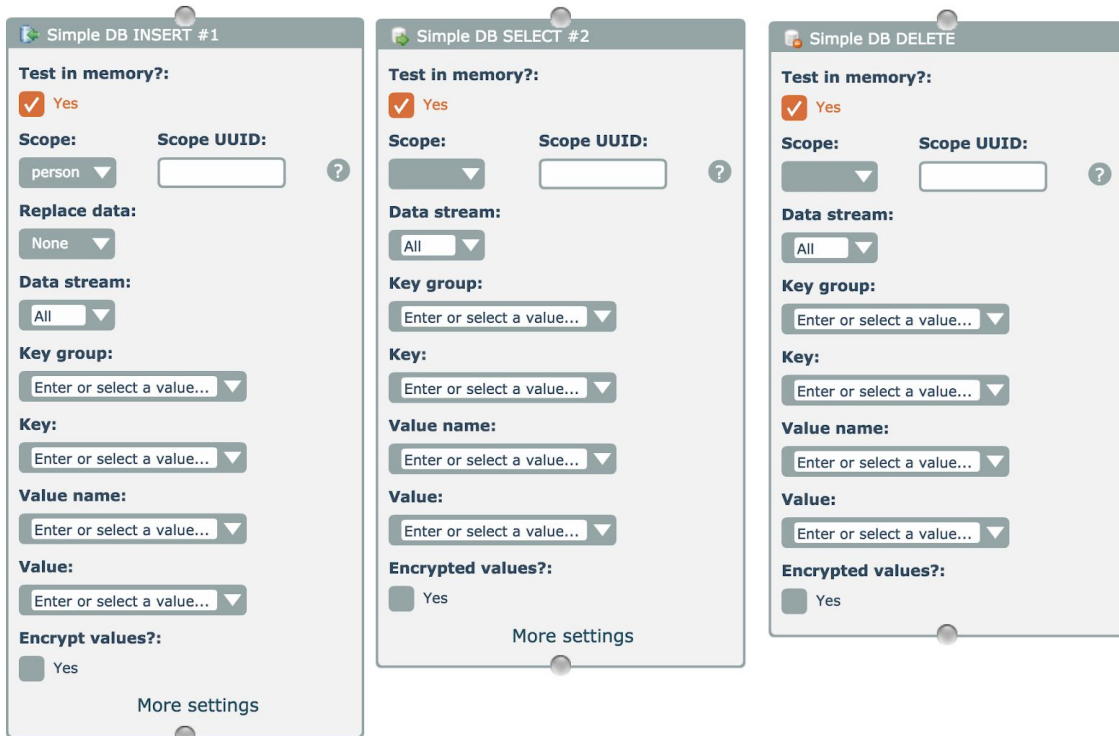
Simple DB

The system provides a generic database storage mechanism called the Simple DB. The Simple DB consists of a database table and modules for inserting, retrieving, and deleting data from the table.

The Simple DB database table has the following schema:

scope	scope_uuid	the_keygroup	the_key	the_value_name	the_value	the_value_secure	date_created
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	url	val	http://www.pyramidcam.com	NULL	2008-09-28 01:12:41
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	jscrip	val	false	NULL	2008-09-28 01:12:41
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	clip_url	val	doClipView.jsp?url=http://www.pyramidcam.com&jscri...	NULL	2008-09-28 01:12:41
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	url	val	http://data.cnbc.com/quotes/GOOG	NULL	2008-09-28 01:15:18
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	jscrip	val	true	NULL	2008-09-28 01:15:18
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	clip_url	val	doClipView.jsp?url=http://data.cnbc.com/quotes/GOO...	NULL	2008-09-28 01:15:18
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	jscrip	val	true	NULL	2008-09-28 01:17:15
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	url	val	http://www.weather.com	NULL	2008-09-28 01:17:15
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	clip_url	val	doClipView.jsp?url=http://www.weather.com&jscrip=...	NULL	2008-09-28 01:17:15
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	url	val	http://www.fallsviewcam.com/	NULL	2008-09-28 17:34:42
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	jscrip	val	true	NULL	2008-09-28 17:34:42
part_instance	c845525b840c4e478bde5165608e2b14	mygroup	clip_url	val	doClipView.jsp?url=http://www.fallsviewcam.com/&js...	NULL	2008-09-28 17:34:42

These modules are used in assemblies to interact with the Simple DB storage:



Data is organized into a 5-level hierarchy:

Scope => Key Group => Key => Value name => Value

All Simple DB modules work with data in a defined scope. The scope determines the visibility of the data across different system entities.



Data stored in the "person" scope is global across all assemblies for the user's account. The Scope UUID field can be set to a person account UUID retrieved from the **Get Account Info** module. If the Scope UUID field is empty, the currently signed-in user account is used instead.

Data stored in the "developer" scope is global across all assemblies owned by the developer. The Scope UUID field must be set to the developer's UUID, obtained by right-clicking on the Assembly Editor grid background and selecting the Get Developer UUID context menu option.

Data stored in the "automation" scope is accessible across all trigger and action assemblies within an automation. The Scope UUID field is unused.

Data stored in the "assembly" scope is accessible across all created instances of the assembly. The Scope UUID field is unused.

Data stored in the "assembly instance" scope is accessible across all modules in a single instance of the assembly. The Scope UUID field is unused.

The mashup and mashup instance scopes should not be used, they are only for visual widgets built in the system.

