Analytics Templates Developer Guide

Salesforce, Summer '25

Last updated: August 8, 2025

CONTENTS

WHY CRM ANALYTICS TEMPLATES?

Templates can bring the investment you make in CRM Analytics app development to life.

When you build an app using CRM Analytics Studio, you load data into CRM Analytics, transform it, create visualizations from the data, and design dashboards in a development or scratch org. Then, you store all the assets in a CRM Analytics app, which is actually a folder used to contain all an app's assets.

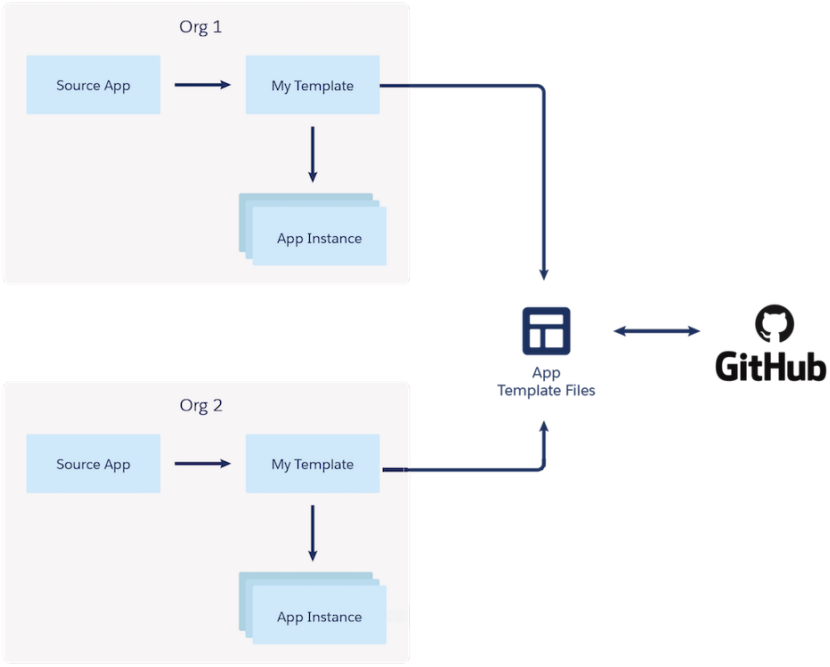
Instead of having to do all that work, other organizations can use your template to create their own version of your app built around their own data. After you develop your app in CRM Analytics Studio, turn it into a template with clicks in Analytics Studio, a simple command in Visual Studio Code, or the Salesforce command line interface (CLI). Then distribute it to customers, partners, and other teams within your company. They can deploy the template to their own Salesforce orgs and, with CRM Analytics Studio, use it to create a custom version of the app using their own data—either from Salesforce or another environment.

You can edit the template files to include features that let users customize the version of the app that they create from the template via a configuration wizard. When they create the app, they can customize the app's appearance, how it consumes and displays data, which dashboards and charts to display, and other aspects of the app's functionality. You can also design your template to create apps that require no customization or user interaction, but are automatically installed into an organization using a managed package. CRM Analytics embedded dashboards surface your templated app into orgs without requiring users to have access to CRM Analytics Studio.

The Source App

For the purposes of CRM Analytics Templates, the app you create in CRM Analytics Analytics Studio is called the *source app*. It's a collection of CRM Analytics assets—datasets, dataflows, recipes, lenses, and dashboards— stored together in a folder.

You create the template—actually a set of JSON files—from that source app. If you update the source app in Analytics Studio, you can also update the template. The same template can also be updated by source apps in different orgs. This lets your team work collaboratively on the source app and its template. That is, more than one developer can work on the template, at the same time.



Note: We use the term *source app* to refer to the app you work on, the one on which you base your template. When organizations use your template to create their own apps, we refer to them as *downstream apps*.

Templates can reuse dashboards, lenses, and datasets that are part of apps other than the source app by creating dependencies on those assets. The reused assets must be part of apps created from templates. See Share CRM Analytics Assets Between Apps.

Templated App Type

The app you're creating a template for can be a standard app that users install using the configuration wizard in CRM Analytics Studio. Or it can be an embedded app that is automatically installed for users and doesn't require customization or access to Analytics Studio. You can also create data templates that users install in the Data Manager. Data templates only create data assets.

In this guide, all template types follow the same steps for creation, iterative development, and testing the template. The main difference between the templates types is how the app or data assets are created in the user org from your template.

Standard App Templates

A standard app template has a `templateType` of `app`. A standard app template is listed in the Analytics Studio in the **Create App** menu. You can design a custom creation experience for CRM Analytics users by creating a configuration wizard. Users can customize the app assets after creation, reconfigure the templated app, and upgrade the app, all in Analytics Studio.

Embedded App Templates

An embedded app template has a `templateType` of `embeddedapp`. An embedded app doesn't require users to have access to Analytics Studio. The embedded app is created when a managed package containing the template and other supporting assets is installed in a Salesforce org. An embedded template uses the auto-install framework to install the app in an org without user interaction.

Embedded App Considerations

Embedded apps are designed for users who don't have the full CRM Analytics experience. As a result, the following restrictions apply for embedded apps:

- Only one embedded app instance per template can be created in a Salesforce org. Multiple embedded app templates can be present in a org.
- Dashboards can't contain Apex or SOQL steps.
- Bulk actions aren't allowed.
- App assets can't be customized by users after the app is created.

Embedded app templates aren't visible in Analytics Studio, nor are they returned in the Rest GET call to the `wave/templates` endpoint.

To create a successful embedded app template, you must include the `folder.json` and the `auto-install.json` files. The `folder.json` file specifies the name and label for the templated app along with setting sharing rules for the app. The embedded app can't be shared with other users and groups in Analytics Studio. The `auto-install.json` file specifies the configuration of the auto-install framework hook and the app installation.

After an embedded app is created, the app dashboards must be surfaced for users using the CRM Analytics Dashboard Lightning component.

Data Templates

A data template has a `templateType` of `data`. A data template only creates data assets, like recipes and datasets. A data template is listed in the Data Manager using the **New Data App** button. You can design a custom creation experience for Data Manager users by creating a configuration wizard. Users can customize and schedule the data assets after creation.

Data Template Considerations

Data templates are designed for users who want to load and transform data in Salesforce. As a result, the following limitations apply for data templates:

- In the `template-info.json`:
  - Use `data` for `templateType`.
  - Only the first entry in `templatePreviews` is shown to the user in the wizard.
  - `customProperties` aren't shown to the user in the wizard.
- A data template can only contain `datasetFiles`, `externalFiles`, and `recipes` assets.
- The following aren't currently supported in the Data Manager wizard:
  - In `template-info.json`:
    - `templateDependencies`
  - In `ui.json`:
    - `displayMessages`
    - `helpUrl`
    - `vfPage`
- The following are supported for data templates, but not app templates:

  - In `ui.json`:
    - `DatasetAnyFieldType` and `ArrayTypes` of `DatasetAnyFieldType` variables can be added to pages

For more information on the user experience for templated data apps, see Get Started Faster with Data Templates (Beta).

The Configuration Wizard

A CRM Analytics template can include a configuration wizard, made up of one or more pages with questions and answers, that allows users to customize the apps they create from the template. The configuration wizard is an optional but potentially powerful part of a template.

Important: If your template is `templateType` of `embeddedapp`, the configuration wizard isn't available for users. Embedded apps aren't visible or installable in CRM Analytics Studio. Instead, they use the auto-install framework to install without user interaction.

The configuration wizard guides users through creation of an app from the template. Users make selections that determine how the app they create from your template modifies the original. Most CRM Analytics templates developed by Salesforce include wizards. It's likely that every customer creates a different version of an app from the template. But every customer starts with the same app template.

A wizard can include questions about the data used to populate app dashboards. For example, it can ask about which Salesforce objects are used and which fields are used to store specific elements, such as user IDs, geographic location, product information, and revenue. A wizard can also ask about how to display data in the app and whether the user wants to include or exclude dashboards. Also, it can let users set elements of the UI, such as labels to identify KPIs or dashboard colors.

Wizards can also include conditional questions. For example, one question can ask if a user wants to display customer satisfaction key performance indicators about service agents. If the user answers yes, the wizard can then display more questions asking about how the user wants dashboards to consume and display customer service data.

Further, the wizard can contain an org data check. App datasets and dashboards usually depend on the Salesforce org where it's created to contain specific data— certain objects, or fields in objects—to render effective visualizations. The data check reviews the org for the required data. If it's in the org, app creation continues. If not, the wizard can display a message telling the user what requires fixing.

To create the configuration wizard, you modify the template JSON files that contain the instructions the CRM Analytics platform uses to create the app. When a user creates an app from a template, CRM Analytics looks at the template JSON files' contents, including your wizard selections. It follows instructions in the JSON to create the app.

The Template Object

Turning the source app into a template creates a `waveTemplates` object. The object contains JSON files that define an app's user experience assets and provide instructions to create the app.

Template objects—called `waveTemplates`—contain two types of JSON files:

- CRM Analytics app asset JSON files: Files that define the assets you create when you develop a CRM Analytics app using the tools available through CRM Analytics Studio: dashboards, lenses, datasets, recipes, dataflows, and components. Users also interact with these elements. Each app must have at least one dashboard and dataset. Dataflows and recipes are optional, but common for anything but the simplest template. Each asset has a corresponding JSON file.
- CRM Analytics template JSON files: Files with instructions used by CRM Analytics to create app assets from the template. You edit these files to give users the ability to customize their version of the app.

`waveTemplates` objects contain the following files.

File type	File name	Description
App asset JSON	*dashboard_name*.json	One JSON file for each dashboard in the source app, with the same name as its corresponding dashboard.
App asset JSON	*lens_name*.json	One JSON file for each lens in the source app, with the same name as its corresponding lens.
App asset JSON	dataset files • internal datasets – internal dataset files – user xmd.json • external datasets – *dataset*.csv – dataset_schema.json – user xmd.json	Most templates include datasets to provide meaningful content, but a template doesn't require a dataset. Apps without datasets can include a dashboard that's a shell with layout only, or one that uses a query to supply data. Dataset files can be internal or external. The template includes internal datasets defined by User XMD. External dataset files refer to CSV datasets such as dataset.csv, but they also include dataset_schema.json and extended metadata (user XMD) files, such as xmd.json. You must have one CSV file for every external dataset in the source app. Without it, there are no external files in the template. The dataset_schema.json and user xmd.json files are optional.
App asset JSON	dataflow files	Dataflows are one method to create internal datasets for your app, extracting and processing useful data. Templates support apps with multiple dataflows. Order the files carefully: The last dataflow can reference datasets inside the first, but the first dataflow can't reference the last.
App asset JSON	recipe files	Recipes are another method to create internal datasets for your app, extracting and processing useful data. Templates support apps with multiple recipes. Order the files carefully: recipes are created in listed order. If dataflows are present, they're executed before the recipes. For more information on adding recipes to a template, see Add a Recipe to a CRM Analytic Template.
App asset JSON	dashboard component files	Dashboard components are a type of dashboard widget that can contain other widgets and pages. These components are used to manage and reuse groups of charts, tables, filters, text, and more in multiple dashboards in your app. Each app created from your template creates a unique instance of the component. The app dashboards reference these components by the unique name to ensure the component functionality is correct.
App asset JSON	data transform files	Data transforms create internal datasets from Data Cloud data model objects. In Winter '24, data transforms can only be added to a template manually.
Template JSON	image files (JSON)	Include if images are used in dashboards in the source app or referred to in template-info.json for display in configuration wizard pages, such as the template picker or template detail page. One image file for each image. Images must be stored as Salesforce static resources. If an

File type	File name	Description
		org is namespaced, set the static resources `namespace` attribute in the `template-info.json` entry for the image.
Template JSON	`template-info.json`	Defines the template metadata referencing all elements of your template, including all the app asset and template asset JSON files defined in this table.
Template JSON	`folder.json`	Lists the order in which dashboards (`featuredAssets`) appear. If not set, dashboards appear in alphabetical order.
Template JSON	`releasenotes.html`	Defines the text describing the features and benefits of the template. Update to describe enhancements to a new template version.
Template JSON	`variables.json`	Defines the variables used by the template JSON files. Includes text for configuration wizard questions and specifications for the answers
Template JSON	`ui.json`	Manages the configuration wizard. It defines the number of wizard pages, the order of wizard questions, and any messages you want the user to see as the make wizard selections. Configuration settings can determine if a wizard page or question is hidden based on selections made in answer to other wizard questions.
Template JSON	`org-readiness.json`	Defines the validations that check if the org can create an app from the template successfully. Some validations run when the template is loaded, before the configuration wizard loads, and others run during app creation. Validation success and error messages can be defined and passed to the user.
Template JSON	`template-to-app-rules.json`	Defines the rules the template follows when creating the app, including changes made to app assets in response to user selections in answer to wizard questions. For example, a rule can determine that if an org doesn't contain data in Salesforce objects, app dashboards don't refer to those objects and the objects are excluded from the dataflow. Rules also determine how the template handles variables. For example, if the wizard asks which fields to include in filters, `template-to-app-rules.json` determines how that choice is reflected in dashboards. Rules are applied when a user creates an app from the template. Typically, a template uses multiple rules files, which must be ordered intentionally. Rules are applied in the order in which you list them. Rules listed later can reference rules listed earlier, but the first rule can't reference any rules listed after it.
Template JSON	`app_to_template_rules.json`	Defines rules for changing template files after editing the source app. If you make a change that impacts many parts of the app, edit this file. For example, a template creates a waterfall chart based on a dataset selected by the user in the wizard. The sections of the waterfall show a series of measures, also selected using the wizard. To enable the selections to be represented as variables, tokenize a field in the dashboard when updating the template. When the app is created, the value of a variable is replaced.

File type	File name	Description

		To replace the hard-coded value with a token, add a rule to `app_to_template_rules.json`, like this:
		`{`
		`"action":"<action_name>",`
		`"key":"\"<field_name>\"",`
		`"value":"\"${Variables.phoo.fieldName}\""`
		`}`

Template JSON	`auto-install.json`	Optional template file to mark the template for automatic installation. Defines the auto-install hook and app configuration settings for the auto-install framework. Templates with this file present must be of template type `app` or `embeddedapp`. This file isn't generated on template creation, you must create it. Templates created for the auto-install framework must be part of a managed package and installed into customer orgs. On package installation, an auto-install request is generated from this file configuration and the templated app is created automatically.

CHOOSE YOUR TOOL

Microsoft Visual Studio (VS) Code together with Analytics plugin for Salesforce command-line interface (CLI) commands is the recommended way to develop CRM Analytics templates. Or, to create and manage CRM Analytics templates, you can use CRM Analytics Studio, the Salesforce Analytics CLI by itself, or VS Code.

SEE ALSO:

- Analytics Development Tools

THE PROCESS AT A GLANCE

After creating a source app in CRM Analytics, follow these steps to create and work with a template. The next sections of the developer guide provide details about each step.

Start by developing an app in CRM Analytics Studio. That becomes the source app you use as the basis for your template. We recommend that you complete Steps 1 through 4 and 8 in Microsoft Visual Studio (VS) Code, issuing command line interface (CLI) commands from the tool's Command Palette and using its JSON editor.

1. **Create the WaveTemplate object and files.** Create the first version of the template from the source app.
2. **Retrieve (export) the WaveTemplate object and files.** Make the files available on your local file system for editing.
3. **Edit the JSON files.** Build the configuration wizard user interface, variables, conditions, and rules into your template by editing the JSON files. You can also customize the wizard with VisualForce and add other enhancements. Use your preferred Integrated Development Environment (IDE) to edit the files, regardless of whether you create them using CLI or VS Code.
4. **Deploy the WaveTemplate object.** After editing the files in an IDE, push the them back to your development org for testing. Deploy the object every time you edit the JSON. Pulling the template files back from the development org overwrites the files on your local workstation.
5. **Test the template.** Testers can now create apps from the template in Analytics Studio for testing. After getting test results, retrieve the template object files again (Step 2), make further edits (Step 3), and redeploy them (Step 4). It's an iterative cycle. And every time you change the JSON files, and deploy your template, test the template again.
6. **Share the template with users.** Enables creation of apps from the template.
7. **Create apps from the template.** Users create apps in Analytics Studio or by installing a managed package. Based on their feedback, continue iterative development of the app and template.
8. **Update an existing template.** Update the template after editing source app assets, including dashboards, lenses, datasets, recipes, and dataflows in Analytics Studio.

SEE ALSO:

Create an App

PREREQUISITES

Take care of these tasks before creating CRM Analytics templates.

Licenses and Permission Sets

Make sure you and members of your team each have a CRM Analytics platform license--either CRM Analytics Growth or CRM Analytics Plus.

Assign the following permission sets to yourself and members of your team:

- For CRM Analytics Growth licenses, CRM Analytics Admin
- For CRM Analytics Plus licenses, CRM Analytics Plus Admin

Org preferences

Set your Org Preferences to enable CRM Analytics templates. In the Admin Setup page, under Settings, select Enable CRM Analytics Templates. You must enable this preference to create CRM Analytics templates.

Developer Edition Org Namespace

Set up a namespace for your development org to enable the use of Managed Packages. Set up your namespace before creating any CRM Analytics assets.

CLI Developers

To use CLI for template development, enable the Dev Hub, which lets you create and manage scratch orgs. Also, install the Analytics CLI Plugin.

1. Enable Dev Hub in your org. See Enable Dev Hub in Your Org.
2. Install the Salesforce Command-Line Interface (CLI). See Install the Salesforce CLI.
3. Create a Salesforce DX project. See Create a Salesforce DX Project.
4. Create a scratch org. See Create Scratch Orgs.
5. Install the CLI Analytics plugin by running the command `sf plugins install @salesforce/analytics`.
6. Verify the installation by listing the available analytics commands. Run the command `sf analytics --help`.

For a reference to all available Analytics CLI commands, see the Salesforce Analytics CLI Plugin Command Reference.

Visual Studio Code Developers

Use VS Code as your template IDE for template customization. VS Code provides the CLI commands from the Command Palette and content assist and validation for the template json files. For more information, see Use Visual Studio (VS) Code and the Salesforce Extensions for CRM Analytics Template Development.

CRM Analytics REST API access

Refer to the documentation on Authentication to the CRM Analytics REST API to access and use the API.

Source app and its ID

Typically, you build an app and it becomes a source app when you create a template from it. You can also use a template to create a source app. However you create the app, note the 18-character app ID in the URL. It's shown here in boldface:`https://salesforce.com/analytics/wave/wave.apexp?tsid=2x0x0002xxx#application/0Ol0b000002AynIAAS`. The ID becomes the `folderId` you use as you work with the WaveTemplate object.

Create a Source App From Scratch

Use CRM Analytics Studio to build an app with lenses, dashboards, datasets, recipes, and the default Salesforce dataflow.

Create a Source App From a Template

1. Create an app from the template using the wizard. Use the Analytics CLI to create the app if you must set variables that the wizard doesn't expose. In the CLI, use an `analytics app create` command.
2. Next, turn the app into the source app by coupling it with the template. With the Analytics CLI, use `analytics template update` with the `folderSourceID`. See Step 8: Update an Existing Template. on page 101

Now, the new app becomes the source app in the org. Any updates made to the folder assets in the new source app are tied to the template.

Org Data Considerations

Creating a template from your app isn't a way to move data from one org to another. Datasets are templatized with header references. For datasets built by recipes and dataflows, they're populated with data from the user org when the app is created. For external datasets created by CSV uploads, only minimal data is present when a new app is created from the template. For example, if the source app contains a dataset created with a CSV upload, the template is created with a sample CSV that is only one dataset part of the original CSV. When a user creates an app from the template, they must upload their own CSV or a CSV provided to them to update and complete the dataset.

STEP 1: CREATE THE WAVETEMPLATE OBJECT

Once you've created an app to serve as your source app, turn it into a template by creating the `WaveTemplate` object and all its files.

You can use CRM Analytics Studio, commands in the CLI or Microsoft Visual Studio (VS) Code to turn your app into a template.

To use CRM Analytics Studio, see Create and Manage CRM Analytics Templates in Analytics Studio.

To use the commands, first locate the 18-character app ID—the folder ID—for the source app. You can find it appended to the URL for the app (in bold):

`https://salesforce.com/analytics/application/`**`0Ol0b000002AynIAAS`**`/edit`

Create a Template with the Command Line Interface

Use `analytics template` commands to create templates with the Salesforce Analytics Command Line Interface (CLI). You can also use the CLI to manage existing templates.

1. Get a list of available apps. In the CLI, enter the command `sf analytics app list`. The command returns a list of all apps in your org.

```
>>sf analytics app list

NAME         LABEL         FOLDERID            STATUS            TEMPLATESOURCEID
___________  ____________  ____________  ______

MyNewApp   MyNewTemplate   00l0S000000MmGQQA0   newstatus
SharedApp1   Shared App    00l0S000000Mm9tQAC   completedstatus   0Nk0S0000008OJDSA2
SharedApp    Shared App    00l0S000000EEVUQA4   newstatus         0Nk0S0000008OJDSA2
```

2. Look for your source app. If you haven't already identified its folderID, find it in the response to the `list` command.
3. Create the template. Enter the command `analytics template create`. Here's syntax for the command.

   `analytics template create -f` ***`folderid`*** `[-u` ***`username`*** `--apiversion` ***`api_version`*** `--json --`***`loglevel`*** `log_level]`

`folderid`	Required. The ID of the source app to use when creating the template. You can use `analytics:app:list` to see information about the apps on your org.
`username`	The user name or alias of the target org. If not specified, CRM Analytics uses the default org.

api_version	The API version to use for this command.
`--json`	Use this flag to specify format the output as JSON.
`--loglevel`	Specify the log level for this command. Allowed values are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`

The command returns something like the following: `Successfully created analytics template [0Nk0S0000008OJNSA2]`

The WaveTemplate object and all its files are on your scratch org. Next, retrieve the object by pulling the files to your local workstation for editing.

SEE ALSO:

Salesforce Analytics Plugin CLI Command Reference: template Commands

Create and Update a Template with Visual Studio Code

Use Microsoft Visual Studio (VS) code to create and update a template.

1. Create the template. In the Command Palette, enter **Create Analytics Template**.

   Select the Analytics app from the dropdown list.

   The command returns something like the following: `Successfully created Analytics template [0Nk0S0000008OJNSA2]`

   The `WaveTemplate` object and all its files are on your scratch org. Next, retrieve the object by pulling the files to your local workstation for editing.

STEP 2: RETRIEVE (EXPORT) THE WAVETEMPLATE OBJECT

CRM Analytics templates are no different than any other metadata (such as Visualforce pages and Apex classes). You can retrieve the `WaveTemplate` object and its JSON files for editing using CLI or Microsoft Visual Studio (VS) Code. Then store them in a source code management system, and work with them using the development environment of your choice.

Note: Performing this step overwrites your template object folder contents. If you've edited JSON files (such as `rules.json`), make sure you first push those changes back to the development org as described in Step 4: Deploy the WaveTemplate Object on page 92.

Use the Salesforce CLI to Retrieve the Template Files

Pull the template metadata from the scratch org to your local workspace using the Salesforce CLI command `sf project retrieve start`.

The command stores the template files on your system under the folder `force-app\main\default\waveTemplates\MyNewTemplate`

Access them locally on your file system for further development using the IDE of your choice.

Note: If you created your template in CRM Analytics Studio, use the CLI to log in to your org and then use the `sf project retrieve start --metadata "WaveTemplateBundle:MyTemplate"` CLI command to retrieve your files for editing. For information on setting up and using the CLI for template development, see Prerequisites.

SEE ALSO:

Salesforce CLI Command Reference: project retrieve start

Use Visual Studio Code to Retrieve the Template Files

Pull the template metadata from the scratch org to your local workspace using the Command Palette **Pull Source from Default Scratch Org**. Use VS Code to view and edit the template files.

The command stores the template files on your system under the folder `main\default\waveTemplates\MyNewTemplate`

Use VS Code to edit and customize the template. You must have the Salesforce Analytics Extension Pack installed. To get content assist and validation when you edit a template json file, make sure the VS Code language mode is set to **ADX Template Json**.

For information on setting up and using the VS Code for template development, see Prerequisites.

Template Folder Structure

Decompressing the exported template file exposes the following folder structure.

-  *<TemplateName>*

  (top directory)

  -  `template-info.json`

    Don't change the name of this file; processing depends on it.

  -  `variables.json`
  -  `ui.json`
  -  `template-to-app-rules.json`

    We recommend this naming convention as the best way to manage your rules files.

  -  `app-to-template-rules.json`

    We recommend this naming convention as the best way to manage your rules files.

  -  `releaseNotes.html`
  -  `dashboards`

    Subdirectory containing one or more dashboard JSON files corresponding to each dashboard in the source app. If the source app has no dashboards, this subdirectory doesn't exist.

  -  `lenses`

    Subdirectory containing one or more lens JSON files corresponding to each lens in the source app. If the source app has no lenses, this subdirectory doesn't exist.

  -  `dataflows`

    Subdirectory containing the dataflow JSON files corresponding to each dataflow in the source app. If the source app has no dataflows, this subdirectory doesn't exist.

  -  `recipes`

    Subdirectory containing the recipe JSON files corresponding to each recipe in the source app. If the source app has no recipes, this subdirectory doesn't exist.

- 
`external_files`

  Subdirectory containing one or more CSV dataset and related JSON files, such as schema and user XMD files, corresponding to each dataset in the source app. If the source app has no CSV datasets, this subdirectory doesn't exist.

- 
`dataset_files`

  Subdirectory containing User XMD files for SFDC datasets. If the source app has no SFDC datasets with User XMD defined, this subdirectory doesn't exist.

- 
`images`

  Subdirectory containing one or more image files. If the source app has no images, this subdirectory doesn't exist.

- 
`folder.json`
- 
`auto-install.json`

  Optional file to create the app with an auto-install request. You must create this file, It is not generated at template creation. The template must be installed as a managed package for app creation.

STEP 3: EDIT THE JSON FILES

Your source app looks fantastic, with widgets rendered beautifully in their dashboards. Next, create a template that gives users the same experience as your app but with data unique to their Salesforce orgs. For standard apps, add a configuration wizard to customize app creation. The wizard helps an admin using your template add data, name a dashboard, label a chart, or add their own touches. For embedded apps, add the `auto-install.json` file. You control the process by editing the files that constitute the template assets, the`template-info.json`, `ui.json`, `variables.json`, and any number of `rules.json` files. These JSON files open a world of flexibility and power in the template creation universe.

The files interact as follows:

- The `ui.json` file uses the variables in the `variables.json` file to dictate what goes on each page of the wizard. The `ui.json` file can contain conditionals that dictate which questions or pages are displayed in the wizard.
- `variables.json` contains all template variables, including text for wizard questions and specifications for the answers. The file can also define conditional questions. For example, a question only appears if an org contains certain data. Or if a wizard question is answered in a way that requires more detail, the wizard presents other conditional questions.
- The `template-info.json` file identifies all the template components, including the references to the rules, variables, and UI files. The `template-info.json` file can contain conditionals that dictate whether assets are generated in the downstream app.
- The `template-to-app-rules.json` files use variables to set constants, and then use rules to set values that dictate how assets within the downstream app are generated. The most common uses for rules include adding and removing dashboard widgets and dataflow actions.
- The `app-to-template-rules.json` file defines templatizing the source app. It's edited only when you change the source app to change the template.
- The `folder.json` file enables you to set a preferred order of the dashboards in the downstream app, as opposed to leaving them alphabetized.
- The optional `auto-install.json` file is used when a template is part of a managed package. It defines the auto-install hook and the automated app configuration settings used to generate an auto-install request when the package is installed. This file is created manually, it isn't generated on template creation.

And yet, editing these JSON files is an optional step. A template is complete even without you touching any of these files. Such a minimal template creates apps that mirror the source app without any customization. You can test your template without any editing to validate that app creation works before moving on to more complex customizations.

Don't confuse the template assets discussed here with the JSON files representing the *source app assets*. For example, `dashboard_name.json` and `lens_name.json` are also exported as part of the Metadata API retrieval. We recommend that you alter app assets and widgets in CRM Analytics Studio, remembering to update the template object using Analytics Studio to update the template. That means you don't manually edit their corresponding JSON files.

Wizard Best Practices

To deliver an optimal user experience, adopt these best practices when creating a wizard.

Edit template-info.json

`template-info.json` describes the template. It references all the information required to create an app from the template.

Edit folder.json

The `folder.json` file describes the `featuredAssets` for the application.

Edit auto-install.json

`auto-install.json` configures the template for the auto-install framework. Always create this optional file if you plan to design your template for auto-installation via a managed package with an auto-install request.

Edit org-readiness.json

The `org-readiness.json` file configures the validations to run on a user org to ensure it can create an app successfully.

Edit variables.json

The `variables.json` file describes all the variables used in the `template-info.json`, `ui.json`, and the different `rules.json` files.

Edit ui.json

The `ui.json` file determines how your template displays the configuration wizard questions defined in `variables.json`.