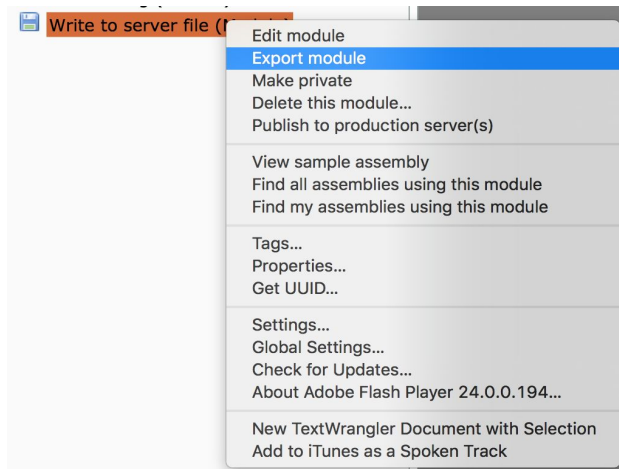
Importing and Exporting

Assemblies and modules can be transferred between APIANT systems via import/export.

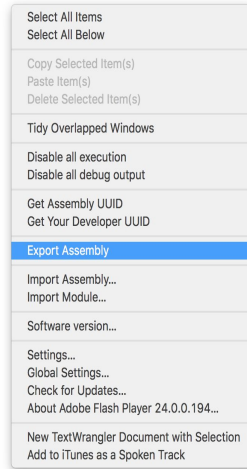
When an item is exported, all deterministic references it has to other content also gets exported.

Exporting

To export an item, right-click on it in the catalog to open the context menu and choose the **Export** menu item:



Or right-click on the background of an open assembly diagram to export that assembly:



The item will then be exported into a single file that will be saved to your local machine.



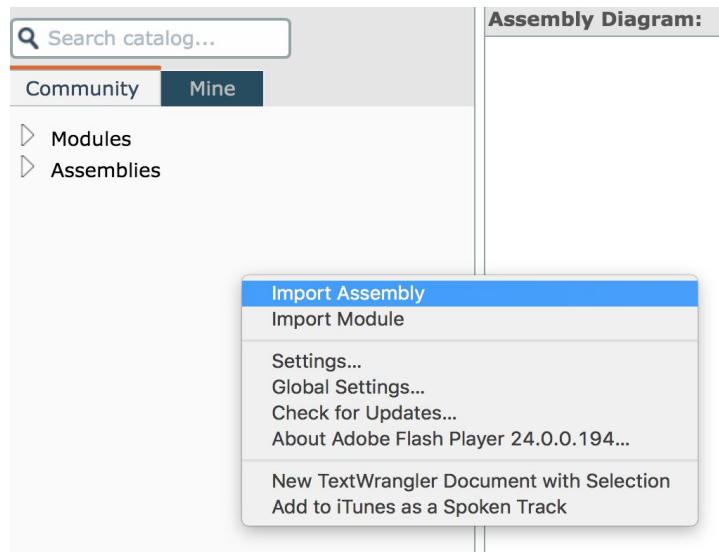
Note: The export menu option will only appear for items you own or those that have been shared with you.



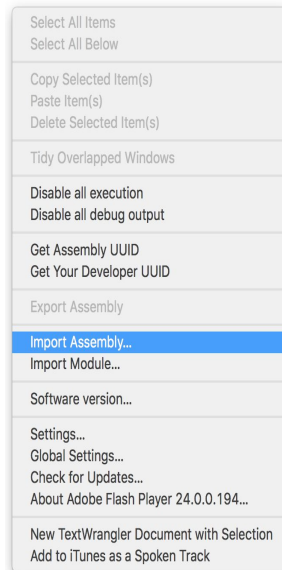
Warning! Do not alter the file name of the exported item.

Importing

Items can be imported in two ways. The first is to right-click on the module catalog's white background area to open the context menu and choose an **Import** menu item:



A second way to import is to right click on the editor's canvas to open its context menu and choose an **Import** menu item:

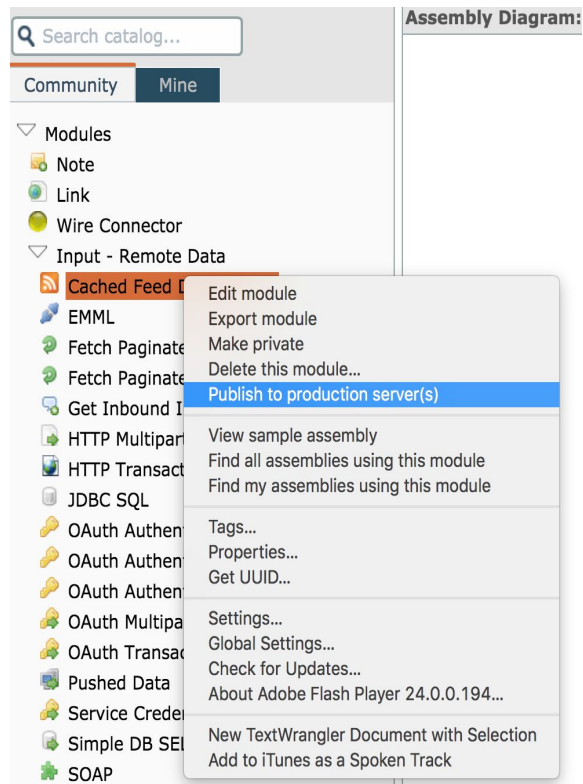


A browser dialog will appear for selecting the exported file from your local system. After a file is selected, it will be uploaded to the server and processed.

Publishing

APIANT can be configured with one or more Development Servers used to develop and test modules and assemblies. After content has been developed and tested, it can be published to one or more Production Servers.

From the catalog in a Development system, right click on an item that you own to access the publish option:



Click the **Publish to production server(s)** menu item to publish the item to all Production Servers associated with the Development Server.