Edit Rules Files

Rules files define how an app gets created by the template.

Use the Apex WaveTemplateConfigurationModifier Class

The `WaveTemplateConfigurationModifier` class checks an org's data. Also, to simplify app creation, it can modify the template configuration wizard accordingly.

Use the Apex Access Methods

The `Access` class provides utility methods to check Integration User access to sObjects and sObjectFields. Use these methods in your `WaveTemplateConfigurationModifier` implementation.

Wizard Best Practices

To deliver an optimal user experience, adopt these best practices when creating a wizard.

- Strike a balance between too few questions and too many. Too few questions provide insufficient flexibility for the user creating the app, while too many questions can be daunting.
- Strike a balance between the number of questions on each wizard page and the number of pages in the wizard. Too many questions on each wizard page require the user to scroll down, a practice to avoid. A wizard with too many pages is cumbersome. We recommend a page that contains 5-8 questions.
- Group wizard questions by theme. For example, questions about quotas on one page and questions about products on another.
- Use templates without a configuration wizard for testing the creation of the template object.
- Organize like variables together. For instance, group variables about the Product dimension together.
- Consider using the Apex `WaveTemplateConfigurationModifier` class to create a smart wizard that can automatically check the data in your org. See Use the Apex `WaveTemplateConfigurationModifier` Class on page 79.
- You can customize smart wizard pages using VisualForce pages and a JavaScript library.
- For embedded apps, don't create any questions that depend on user interaction unless you create your own user interface for installation or you add a `WaveTemplateConfigurationModifier` class. Auto-install requests automatically create the app in customer orgs on package install of the template, with no CRM Analytics Studio configuration wizard for users.

Edit template-info.json

`template-info.json` describes the template. It references all the information required to create an app from the template.

`template-info.json` can also contain conditions. For example, a source app can have a dashboard with charts. You can include conditions in the `template-info.json` file to remove specific charts when the user creates the app. Conditions can include

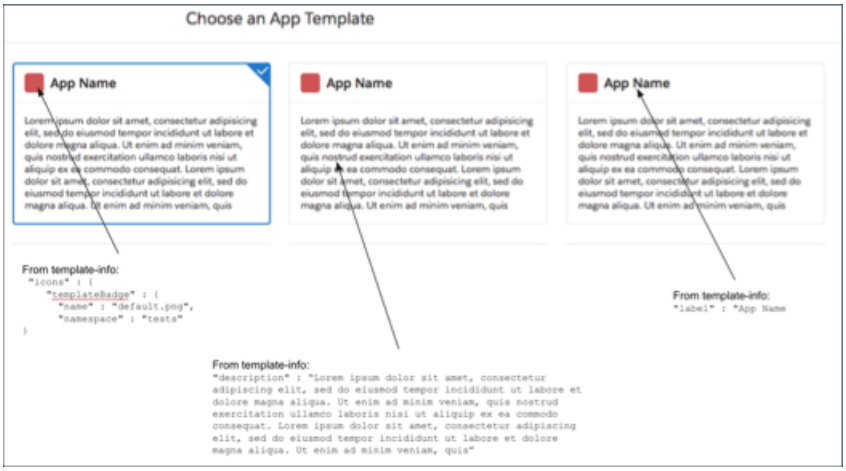
overrides that enable the addition or exclusion of assets and data from the app. The override is defined in `variables.json`. See Complex `variables.json` Variable Types.

The `template-info.json` file has multiple parts:

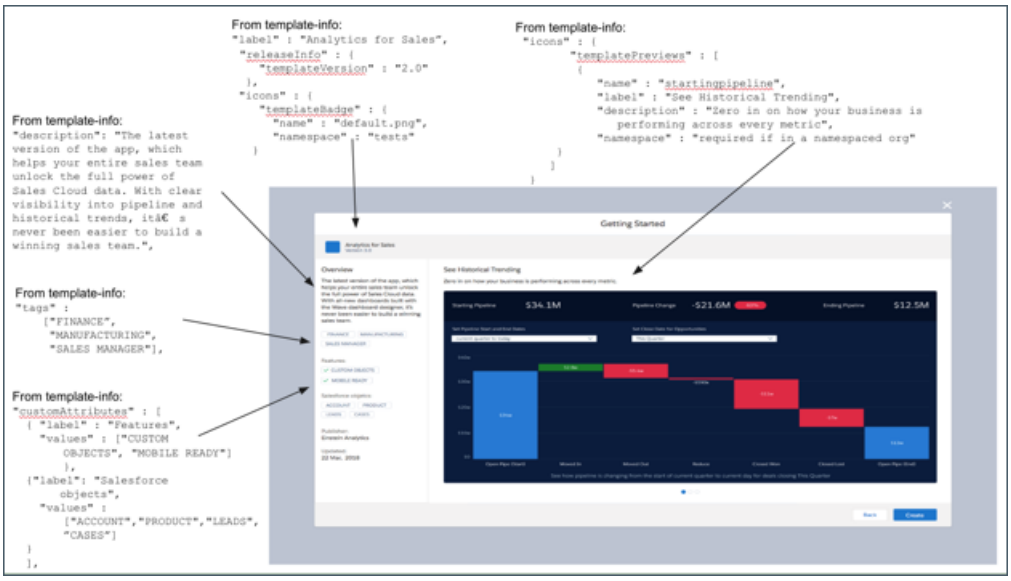
- Metadata information for the template: Name, description, icons, tags, and custom attributes.

```
{
  "label": "Analytics Training Materials",
  "name": "analyticsEducation",
  "description": "An example template to generate datasets, dashboards and lenses for
the Analytics introduction course.",
  "assetVersion": 49,
  "templateType": "app",
  "icons": {
     "templateBadge" : {
         "name" : "learn"
     },
     "appBadge" : {
          "name" : "16.png"
     },
     "templatePreviews" : [
          {
             "name": "preview_education",
             "label": "Analytics Training",
             "description": "Learn about templates to generate Analytics assets."
          }
     ]
  },
  "tags": [
     "Learn",
     "Templates",
     "Apps"
  ],
  "customAttributes": [
     {
          "label" : "Features",
          "values" : [
              "Prebuilt Apps",
              "Mobile Ready"
           ]
     }
  ]
},
```

  - `label` contains the user's name for the app. Analytics Studio displays in the app creation wizard, such as in the template picker and template details page.



- The `name` attribute is the unique developer ID for the template ID. Changing it generates a new template.
- `description` specifies the description of the template. The template picker displays the contents of `description`, as shown in the previous image.
- `assetVersion` specifies the version for the template assets. The version is set at the time of creation, generating a snapshot of the asset JSON files. The version doesn't update unless indicated when updating a template. For more information, see Update an Existing Template.
- `templateType` specifies the type of template:
  - `app`: a standard app
  - `embeddedapp`: an embedded app
  - `data`: a collection of data assets
  - `dashboard`: a single dashboard
- `icons` attribute contains three possible options:
  - `templateBadge` defines the small icon that appears in the app picker (shown in the previous image) and at the top of the template details window (shown in the next image). Don't use file extension with `name` attribute, for example `"learn"`, not `"learn.png"`.
  - `appBadge` defines the icon shown by Analytics Studio for the app after creation. Must be `"1.png"` through `"21.png"`. Use file extension with `name` attribute, for example `"16.png"`, not `"16"`.
  - `templatePreviews` defines preview images displayed on the template details window during the app creation process. Supports multiple images. Don't use file extension with `name` attribute, for example `"preview_education"`, not `"preview_education.png"`.
- `tags` specifies tags that are used in template search and that appear on the template details page (shown in the next image).
- `customAttributes` specifies information describing the template in the app creation wizard, such as Salesforce objects used or notable features (shown in the next image).



- Template version information: The `templateVersion` is a string validated as "`#.#`". The `notesFile`, when present, must be an HTML file. If `releaseInfo` is present, users can reconfigure or upgrade an app that's been created from the template from Analytics Studio.

```
"releaseInfo":{
    "templateVersion": "1.0",
    "notesFile": "releaseNotes.html"
},
```

  Note: After creating an app from a template for the first time, users can reconfigure the app based on the existing version of the template. When you update the number in `templateVersion`, CRM Analytics prompts users who created an app from the template to upgrade the app to the new version. for more information about reconfiguring and upgrading apps from templates, see Reconfigure an Analytics App and Upgrade an Analytics App.

- Reference to the variables file: The file that contains all the variables used in the template is `variables.json`.

```
"variableDefinition": "variables.json",
```

- Reference to the configuration wizard file: The file that defines the wizard the user fills out, answering questions to set variables, is `ui.json`.

```
"uiDefinition": "ui.json",
```

- Reference to the org readiness file: The file that defines the validations to perform when the wizard loads is `org-readiness.json`. These validations verify that the org has the necessary data and setup to support creating an app from the template without failing.

```
"readinessDefinition" : "org-readiness.json"
```

- Reference to one or more rules files: The file or files defining any rules to be applied to the template assets is `rules.json`.

```
"rules" : [ {
    "type" : "templateToApp",
    "file" : "template-to-app-rules.json"
  }, {
    "type" : "appToTemplate",
```

```
    "file" : "app-to-template-rules.json"
  }
],
```

The `templateToApp` rules file defines rules that run when a downstream app is created or updated from a template. These rules are the rules you're most likely to edit.

The `appToTemplate` rules file defines rules that run when a template is created or updated from a source app. These rules are created by the framework code. You aren't likely to edit them.

- List of objects defining dashboards and lenses: A dashboard or lens entry can contain a condition statement to determine whether the asset is added at app creation time based on a given variable. This condition statement can be an empty list (`"[]"`).

```
"dashboards": [
  {
    "file": "dashboards/Elections_2012_Dashboard.json",
    "label": "elections_dashboard",
    "name": "Elections Dashboard",
    "condition" : "${Variables.StringValueInclude == 'Yes'}"
  }
],
"lenses":[
  {
    "file": "lenses/Election_Breakdown_By_State.json",
    "name": "Election Breakdown By State"
  }
],
```

- Reference to external datasets: List of files that define external datasets (CSVs) to create, can include XMD. Each dataset entry can contain an entry for a conditional statement, allowing for decisions to be made by variables on whether a dataset asset is added at app creation time. This condition statement can be an empty list (`"[]"`).

```
"externalFiles":[
  {
    "type" : "CSV",
    "name" : "State_Codes",
    "file" : "external_files/StateCode.csv",
    "schema" : "external_files/StateCode_schema.json"
  },
  {
    "type" : "CSV",
    "name" : "Election_2012",
    "file" : "external_files/Election_2012.csv",
    "schema" : "external_files/Election_2012_schema.json", (optional)
    "userXmd" : "external_files/Election_2012_XMD.json" (optional)
  }
],
```

- References to datasets: The SFDC dataset builder creates these datasets. Label is optional.

```
"datasetFiles": [
  {
    name": "Election_1980",
    "label": "Election 1980",
    "userXmd": "dataset_files/Election_1980_XMD.json"
  },
```

```
  {
    "name": "Election_2012",
    "label": "Election 2012"
  }
],
```

User XMD is NOT required. If it's present, there's an XMD JSON file in the `dataset_files` directory. Otherwise, no XMD JSON file is present. If User XMD exists, it must be v2.0.

Each dataflow or recipe file must contain a reference to the corresponding dataset (Extract and Register steps). That enables the dataset to be recreated in the creation process of any downstream app.

Live datasets are supported with the `liveConnection` parameter. For more information, see Use Live Datasets in a CRM Analytics Template.

- Reference to dataflow files: List of files that define dataflows. This reference can be an empty list (`"[]"`). If your app includes multiple dataflows, reference each dataflow file. When a user creates an app from the template, dataflows are created in the order they're entered here.

```
"eltDataflows" : [
  {
    "label" : "One DF",
    "name" : "One_DF",
    "file" : "dataflow/One_DF.json"
  },
  {
    "label" : "Two DF",
    "name" : "Two_DF",
    "file" : "dataflow/Two_DF.json"
  }
],
```

- Reference to recipe files: List of files that define recipes. This reference can be an empty list (`"[]"`). If your app includes multiple recipes, reference each recipe file. When a user creates an app from the template, recipes are created in the order they're entered here. Edit the order as needed. If dataflows exist in the app, they execute before the recipes. For more information, see Add a Recipe to a CRM Analytics Template You can also specify whether a recipe should sync and execute during app creation with the `executeCondition` attribute. For more information, see Configure Recipe Execution.

```
"recipes" : [
  {
    "label" : "Filter States Recipe",
    "name" : "Filter_States_Recipes",
    "file" : "recipes/filterRecipe.json"
  },
  {
    "label" : "Append Cities Recipe",
    "name" : "Append_Cities_Recipe",
    "file" : "recipes/appendRecipe.json",
    "executeCondition" : "CreateOnly"
  }
],
```

- Reference to dashboard components: List of files that define components in dashboards. This reference can be an empty list (`"[]"`). Each dashboard component in the app's dashboards gets a unique entry in this list. Each component entry can contain an entry for

a conditional statement, allowing for decisions to be made by variables on whether a component asset is added at app creation time. This condition statement can be an empty list (`"[]"`).

```
"components" : [
  {
    "label" : "ImageComponent",
    "name" : "ImageComponent_tp",
    "file" : "components/ImageComponent.json"
  }
],
```

- Reference to dependencies: For templates with dependencies, add a line to `template-info.json` that refers to templates to include in the dependency. See Create Dependent Templates.
- Reference to the images: These images associated with the app and used in dashboard files. Each image can contain an entry for a condition statement. Specifying a condition enables decisions to be made with variables on whether an image is added at app creation time. Images must be stored as static resources. References to each image must specify a `namespace` attribute if the org uses namespace.

```
"imageFiles" : [
  {
    "name" : "image1",
    "condition" : "${Variables.ShowImage1}",
    "file" : "images/image1.png"
    "namespace" : "${Org.Namespace}"
  },
  {
    "name" : "image2",
    "file" : "images/image2.png"
    "namespace" : "${Org.Namespace}"
  }
],
```

- Reference to the folder file: The file that contains the folder information including `featuredAssets` and `shares` is `folder.json`.

```
"folderDefinition": "folder.json",
```

- Reference to the auto-install file: The file that contains the auto-install hook configuration and the installed app configuration is `auto-install.json`. This file must exist if your template is installed with a managed package and the app is created with an auto-install request.

```
"autoInstallDefinition": "auto-install.json",
```

You can also call a smart wizard from `template-info.json` to perform computations or detection on the user's data. The call to the smart wizard must be backed up with an Apex callback class. The Apex callback class can run the smart wizard before or after the configuration wizard runs and when the creation of the app is kicked off. You call the Apex class from `template-info.json` with something like this:

```
"apexCallback": {
  "namespace": "${Org.Namespace}",
  "name": "GenericConfigurationModifier"
},
```

For more about the smart wizard, refer to the Best Practices section.

SEE ALSO:

template-info.json Attributes

Edit folder.json

The `folder.json` file describes the `featuredAssets` for the application.

Note: This file is required for embedded apps because it specifies the name and label for the templated app along with setting sharing rules for the app. The embedded app can't be shared with other users and groups using CRM Analytics Studio.

Template developers can use it to specify the order of the dashboards in the application instead of accepting the default behavior of alphabetizing by dashboard label.

Example:

```
{
    "name": "My Templated App",
    "featuredAssets": {
        "default": {
            "assets": [
                {"id": "${App.Dashboards['ZDashboard'].Id}"},
                {"id": "${App.Dashboards['BDashboard'].Id}"},
                {"id": "${App.Dashboards['ADashboard'].Id}"}
            ]
        }
    },
    "shares": [
        {
            "shareType": "Organization",
            "accessType": "View"
        }
    ]
}
```

The previous example tells the application to display the "ZDashboard" first, as the most prominent dashboard, rather than the default of A, B, then Z. If a dashboard has a conditional attribute in `template-info.json` that resolves so that the dashboard isn't added at app creation time, a rule must be added to the `template-to-app-rules.json` to remove the dashboard from the `folder.json` file. This rule removes that dashboard entry from the `featuredAssets` list at runtime and prevents app creation from failing with a "Dashboard not found" error. The rule looks like this:

```
{
    "name": "RemoveDashboardZFromFolderJson",
    "condition": "${!Variables.HasDashboardZ}",
    "appliesTo": [
        {
            "type": "folder",
            "name": "*"
        }
    ],
    "actions": [
        {
```

```
            "action": "delete",
            "description": "Remove conditional dashboard",
            "path": "$.featuredAssets.default.assets..[?(@.id=~ /^.DashboardZ.$/i)]"
        }
    ]
}
```

SEE ALSO:

folder.json Attributes

Edit auto-install.json

`auto-install.json` configures the template for the auto-install framework. Always create this optional file if you plan to design your template for auto-installation via a managed package with an auto-install request.

Template developers use this file to specify the configuration of the auto-install framework hook and the app installation. When this file is created, add the `autoInstallDefinition` attribute to the `template-info.json` file.

Example:

```
{
  "hooks": [
    {
      "type" : "PackageInstall",
      "requestName" : "Hook for the Auto-Install Template"
    }
  ],
  "configuration" : {
    "appConfiguration" :{
      "failOnDuplicateNames" : false,
      "autoShareWithLicensedUsers" : true,
      "autoShareWithOriginator" : true,
      "deleteAppOnConstructionFailure" : true,
      "values" : {
        "value1" : "testString",
        "value2" : false
      }
    }
  }
}
```

The example creates an auto-install hook to install the template via a managed package, naming the auto-install request `Hook for the Auto-Install Template`. The supported auto-install hooks are `PackageInstall` only.

The `appConfiguration` specifies that the installed app will:

- not fail creation if an app or assets exist with the same developer name
- be automatically shared with appropriately licensed users
- be automatically shared with the user that created the auto-install request
- be deleted if app creation fails for any reason; for create requests only, it doesn't apply to app upgrades
- be created with the template variable values in the `values` map

Important: Because users don't use the CRM Analytics Studio wizard to create the app, an app `name` attribute is required. To specify the app name, add a `name` attribute in the template's `folder.json` file.

Note: Templates packaged for automatic install can't be dependent templates. The creation of auto-install requests can't guarantee dependent templates are installed in the correct order.

SEE ALSO:

auto-install.json Attributes

Edit org-readiness.json

The `org-readiness.json` file configures the validations to run on a user org to ensure it can create an app successfully.

The org readiness file defines the values to use for validation, the template requirements for the org with success and failure messages to display to the user, and the validation definitions. The evaluated result of each `definition` is stored in the template context as `Readiness['<definition_name>']`.

```
{
  "values" : {
    "Sentiment_Analysis_transformField": {
       "sobjectName": "Case",
       "fieldName": "Description"
     },
     "Sentiment_Analysis_dataset_sourceObjectAdditionalFields" : [ ]
  },
  "templateRequirements": [
    {
      "expression": "${Readiness['Total Accounts'] > Variables.MinimumNumberOfAccounts}",

      "type": "SobjectRowCount",
      "successMessage": "Congratulations! The expected number of accounts matched!",
      "failMessage": "Sorry, you don't have enough accounts, you need at least
${Variables.MinimumNumberOfAccounts} accounts in your org.",
      "tags": [ "Account" ],
      "image": { "name": "sales.png" }
    }
  ],
  "definition": {
    "Total Accounts": {
      "type": "SobjectRowCount",
      "sobject": "Account"
    }
  }
}
```

SEE ALSO:

org-readiness.json Attributes

Edit variables.json

The `variables.json` file describes all the variables used in the `template-info.json`, `ui.json`, and the different `rules.json` files.

`variables.json` includes text for wizard questions, descriptions, and specifications for the answers. Variables also define conditional questions. For example, if you want some questions to appear in the wizard only if an org contains certain data. Or you want to add more specific questions based on the answers to more general wizard questions.

Variables enable the customization of apps; without them, everything is hard-coded. Variables allow the framework to replace tokenized data with customer-specific data.

Variables Syntax

Variables are declared in the following format:

```
{
    "<variableName>" : <variableDefinition>
}
```

Each variable has a unique name. Here are the parts of a variable definition:

- `label`—(Required) Text for label in the wizard, formatted as a question the user selection answers.
- `description`—(Optional) Text for description in the wizard, formatted to help the user select the correct response.
- `defaultValue`—(Optional) Default value for the answer to wizard question.
- `required`—(Optional) Indicates whether an answer to wizard question is required (`true`) or not (`false`). The default is `false`.
- `variableType`—(Optional) The type of the variable. The list of valid variable types is here on page 137. Defaults to `StringType` if not specified.
- `excludes`—(Optional) Defines values to exclude from selections in response to wizard question. The values are a list of comma-separated field names or regex tokens to not show in a picker.
- `excludeSelected`—(Optional) Excludes previously selected values from selections in response to subsequent wizard questions. The default is `false`. If there are two or more pickers with same source and variable type and this field is `true` for both, the selected option is excluded from the list of options in the other picker.

Examples

- Generic variable

```
"<Name of a variable>" : {


    "label" : "<Question to display>",
    "description" : "<Text to help user select/provide the right value>",


    "variableType" : { "type" : "SobjectType" },
    "defaultValue" : { "sobjectName": "User" },


    "required" : true,
    "excludes" : [ "AboutMe", "Division"],
```

```
    "excludeSelected" : true
}
```

- `NumberType` variable with enumerated values

```
"numberExampleByTens" : {
   "label" : "What's the maximum number to use for the offset?",
   "description" : "Choose a value between 10 and 100",
   "defaultValue" : 80,
   "required" : true,
   "variableType" : {
       "type" : "NumberType",
       "min" : 0,
       "max" : 100,
       "enums" : [10,20,30,40,50,60,70,80,90,100]
   }
}
```

- `BooleanType` variable

```
"booleanExample" : {
    "label" : "Please define the boolean parameter?",
    "description" : "Some boolean value.",
    "defaultValue": false,
    "required" : true,
    "variableType" : {
        "type" : "BooleanType",
    }
}
```

- array `SobjectFieldType` variable

```
"sobjectFieldArrayExample" : {
    "label" : "Select the user name fields?",
    "description" : "The user fields to use.",
    "defaultValue" : [
       {"sobjectName" : "User", "fieldName" : "FirstName"},
       {"sobjectName" : "User", "fieldName" : "LastName"},
    ],
    "required" : true,
    "variableType" : {
       "type":"array"
    }
    "itemsType" : {
        "type" : "SobjectFieldType",
        "dataType" : "xsd:string"
     }
    }
}
```

To use this variable call it with the following.

```
${Variables.sobjectFieldArrayExample[0].sobjectName}
${Variables.sobjectFieldArrayExample[0].fieldName}
${Variables.sobjectFieldArrayExample[0]}
```

See additional detail and examples in the sections that follow.

Simple Variable Types for variables.json
We call the following variable types "simple" because they're standard, predefined datatypes.

Complex variables.json Variable Types
Complex variable types are unique to Salesforce, for example, `sobject`, `sobjectField`. Use them to query the org for access to data from Salesforce objects. Scroll to the right to view example values. Examples for each type appear below the chart.

Array Variable Type for variables.json
Use an `ArrayType` variable to create a wizard question that accepts multiple selections. The user can make multiple choices from a list of values. Define a minimum and maximum for the number of items that can be selected.

excludes and excludeSelected Attributes in Variables
In variables, use the `excludes` attribute in variables to define values to exclude from selections in response to wizard questions. Use the `excludeSelected` attribute to exclude values that have already been selected in previous wizard questions.

Use Case and Syntax for Variables with Related Values
Construct variables that reference related values, for example an object and fields from the object, and a dataset and dates, dimensions, and measures from the dataset.

SEE ALSO:
variables.json Attributes

Simple Variable Types for variables.json

We call the following variable types "simple" because they're standard, predefined datatypes.

Type Name	Description	Type Declaration
`BooleanType`	References `true` or `false` values.	`"variableType" : { "type": "BooleanType" }`
`StringType`	References a string. Can use `Array<String>` to define restricted valid values for the string. If null or empty, any value is allowed.	`"variableType" : { "type": "StringType" }`
`NumberType`	References numerical values.	`"variableType" : { "type": "NumberType" }`
`NumberType` (Range)	References range of numerical values. Use `min` to specify minimum value and `max` to specify maximum value. Optional.	`"variableType" : { "type": "NumberType", "min":0, "max":100 }`

Enumerated Values

The `StringType` and `NumberType` can both contain lists of values, `enums`, and an optional array of labels for those values, `enumsLabels`. The order of values in `enumsLabels` must match the order of values in `enums`.

Example:

```
"Colors": {
    "label": "Select a color",
    "variableType": {
      "type": "StringType",
      "enums": [ "#ff0000", "#00ff00", "#0000ff" ],
      "enumsLabels": [ "Red", "Green", "Blue" ]
    }
}
```

Complex `variables.json` Variable Types

Complex variable types are unique to Salesforce, for example, `sobject`, `sobjectField`. Use them to query the org for access to data from Salesforce objects. Scroll to the right to view example values. Examples for each type appear below the chart.

`SobjectType`

References an sObject within your org.

```
"variableType" : {
  "type": "SobjectType"
 }
```

Example: **SobjectType Example**

```
"AccountObjVariable": {
  "label": "What object do you use to represent accounts?",
  "description": "The default is the Account object. If you use a different object,
select it from the list below.",
  "variableType": {
    "type": "SobjectType"
  },
  "defaultValue": {
    "sobjectName": "Account"
  }
}
```

`SobjectFieldType`

References a field within an sObject.

```
"variableType" : { "type": "SobjectFieldType", "dataType":"xsd:double" }
```

The `dataType` limits values for `sObjectField`. For example, specify `xsd:double` to limit the list of fields to numeric sObject fields only. See Primitive Data Types in the *Salesforce Object Reference* for more values for `dataType`.

Example: **SobjectFieldType Example**

```
AccountObjFieldsVariable" : {
  "label" : "Which field in the Account object do you use to track annual revenue?",
  "description" : "The default is the AnnualRevenue. If you use a different field,
select it from the list below.",
  "variableType" : {
    "type" : "SobjectFieldType",
    "dataType" : "xsd:string"
  },
  "defaultValue": {
    "fieldName" : "AnnualRevenue",
    "sobjectName" : "{{Variables.AccountObjVariable.sobjectName}}"
  }
}
```

ConnectorType

References a connector within your org to use for recipes or dataflows. For more information, see Add a Remote Connector to a CRM Analytics Template.

```
"variableType" : { "type": "ConnectorType" , "connectorType" : "SalesforceExternal" }
```

Example: **ConnectorType Example**

```
"ConnectorVariable" : {
  "label" : "Select a connector",
  "description" : "A connector that exists in this org.",
  "variableType" : {
    "type" : "ConnectorType",
    "connectorType" : "SalesforceExternal"
  }
}
```

DatasetType

References a dataset within your org.

```
"variableType" : { "type": "DatasetType" }
```

Example: **DatasetType Example**

```
"DatasetVariable" : {
  "label" : "Select a dataset",
  "description" : "A dataset that exists in this org.",
  "variableType" : {
    "type" : "DatasetType"
  }
}
```

DatasetDimensionType

References a dimension within a dataset. Use to let app creators select dataset dimension fields to populate app dashboards. Requires that a `DatasetType` variable is set so `datasetId` is referenced at runtime.

```
"variableType" : {  "type": "DatasetDimensionType" }
```

Example: **DatasetDimensionType Example**

```
"DimensionVariable" : {
  "label" : "Select a dimension",
  "description" : "A dimension from the selected dataset.",
  "variableType" : {
    "type" : "DatasetDimensionType"
  },
  "defaultValue" : {
    "datasetId" : "{{Variables.DatasetVariable.datasetId}}",
    "fieldName" : ""
  }
}
```

DatasetMeasureType

References a measure within a dataset. Use to let app creators select dataset measure fields to populate app dashboards. Requires that a `DatasetType` variable is set so `datasetId` is referenced at runtime.

```
"variableType" : { "type": "DatasetMeasureType" }
```

Example: **DatasetMeasureType Example**

```
"MeasureVariable" : {
  "label" : "Select a measure",
  "description" : "A measure from the selected dataset.",
  "variableType" : {
    "type" : "DatasetMeasureType"
  },
  "defaultValue" : {
    "datasetId" : "{{Variables.DatasetVariable.datasetId}}",
    "fieldName" : ""
  }
}
```

DatasetDateType

References a date within a dataset. Use to let app creators select dataset date fields to populate app dashboards. Requires that a `DatasetType` variable is set so `datasetId` is referenced at runtime.

```
"variableType" : {  "type": "DatasetDateType" }
```

Example: **DatasetDateType Example**

```
"DateVariable" : {
  "label" : "Select a date",
  "description" : "A date from the selected dataset.",
  "variableType" : {
```

```
    "type" : "DatasetDateType"
  },
  "defaultValue" : {
    "datasetId" : "{{Variables.DatasetVariable.datasetId}}",
    "dateAlias" : ""
  }
}
```

DatasetAnyField

References any type of field within a dataset. Use to let app creators select dataset dimensions, measures, and dates to populate app dashboards from a single list. Requires that a `DatasetType` variable is set so `datasetId` is referenced at runtime.

```
"variableType" : { "type": "DatasetAnyField" }
```

Example: **DatasetAnyType Example**

```
"AnyFieldVariable" : {
  "label" : "Select any field",
  "description" : "Any field from the selected dataset.",
  "variableType" : {
    "type" : "DatasetAnyFieldType"
  },
  "defaultValue" : {
    "datasetId" : "{{Variables.DatasetVariable.datasetId}}",
    "fieldName" : ""
  }
}
```

DateTimeType

References a time and date. Can't be used in non-VisualForce page in ui.json. Value must be an ISO 8601 formatted date time string, as per RFC 3339, section 5. For example, `2020-02-19T14:31:42-0700`.

```
"variableType" : { "type": "DateTimeType" }
```

Example: **DateTimeType Example**

```
"dateExample" : {
  "label" : "Select a date",
  "description" : "The date and time",
  "variableType" : {
    "type" : "DateTimeType"
  }
}
```

ObjectType

References a user-defined object in your org. `ObjectType` isn't supported in the UI and referencing it in ui.json results in app creation errors. Use `ObjectType` for runtime functionality, such as `Overrides`.

```
"variableType" : { "type": "ObjectType" }
```

Example: **ObjectType Example**

```
"ObjectVariable" : {
  "label" : "Define some object...",
  "variableType" : {
    "type" : "ObjectType",
    "properties" : {
      "stringVar" : {
        "type" : "StringType"
      },
      "booleanVar" : {
        "type" : "BooleanType"
      },
      "numberVar" : {
        "type" : "NumberType",
        "min" : 5,
        "max" : 10
      }
    },
    "strictValidation" : true,
    "required" : ["stringVar", "numberVar"]
  }
}
```

DataLakeObjectType

References a data lake object within your org.

`"variableType" : { "type": "DataLakeObjectType" }`

Example: **DataLakeObjectType Example**

```
"DataLakeObjectVariable" : {
  "label" : "Select a data lake object",
  "description" : "A data lake object that exists in this org.",
  "variableType" : {
    "type" : "DataLakeObjectType"
  }
}
```

DataLakeObjectFieldType

References a field within a data lake object.

`"variableType" : { "type": "DataLakeObjectFieldType", "dataType":"number" }`

The `dataType` limits values for `dataLakeObjectField`. For example, specify `number` to limit the list of to numeric fields only. Valid values are `number` `string`, `date`, `date_time`, or null for all values.

Example: **DataLakeObjectFieldType Example**

```
"DataLakeObjectNumberFieldVariable" : {
  "label" : "Select a numeric field from your data lake object.",
  "variableType" : {
```

```
    "type" : "DataLakeObjectFieldType",
    "dataType" : "number"
  },
  "defaultValue" : {
    "objectName" : "{{Variables.DataLakeObjectVariable.objectName}}",
    "fieldName" : ""
  }
}
```

DataModelObjectType

References a data model object within your org.

`"variableType" : { "type": "DataModelObjectType" }`

Example: **DataModelObjectType Example**

```
"DataModelObjectVariable" : {
  "label" : "Select a data model object",
  "description" : "A data model object that exists in this org.",
  "variableType" : {
    "type" : "DataModelObjectType"
  }
}
```

DataModelObjectFieldType

References a field within a data model object.

`"variableType" : { "type": "DataModelObjectFieldType", "dataType":"string" }`

The `dataType` limits values for `dataModelObjectField`. For example, specify `string` to limit the list to string fields only. Valid values are `number` `string`, `date`, `date_time`, or null for all values.

Example: **DataModelObjectFieldType Example**

```
"DataModelObjectStringFieldVariable" : {
  "label" : "Select a text field from your date model object.",
  "variableType" : {
    "type" : "DataModelObjectFieldType",
    "dataType" : "string"
  },
  "defaultValue" : {
    "objectName" : "{{Variables.DataModelObjectVariable.objectName}}",
    "fieldName" : "
  }
}
```

CalculatedInsightType

References a calculated insight within your org.

`"variableType" : { "type": "CalculatedInsightType" }`

Example: **CalculatedInsightType Example**

```
"CalculatedInsightVariable" : {
  "label" : "Select a calculated insight",
  "description" : "A calculated insight that exists in this org.",
  "variableType" : {
    "type" : "CalculatedInsightType"
  }
}
```

`CalculatedInsightFieldType`

References a field within a calculated insight.

`"variableType" : { "type": "CalculatedInsightFieldType", "dataType":"dimension" }`

The `dataType` limits values for `calculatedInsightField`. For example, specify `dimension` to limit the list of dimension fields only. Valid values are `dimension` `measure`, `date`, `date_time`, or null for all values.

Example: **CalculatedInsightFieldType Example**

```
"CalculatedInsightFieldVariable" : {
  "label" : "Select a dimension field from your calculated insight.",
  "variableType" : {
    "type" : "CalculatedInsightFieldType",
    "dataType" : "dimension"
  },
  "defaultValue" : {
    "objectName" : "{{Variables.CalculatedInsightVariable.objectName}}",
    "fieldName ": ""
  }
}
```

Generated `Overrides` in `variables.json`

The `variables.json` file generated upon template creation contains an `Overrides` variable for app assets. `Overrides` are referenced in `template-info.json` as a conditional for whether assets are created at runtime or not. It's best practice to edit `Overrides`. `Overrides` can be removed as long as all references to `Overrides` are also removed from `template-info.json`.

Example: **Overrides Variable Example**

```
{
  "Overrides" : {
    "label" : "Overrides",
    "description" : "Internal configuration to allow asset creation overrides, not to
 be displayed in UI.",
    "defaultValue" : {
      "createAllDashboards" : true,
      "createAllLenses" : true,
      "createAllExternalFiles" : true,
      "createDataflow" : true,
      "createAllDatasetFiles" : true,
      "createAllImages" : true
```

```
  },
  "required" : true,
  "excludeSelected" : false,
  "excludes" : [ ],
  "variableType" : {
    "required" : [
      "createAllExternalFiles",
      "createAllDashboards",
      "createAllImages",
      "createAllDatasetFiles",
      "createAllLenses",
      "createDataflow"
    ],
    "type" : "ObjectType",
    "properties" : {
      "createAllDashboards" : {
        "type" : "BooleanType",
        "enums" : [
          true,
          false
        ]
      },
      "createAllLenses" : {
        "type" : "BooleanType",
        "enums" : [
          true,
          false
        ]
      },
      "createAllExternalFiles" : {
        "type" : "BooleanType",
        "enums" : [
          true,
          false
        ]
      },
      "createDataflow" : {
        "type" : "BooleanType",
        "enums" : [
          true,
          false
        ]
      },
      "createAllDatasetFiles" : {
        "type" : "BooleanType",
        "enums" : [
          true,
          false
        ]
      },
      "createAllImages" : {
        "type" : "BooleanType",
        "enums" : [
          true,
```

```
            false
          ]
        }
      },
      "strictValidation" : true
    }
  }
}
```

Array Variable Type for variables.json

Use an `ArrayType` variable to create a wizard question that accepts multiple selections. The user can make multiple choices from a list of values. Define a minimum and maximum for the number of items that can be selected.

```
"variableType" : {

  "type" : "ArrayType",

  "itemsType" : {

    "type" : "NumberType"

  }
  "sizeLimit"  :{
    "min" : 1,
    "max" : 100
  }

}
```

Example for ArrayType with `enum`:

```
"variableType" : {
  "type" : "ArrayType",
  "itemsType" : {
    "type" : "StringType",
    "enums" : [
      "Leads",
      "Campaigns",
      "Campaign Members"
     ]
  },
  "sizeLimit" : {
    "max" : 3
  }
}
```

excludes and excludeSelected Attributes in Variables

In variables, use the `excludes` attribute in variables to define values to exclude from selections in response to wizard questions. Use the `excludeSelected` attribute to exclude values that have already been selected in previous wizard questions.

"excludes" Attribute

Regex Token Definition: `"/<regex pattern>/flags"`

Flags are characters that Javascript RegEx global object takes, for example: g, i, m, u, and so on. Refer to the RegEx Object API documentation for details.

Example `excludes` statements:

- `"excludes" : ["Name", "/^(Billing).+/"]`—Excludes `Name` field and all the fields that start with `Billing`.
- `"excludes" : ["/^(?!Billing)^(?!Shipping).+/"]`—Excludes all fields except fields that start with `Billing` or `Shipping`.
- `"excludes" : ["/.+(_kav)$/"]`—Excludes all fields that end with `_kav`.
- `"excludes" : ["/^(?:(?!_kav).)*$/"]`—Excludes all fields except fields that end with `_kav`.
- `"excludes" : ["/(?!^Case$|^Account$)(^.*$)/"]`—Excludes all objects except `Case` and `Account`.

Note: Including a value in `defaultValue` that is excluded results in an error.

excludeSelected Attribute

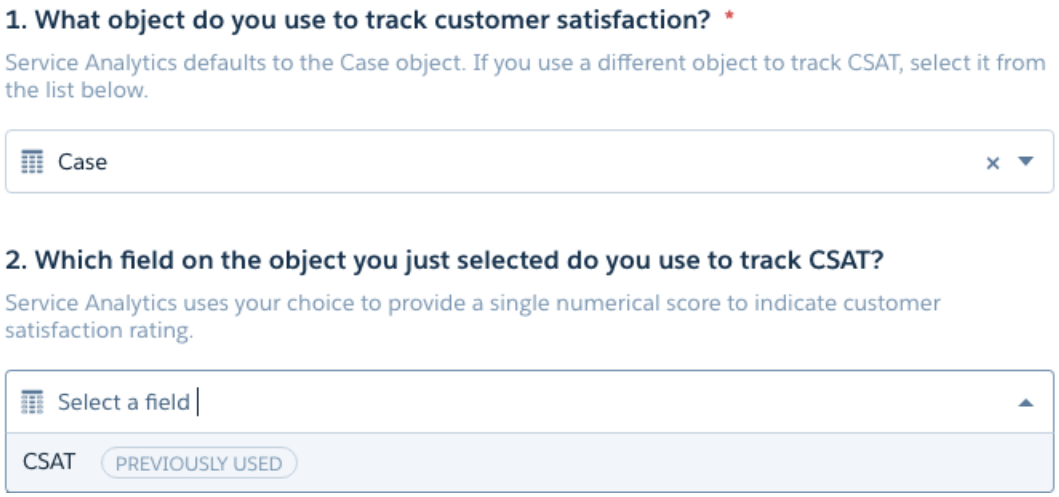
In this example, the option selected in `SObjectField1` picker isn't available in `SObjectField2` picker.

```
"SObjectField1" : {
  "label" : "Select a field from account",
  "description" : "First account field.",
  "defaultValue" : {
    "sobjectName" : "Account",
    "fieldName" : ""
  },
  "required" : true,
  "excludeSelected" : true,
  "variableType" : {
    "type" : "SobjectFieldType"
   }
},
"SObjectField2" : {
  "label" : "Select another field from account",
  "description" : "Second account field.",
  "defaultValue" : {
    "sobjectName" : "Account",
    "fieldName" : ""
  },
  "required" : true,
  "excludeSelected" : true,
  "variableType" : {
    "type" : "SObjectFieldtype"
  }
}
```

Use Case and Syntax for Variables with Related Values

Construct variables that reference related values, for example an object and fields from the object, and a dataset and dates, dimensions, and measures from the dataset.

For example, the first question on a wizard page asks the user to select an object (`sObjectType` variable). The next questions can ask the user to select fields (`sObjectFields` variable) from that object. The wizard populates the questions about fields with `sObjectFields` from the `sObject` selected in the first question. The following image shows an `sObject` picker first, followed by an `sObjectField` picker.



You can take the same approach with datasets and dataset dates, measures, and dimensions. The first question references a dataset (`datasetType` variable). Subsequent questions reference dates (`datasetDateType`), dimensions (`datasetDimensionType`), and measures (`datasetMeasureType`) from the dataset selected in the first question.

Following is template JSON used to render the UI shown in the image, using `sObject` and `sObjectFields` variables. In the `variables` section, notice `CSATField` variable's type info: the `defaultValue` contains a reference to the variable's value `CSATObj` and the syntax is `{{Variables.CSATObj.sobjectName}}`. The intention here's to replace `{{Variables.CSATObj.sobjectName}}` with the value that the user picked for the variable `CSATObj`.

The `ui.json` file for the template, shown at the top of the example, orders the wizard questions defined in `variables.json`.

```
{
  "ui" : {
    "pages" : [
      {
        "title ": "Page1",
        "variables" : [
          {
            "name" : "CSATObj"
          },
          {
            "name" : "CSATField"
          }
        ]
      }
    ]
  },
  "variables" : {
    "CSATObj" : {
     "description" : "Service Analytics defaults to the Case object. If you use a different
 object to track CSAT, select it from the list below",
      "label" : "1. What object do you use to track customer satisfaction?",
```

```
      "required" : true,
      "variableType" : {
        "type" : "SObjectType"
      },
      "defaultValue" : {
        "sobjectName" : "Case"
      }
    },
    "CSATField" : {
      "description" : "Service Analytics uses your choice to provide a single numerical
score to indicate customer satisfaction rating.",
     "label" : "2. Which field on the object you just selected do you use to track CSAT?",

      "defaultValue" : {
        "datasetId" : "{{Variables.CSATObj.sobjectName}}",
        "fieldName" : ""
      },
      "required" : true,
      "excludeSelected" : true,
      "variableType" : {
        "type" : "SobjectFieldType",
        "datatype" : "xsd:double"
      }
    }
  }
}
```

Variable Value Reference Syntax

```
"${Variables.Dataset1.datasetId}
${Variables.<SObjectVariableName>.sobjectName}"
```

Edit ui.json

The `ui.json` file determines how your template displays the configuration wizard questions defined in `variables.json`.

Note: This file isn't used in an embedded app. Templates with a `templateType` of `embeddedapp` aren't visible in CRM Analytics Studio for install.

The UI file determines wizard page layout, defining the order in which variables described in `variables.json` appear on a page.

```
{

"ui" :{

  "pages": [
    {
      "title":"Customization Questions",
      "variables":[

        { "name": "customizeUserObjectInfo" }
```

```
      ]
    }
,
    {

      "title":"User Object Information",
      // Only show this page if the user checked the "customizeUserObjectInfo" checkbox.

      "condition":"{{Variables.customizeUserObjectInfo == 'Yes'}}",

      "variables":[

        { "name": "userObjectUsernameField" },

        { "name": "userObjectName" },
        { "name": "pageConditionVariable" },

        { "name": "showBoolean" }
      ],
      "helpUrl":"https://salesforce.com/wave/salesapp/page1/help.html"

    },

    {

      "title":"Job Information",

      "variables":[

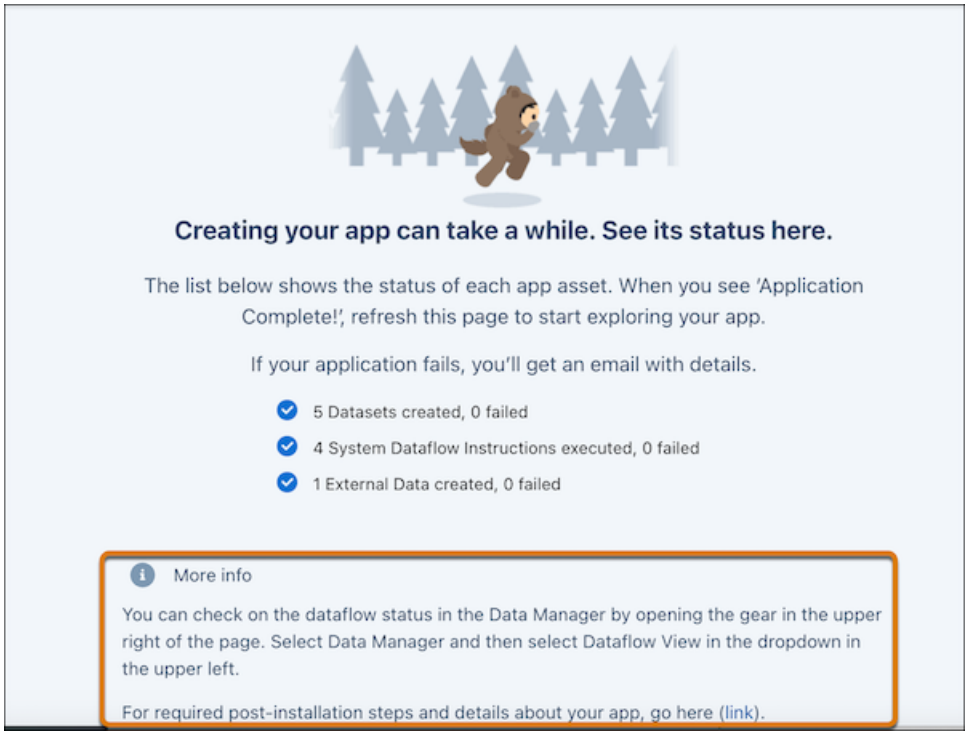
        { "name": "maxAllowedOffset" },
        { "name": "booleanExample", "visibility": "{{Variables.showBoolean ? 'Visible' :
'Hidden'}}" },
      ],
      "helpUrl":"https://salesforce.com/wave/salesapp/page2/help.html"

    }

  ],
  "displayMessages": [
    {
      "text":"When we're done creating the app, we'll send you an email.",
      "location": "AppLandingPage"
    }
  ]

}
```

The displayMessages text is visible to the user on the page that displays the app creation progress.



Page Condition Syntax for ui.json

You can make a page's appearance conditional on the value of a variable. For example, if the user answers "No" when asked if there is a Products dimension, you can use a conditional to ensure that no Products-related page displays in the wizard. Adding page conditions enables wizard page flow. If the condition is met, then the page with the condition will display; if the condition is not met, the page will not display. Use the following syntax to instruct which conditionals control the display of pages.

Variables Array in ui.json

Use a `variables` array to display a question to users conditionally. You can optionally display a question as disabled in the case where an answer has been found or computed using Apex Class. To accommodate variable condition and other fields, the `variables` array must be an array of objects instead of an array of variable names.

Use VisualForce To Customize the Wizard UI

To customize wizard appearance beyond the default interface, use VisualForce and a JavaScript library.

SEE ALSO:

ui.json Attributes

Page Condition Syntax for ui.json

You can make a page's appearance conditional on the value of a variable. For example, if the user answers "No" when asked if there is a Products dimension, you can use a conditional to ensure that no Products-related page displays in the wizard. Adding page conditions enables wizard page flow. If the condition is met, then the page with the condition will display; if the condition is not met, the page will not display. Use the following syntax to instruct which conditionals control the display of pages.

Variable Type	Supported Operations	Example
`StringType`	`==, !=`	`{{Variables.x == 'Yes'}}`
`BooleanType`	`==, !=`	`{{Variables.x == true}}`
`NumberType`	`==, !=, <, <=, >, >=`	`{{Variables.x == 5}}`
`SobjectType`	`==, !=`	`{{Variables.x.sobjectName == 'x'}}`
`SobjectFieldType`	`==, !=`	`{{Variables.x.fieldName == 'x'}}`
`ArrayType --> StringType`	`contains`	`{{Variables.x contains 'x'`123}}`
`ArrayType --> NumberType`	`contains`	`{{Variables.x contains 5}}`

Variables Array in ui.json

Use a `variables` array to display a question to users conditionally. You can optionally display a question as disabled in the case where an answer has been found or computed using Apex Class. To accommodate variable condition and other fields, the `variables` array must be an array of objects instead of an array of variable names.

Variable Object

A variable object can have following fields:

```
{
   "name": "userObjectUsernameField",

   "visibility": "{{Variables.orgHasUserObject == 'Yes'}}"
}
```

Key	Required	Value Description
`name`	Yes	Name of the variable.
`visibility`	No	• Display condition: – If the condition is evaluated to `'true'`, then this variable (question+input control) is displayed.

Key	Required	Value Description
		– If evaluated to `'false'`, this variable won't be displayed. – If evaluated to `'disabled'`, this variable is grayed out. • This is evaluated on the client side. • Valid values `'Disabled' | 'Visible' | 'Hidden'` • Example Expressions: – `"{{Variables.booleanType ? 'Visible' : 'Hidden'}}"` – `"{{Variables.orgHasUserObject == 'Yes' ? 'Visible' : 'Hidden'}}"` – `"{{Variables.orgHasUserObject == 'Yes' ? 'Disabled' : 'Visible' }}"`

Example: Example of variable object:

```
{

    "ui" :{

        "pages": [

            {

                "title":"User Object Information",
                // Only show this page if the user checked the "Customize" button.

                "condition":"{{Variables.customizeOpportunities == 'Yes'}}",

                "variables":[
                    {
                        "name": "userObjectUsernameField",

                     "visibility": "{{Variables.orgHasUserObject == 'Yes' ? 'Visible'
 : 'Hidden'}}"
                    },
                    {
                        "name": "userObjectName",

                     "visibility": "{{Variables.orgHasUserObject == 'Yes' ? 'Visible'
 : 'Hidden'}}"
                    }
```

```
                ],
                "helpUrl":"https://salesforce.com/wave/salesapp/page1/help.html"

            },

            {

                "title":"Job Information",

                "variables":[

                    "maxAllowedOffset"
                ],
                "helpUrl":"https://salesforce.com/wave/salesapp/page2/help.html"

            }
        ],
        "displayMessages": [
             {
               "text":"When we're done creating the app, we'll send you an email.
Before you access the dashboards, you need to wait for the dataflow for the app to
finish running. After that, refresh this page and start exploring.",
   "location": "AppLandingPage"
             }
         ]
    }
}
```

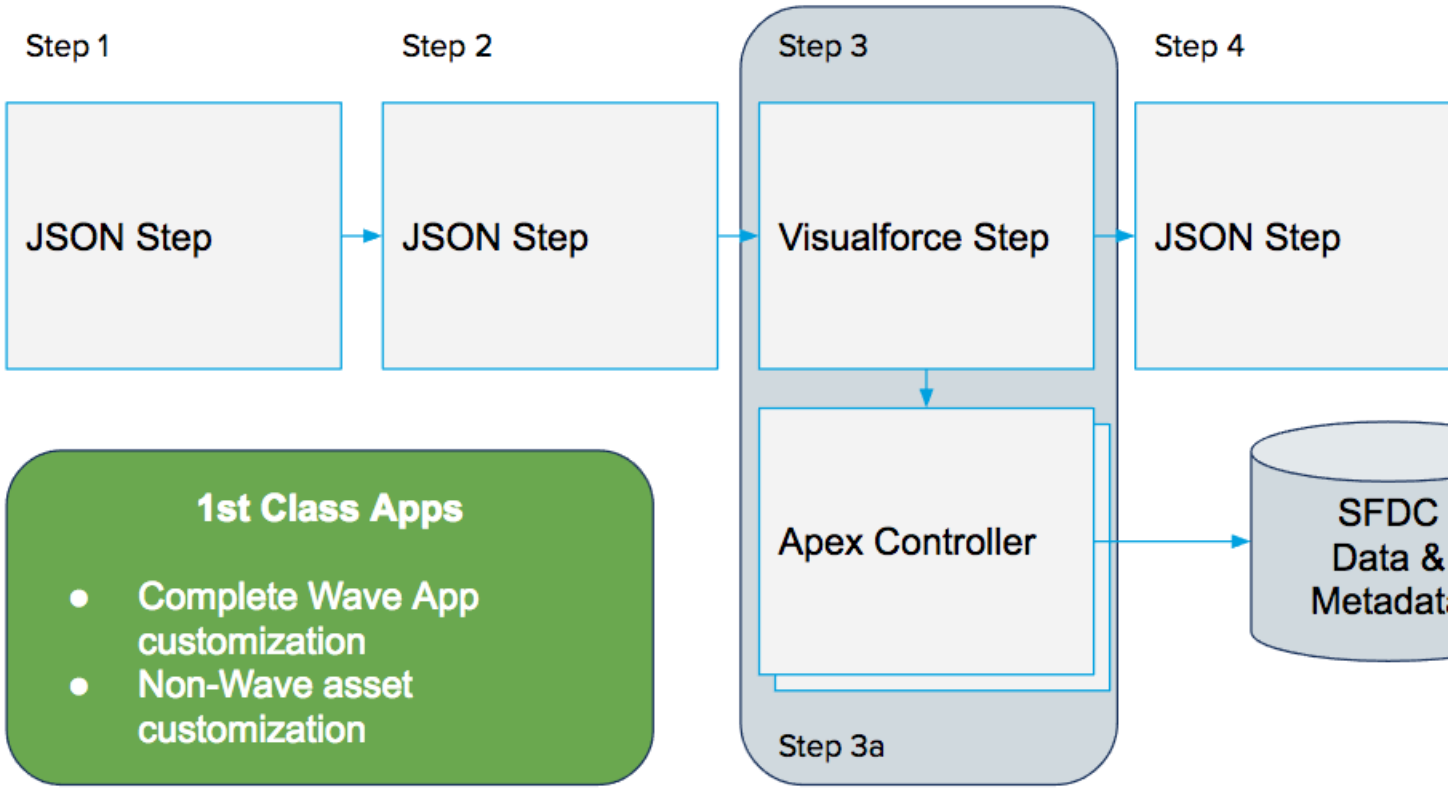
Use VisualForce To Customize the Wizard UI

To customize wizard appearance beyond the default interface, use VisualForce and a JavaScript library.

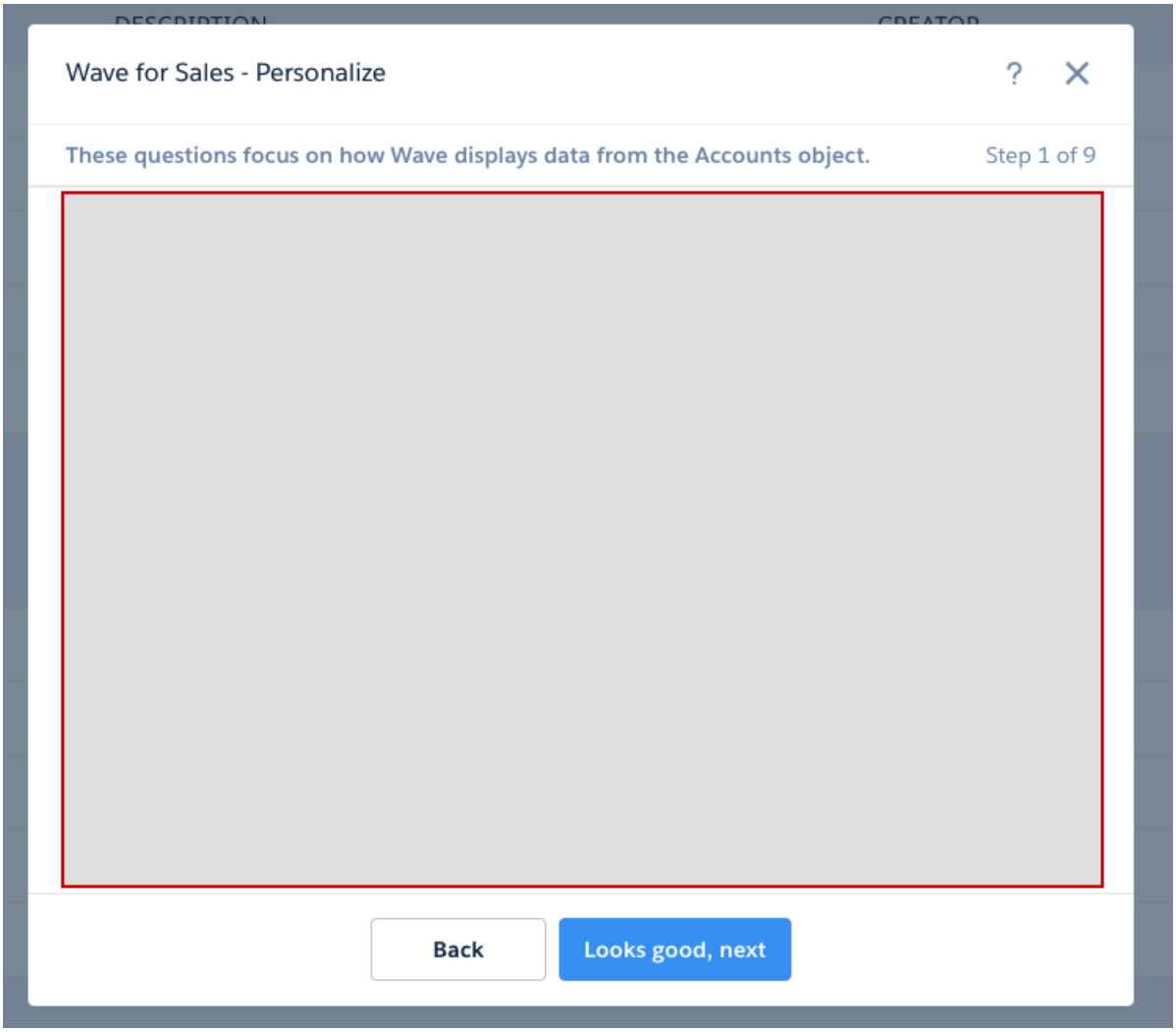
CRM Analytics runs the wizard before generating the app from the template. The wizard consists of a set of pages, each of which contains questions or other elements you use to set up the app.

The framework lets you run a custom APEX class at any time during app creation. This class can answer questions about the org automatically or validate the org to ensure it meets the requirements to create the app. CRM Analytics uses a default format for the error messages generated by the validation process. VisualForce and a JavaScript library lets you customize the format and include links to help the admin edit the org to meet app-creation requirements.

VisualForce lets you replace any or all of the autogenerated wizard pages with a custom VisualForce page that can display anything you want. The illustration shows how you can integrate a custom VisualForce page into the app creation flow.



The VisualForce renders everything inside the red lines in the example wizard shown here. It interacts with the wizard through a JavaScript SDK.



The JavaScript SDK is essentially be a pub/sub eventing mechanism with a set of events fired and consumed during a well-defined lifecycle. The SDK is versioned and can be included in any VisualForce page as follows:

```
<script type="text/javascript"
src="/analytics/wave/sdk/js/<version_number>/wizard.js"></script>
```

Here's example code for the VisualForce page:

```
// Subscribe to event that gets fired when the page first loads
Wave.wizard.publish(
   {name : 'wizard.ready', "payload" : {}, callback : function (response) {
var payload = response.payload;
var metadata = {
   page : payload.page,
   variableDefinitions : payload.variableDefinitions,
   values : payload.initialValues
}
   }});

// Update and validate a single answer to a question
```

```
Wave.wizard.publish({
   name: "wizard.update",
   payload: {name: "Fiscal_Month", value: "01 - January"},
   function(response) {
       var errors = response.payload;
   }
});
```

Declaring the VisualForce Page in ui.json

In order to replace the default rendering of a wizard page the application developer simply specifies the Visualforce name in `ui.json` page as follows:

```
   "pages" : [
   {
       "title": "These questions focus on how CRM Analytics displays data for Accounts”,
"variables": [
           { "name": "Customer" },
           { "name": "Customer_L2" },
           { "name": "Source" },
           { "name": "Geo_Maps" },
           { "name": "Geography" },
           { "name": "Geography_L2" }
       ],
       "vfPage":{
           "namespace": "<visualforce_namespace>",
           "name": "<visualforce_page_name>"
       }

   }]
```

Important: Make sure any Visualforce page included in a class can be accessed by users in the organization with Manage Analytics Templated Apps user permission. Otherwise, trying to create an app from your template results in an error. In Setup, go to **Visualforce Pages**, click **Security** next to the page, then add the profile with the Manage Analytics Templated Apps user permission.

Refer to the "VisualForce Events for Customizing the Wizard UI" reference section at the end of this guide for details on the events to which you can subscribe or publish.

Edit Rules Files

Rules files define how an app gets created by the template.

There are two type of rules files: `template-to-app-rules.json` and `app-to-template-rules.json`. They use the same syntax, but serve different purposes.

- `template-to-app-rules.json` defines the rules that the template follows when creating an app. For example, you can define a rule that specifies that if an org doesn't use certain objects, the app doesn't use them in dashboards or the dataflow. Another rule can replace a label name in a chart with the name of a field from a Salesforce object selected in a wizard question. Rules also define how variables are handled. For example, if the wizard asks which fields to include in filters for accounts, `template-to-app-rules.json`determines how that choice is reflected in dashboards.
- `app-to-template-rules.json` provides rules for converting an updated source app back to template form. After changing the source app, you update the template object with a PUT call to the CRM Analytics REST API. For example, you can tokenize all

asset files every time they are pulled out of the source app. You use app-to-template rules infrequently, because you're unlikely to update the source app often.

The rules file is referenced from `template-info.json`:

```
"rules" : [
    { "type" : "appToTemplate", "file" : "appToTemplateRules.json" },
    { "type" : "templateToApp", "file" : "templateToAppRules.json" }
],
```

The bulk of what follows details the use of `template-to-app-rules.json`.

SEE ALSO:

rules.json Attributes

Rules Files Structure

Rules files can contain three sections: rules, constants, and macros.

- Rules. This is where you define the rules followed by the template for creating the app.
- Constants. Use constants to create shortcuts for longer expressions. Constants are like variables, but are not passed in or declared in the UI. Constants can be referenced in the document via the normal expression language `${Constants.<constantName>}`.
- Macros. Rule macros let you define repeatable, well-tested rule code units that can be called within the context of any JSON transformation, simplifying and making the rule definition source easier to maintain

Structuring and Organizing Rules

The rules section of `template-to-app-rules.json` defines the rules followed by the template.

The section is made up of three parts:

- Action: Designates the operation to perform.

- Path: Defines the path to the location where the action is performed.
- AppliesTo: Determines the asset files to apply the rules to. Consists of a name.

Rules can be divided into multiple files, which are executed in the order in which you list them in `template-info.json`. Be sure that a rule that runs first doesn't refer to a constant referenced in a rule that runs later.

Here are a few guidelines to consider when working with multiple rules files:

- One rules file can contain all constants, another all actions.
- Use multiple rules when actions grow to 500 lines. You could divide them into a rules file for dashboards, another for datasets or the dataflow.
- If you have content to be used by many rules, include it in the first rules file to be executed.

Here is a simple example, a rule for changing the title text on a dashboard:

```
{
    "name": "Update dashboard title",
    "appliesTo": [
        {
            "type": "dashboard",
            "name": "DashboardOne"
        }
    ],
    "actions": [
        {
            "action": "set",
            "description": "change the title text of the dashboard",
            "path": "$.state.widgets.title_text.parameters.text",
            "value": "${Variables.DashboardTitle}"
        }
    ]
}
```

Make rules more powerful using functions. See Use Functions in Rules.

Rules Syntax

The rules object may contain any number of rule objects. A rule object is made up of:

Attribute	Type	Example	Required	Notes
`name`	String	`"name": "ruleName"`	Yes	
`condition`	String	`"condition": "${Variables.foo == "Yes"}"`	No	Use freemarker conditions to apply the rule (same syntax as conditions used in `template-info.json`)
`appliesTo`	Array	`{ "type": "dashboard",`	Yes	Use `"type": "*"` to apply to all JSON assets. Other valid values are

Attribute	Type	Example	Required	Notes
		`"name": "dashboardOne"`		`"dashboard"`, `"lens"`, `"workflow"`, `"schema"`, and `"xmd"` The `"name"` string allows specific JSON assets to be referenced The `"name"` can take `"*"` as well, to apply to all of one asset type or a group of one asset type Note: NOTE: for workflow type, name has to be `"*"`
`label`	String			
`actions`	Array	`{ "action": "set", "description" : "A desc", "path": "$.json.path", "value": "setValue" }`	Yes	4 actions types: `add`, `put`, `set`, and `delete` Can have as many actions as needed in array

Actions in Rules

Actions define what a rule does. Use constants to make it easier to perform any action.

Actions

Actions define the way rules change an app's JSON files. Valid actions are:

- `delete`: Delete a node in the document.
- `put`: Add a node to an object.
- `set`: Set the value of an existing node in the document.
- `add`: Add an element to an array
- `eval`: Evaluate a specified expression and assign results to a variable.
- `replace`: Search and replace a text string globally.

Each action has a path attribute, which is a JsonPath that points to a node in the document. JsonPath is to JSON as XPath is to XML. Practice it in the Jayway JsonPath Evaluator.

Note: If a template is updated to change the `assetVersion`, it can impact the JsonPath. Always test the rules for app creation after a template update.

Actions can be conditionally applied to a document. Set the condition on a set of actions or on a collection of actions.

Actions Syntax for Rules

Action	Example	Notes
`add`	`{ "action": "add", "description" : "Add desc", "path": "$.json.path.to.existing.json,array", "index": 0, "value": "value to add" }`	Adds an entry to an existing array. Needs `"index"` and `"value"`. `"index"` should be 0 for first element in array, any number after that for middle of array. If `"index"` is larger than array size, value is added to end of array.
`put`	`{ "action": "put", "description" : "Put desc", "path": "$.json.path.to.existing.json", "key": "jsonAttributeName", "value": "new value" }`	Add an attribute and value to an existing JSON value. Needs `"key"` and `"value"`. `"value"` can be a string or an array.
`set`	`{ "action": "set", "description" : "Set desc", "path": "$.json.path.to.attribute", "value": "setValue" }`	Set a value on an existing JSON attribute. Needs `"value"`.
`delete`	`{ "action": "delete", "description" : "Delete desc", "path":`	Remove a node in the existing JSON tree.

Action	Example	Notes
	`"$.json.path.to.delete"` `}`	

`eval`	`[`
	`{`
	`"action": "eval",`
	`"key": "helloResult",`
	`"value": "${myMacros:sayHello('Hello')}"`
	`},`
	`{`
	`"action": "set",`
	`"path": "$.path.json.node",`
	`"value": "${Rules.Eval.helloResult}"`
	`}`
	`]`
		Evaluate the expression specified in `'value'` and assign results to the context variable specified by `'key'` (if set). Context attribute can be referenced using the expression: `${Rules.Eval.<variable_name>}`.
		Scope of the eval variable is one of the following:
		• The entire document if performed in a rule.
		• The macro if performed within a macro.

`replace`	`{`
	`"action": "replace",`
	`"key": "RPLCM_Opportunity_Amount_Field",`
	`"value": "${Variables.Opportunity_Amount.fieldName}"`
	`},`
		Replace any text string throughout entire document. Use when regular expressions are not an option. Can still use expressions.
		Instead of using the `'replace'` action, you can perform targeted search and replace (JSON Path) with `${string:replace(...)}`.
		In cases requiring regular expressions, use `${string:replaceFirst(...)}`, `${string:replaceAll(...)}`.

Use Functions in Rules

Functions make rules more powerful. For example, use functions to apply rules iteratively to arrays or to make rules conditional on the content of a string or an array. CRM Analytics Templates supports string, array, and json functions as well as static math and sfdc_fma functions.

Use the `"Rules.CurrentNode"` property in conjunction with functions. It contains the last results of the JSON `'path'` argument supplied in the action of your rule.

Use the EL Expression Language to invoke custom functions with an expression. Functions can appear in the static text of an EL expression, for example:

```
"name" : "my name in lowercase is
    ${string:toLowerCase(Variables.myName)}",
```

See the sections that follows for complete information about each function type.

`string` Functions

Use `string` functions in rules to manipulate text strings in asset JSON at runtime.

For instance, a string function can replace all occurrences of one string with another or convert strings from uppercase to lowercase. If a user enters a dashboard title in the wizard using all lowercase letters, the string function changes the text to sentence case at runtime.

`toUpperCase`

Parameters: Object

Returns: Object

Recursive: Supported

Description: Converts all characters in an object to uppercase. The object can be a single string or a JSON object with multiple strings

Example: This JSON

```
    "objecttest" : {
        "obj" : {
            "greeting1" : "hello world",
            "greeting2" : "salut world",
            "greeting3" : "hallo world"
        },
        "list" : ["hello world", "salut world", "hallo world"]
    },
```

And the rule:

```
{
    "action": "set",
    "description": "Convert all strings to upper case",
```

```
    "path": "$.objecttest",
    "value" : "${string:toUpperCase(Rules.CurrentNode)}"
}
```

Returns:

```
"objecttest" : {
        "obj" : {
            "greeting1" : "HELLO WORLD",
            "greeting2" : "SALUT WORLD",
            "greeting3" : "HALLO WORLD"
        },
        "list" : ["HELLO WORLD", "SALUT WORLD", "HALLO WORLD"]
    },
```

`toLowerCase`

Parameters: Object

Returns: Object

Recursive: Supported

Description: Converts all characters in a string to lowercase.

`replace`

Parameters: Object `obj`, String `oldStr`, String `newStr`

Returns: Object

Recursive: Supported

Description: Returns a new object with all occurrences of `oldStr` replaced with `newStr`.

`replaceAll`

Parameters: Object `obj`, String `regex`, String `newStr`

Returns: Object

Recursive: Supported

Description: Replaces all substrings of this string that match the given regular expression.

`replaceFirst`

Parameters: Object `obj`, String `regex`, String `newStr`

Returns: Object

Recursive: Supported

Description: Replaces the first substring of this string that matches the given regular expression.

`split`

Parameters: Object `obj`, String `regex`

Returns: Object

Recursive: Supported

Description: Parses a string and splits it into an array defined by the specified delimiter. For example:

```
${string:split('one:two:three', ':')}
```

Results in array of strings:

```
['one','two','three']
```

If more complex regular expression matching is required, use `match` (next).

`match`

Parameters: Object `obj`, String `regex`

Returns: Object

Recursive: Supported

Description:

Performs complex regular expression matching and returns an array of matching results. For example, to capture items surrounded by square single quote and square brackets:

```
${string:match('['foo']['bar']['baz']', '\\\\[\\'(.*?)\\'\\\\]')
```

Returns the following array:

```
foo

bar
baz
```

`join`

Parameters: Object `obj`, String delimiter

Returns: Object

Recursive: Supported

Description: Creates a string from an array of strings and separates each item by the specified delimiter.

```
${string:join(<string>, <delimiter>)}
```

`array` Functions

Use `array` functions in rules to manipulate arrays in asset JSON at runtime.

Array functions can use the answer to a configuration wizard question to set runtime conditions. For example, array functions can check all selected values and set a condition to true or false. They can also take multiple values from an answer and iterate over a JSON array to put those values into the runtime JSON correctly.

Or, say that your wizard contains multiple questions with limited values to choose from, such as Yes/No, or a time frame in days or months. An array function can replace them with one question that asks the user to select multiple features to include in dashboards. In this case, the array function loops over the one question to check for values and set the features that the user picked via a conditional.

Function Name: `forEach`

Parameters: `Collection<Object>` array, String `returnValue`

Returns: `Collection<Object>`

Recursive: Not supported>

Description: Evaluates returnValue as an EL expression for each item in the given array and collects each eval result in an array and returns it.

Example: JSON template:

```
{
    "fields": "${array:forEach(Variables.myArray, '{\"name\": \"${var}\"}')}"
}
```

Produces:

```
{
    "fields": [{"name": "ProductName"},
               {"name": "Amount"},
               {"name": "CompanyName"},
               {"name": "Revenue"}]
};
```

`forEachIndex`

Parameters: Int start, Int end, String `returnValue`

Returns: `Collection<Object>`

Recursive: Not supported

Description: Evaluates `returnValue` as an EL expression for the index number(s) int the array where the string is found.

`concat`

Parameters: `Collection<Object>` array1, `Collection<Object>` array2

Returns: `Collection<Object>`

Recursive: Not supported

Description: Returns an array after concatenating the given array1 and array2.

`unique`

Parameters: `Collection<String>` array

Returns: `Collection<String>`

Recursive: Not supported

Description: Returns an array of unique items. Uniqueness determined by the underlying objects contained in the array. For example, if the collection is an array of strings, all strings with identical values (case sensitive) are removed.

`uniqueBy`

Parameters: String `byFieldName`, `Collection<Object>` array

Returns: `Collection<Object>`

Recursive: Not supported

Description: Returns an array of unique items. Expects an array of complex objects of similar shape. Uniqueness determined by comparing the value of a field identified by the given `byFieldName`.

union

Parameters: `Collection<Object>` array1, `Collection<Object>` array2

Returns: `Collection<Object>`

Recursive: Not supported

Description: Returns a unique union between `array1` and `array2`. Uniqueness determined by the underlying objects contained in the array.

unionBy

Parameters: String `byFieldName`, `Collection<Object>` array1, `Collection<Object>` array2

Returns: `Collection<Object>`

Recursive: Not supported

Description: Returns an array which is a union of array1 and array2 items. Expects an array of complex objects of similar shape. Uniqueness determined by comparing the value of a field identified by the given `byFieldName`.

contains

Parameters: Collection `<Object>` array, Object item

Returns: Boolean

Recursive: Not supported

Description: Returns `true` if the given item is in the given array, otherwise `false`.

size

Parameters: Collection `<Object>` array

Returns: Integer

Recursive: Not supported

Description: Return the size of the array.

last

Parameters: Collection `<Object>` array

Returns: Object

Recursive: Not supported

Description: Return the last item of the array.

Use Array Functions for Multi-Select Widgets and Looping

Here's how to use an array function to create a configuration wizard widget that lets the user select multiple values.

The following code

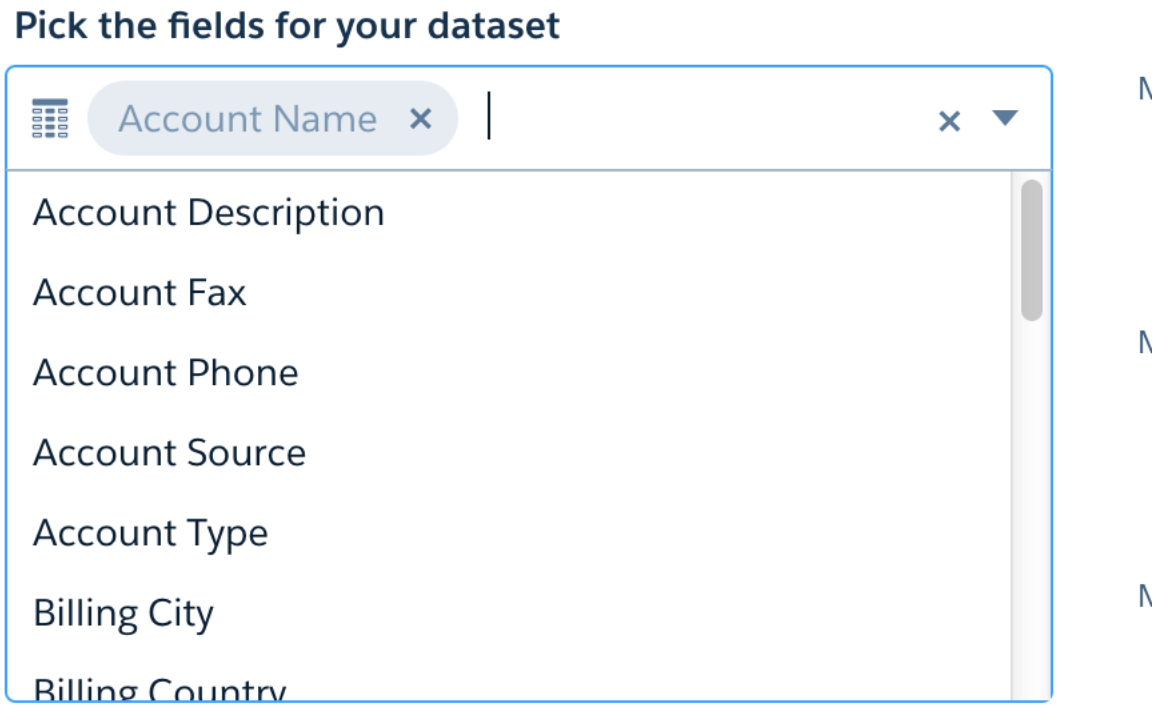
```
"Account_Fields": {
        "label": "Pick the fields for your dataset",
        "description": "Multiselect sobjectfield test",
        "defaultValue": [
          {
            "sobjectName": "Account",
            "fieldName": "Name"
          }
        ],
        "required": false,
        "variableType": {
            "type": "ArrayType",
            "itemsType": {
                "type": "SobjectFieldType"
            }
        },
        "excludeSelected": true
    }
```

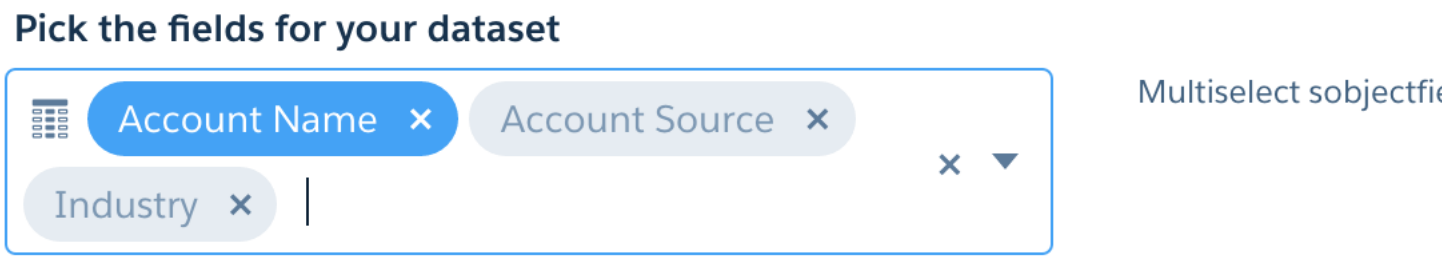
Results in the following wizard widget:

The `ArrayType` variable contains an `itemsType` attribute, which accepts the following:

- `SObjectType`
- `SObjectFieldType`
- `DatasetType`
- `DatasetDateType`
- `DatasetDimensionType`
- `DatasetMeasureType`
- `StringType`, or
- `NumberType`

The widget places values in an array. The following show how the widget displays values for the user to select:





Items in the array can be referenced by index. Arrays of `StringType` and `NumberType` are easy to work with, but arrays of `SObjectType` and `SObjectFieldType` contain multiple attributes for each item in the array:

Array of `SobjectFieldType`:

- `${Variables.Account_Fields[0]}` returns (`sobjectName`=Account, `fieldName`=Name, `sobjectLabel`=Account, `fieldLabel`=Account Name)
- `${Variables.Account_Fields[0].fieldName}` returns Name
- `${Variables.Account_Fields[0].fieldLabel}` returns Account Name
- `fieldName` is required for dataflow

The selection values from the widget are stored in the variable and there are two ways to access them in the template files (rules, dashboards, workflow):

- Use array values from `SobjectFieldType` without looping, accessing by index

```
"fields":[
        {"name":"${Variables.Account_Fields[0].fieldName}"},
        {"name":"${Variables.Account_Fields[1].fieldName}"},
        {"name":"${Variables.Account_Fields[2].fieldName}"},
        {"name":"${Variables.Account_Fields[3].fieldName}"},
        {"name":"${Variables.Account_Fields[4].fieldName}"}
    ]
```

  This example, without looping, would be tricky to use, because there would always need to be at least five items in the array. Any fewer would fail, and anything over five would not be used.

- Use array values from `SobjectFieldType` with looping via the `array:forEach` function. In this example, a `"set"` action is going to loop through the `SObjectFieldTypes` in the array and add them to the `"fields"` attribute in the `Extract_Account` step of the workflow. Each entry in `"fields"` will need "name" and `"var.fieldName"`.

- – `rules.json`

```
"name": "AddAccountFilesToDataflow",
  "appliesTo": [
    {
      "type": "workflow",
      "name": "*"
    }
  ],
  "actions": [
    {
      "action": "set",
      "description": "use selected values for sfdcDigest in dataflow",
      "path": "$.workflowDefinition.Extract_Account.parameters.fields",
      "value": "${array:forEach(Variables.Account_Fields, '{\"name\":
\"${var.fieldName}\"}')}"
    }
  ]
```

  – `dataflow.json`

```
"Extract_Account":{
   "action":"sfdcDigest",
   "parameters":{
       "fields":[],
       "object":"Account"
   }
}
```

  – Results after processing if Account Name, Account Source, and Industry are selected

```
"Extract_Account":{
    "action":"sfdcDigest",
    "parameters":{
        "fields":[
            {"name":"Name"},
            {"name":"AccountSource"},
            {"name":"Industry"}
        ]
```

```
    }
]
```

Here are more ways to work with arrays that extend the power of the `array:forEach` function:

- `array:union` (only uses unique values, so no duplicated values)

```
"value": "${array:union(array:forEach(Variables.Account_Fields,
 '{\"name\": \"${var.fieldName}\"}'),array:forEach(Variables.Account_Fields2,
 '{\"name\": \"${var.fieldName}\"}'))}"
```

- `array:concat` (2 new arrays)

```
"value": "${array:concat(array:forEach(Variables.Account_Fields,
'{\"name\": \"${var.fieldName}\"}'),array:forEach(Variables.Account_Fields2,
 '{\"name\": \"${var.fieldName}\"}'))}"
```

- `array:concat` (1 new array with existing array)

```
"value":"${array:concat(Rules.CurrentNode,
 array:forEach(Variables.Account_Fields,
 '{\"name\":\"${var.fieldName}\"}'))}"
```

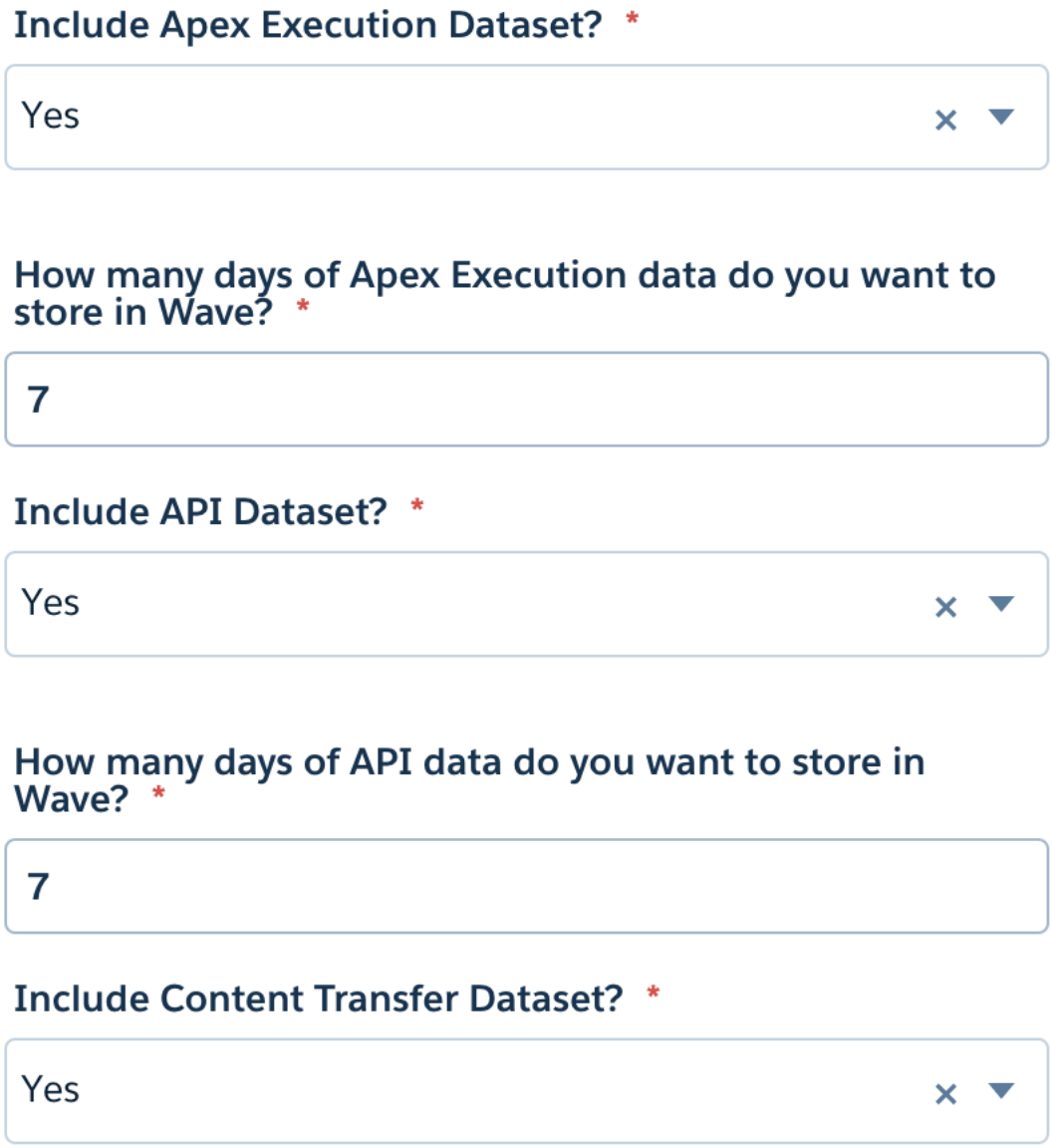
- You can use array functions in the `"set"` and `"put"` actions, but not in the `"add"` or `"delete"`.
  - In `"add"`, the value is being added to an existing array, so the result is an array inside an array, which is not well-formed json. To add more array values to an existing array, use `"set"` with the `array:concat` function
  - For `"delete"`, value is not used
- Use array values from `StringType` (same for `NumberType`)
  - `variables.json`

```
"Account_Fields_String": {
        "label": "Pick string fields for your dataset",
        "description": "Multiselect string test",
        "defaultValue": ["Name"],
        "required": true,
        "variableType": {
            "type": "ArrayType",
            "itemsType": {
                "type": "StringType",
                "enums": ["Name", "AccountSource", "Industry", "Type"]
            }
        }
    }
```

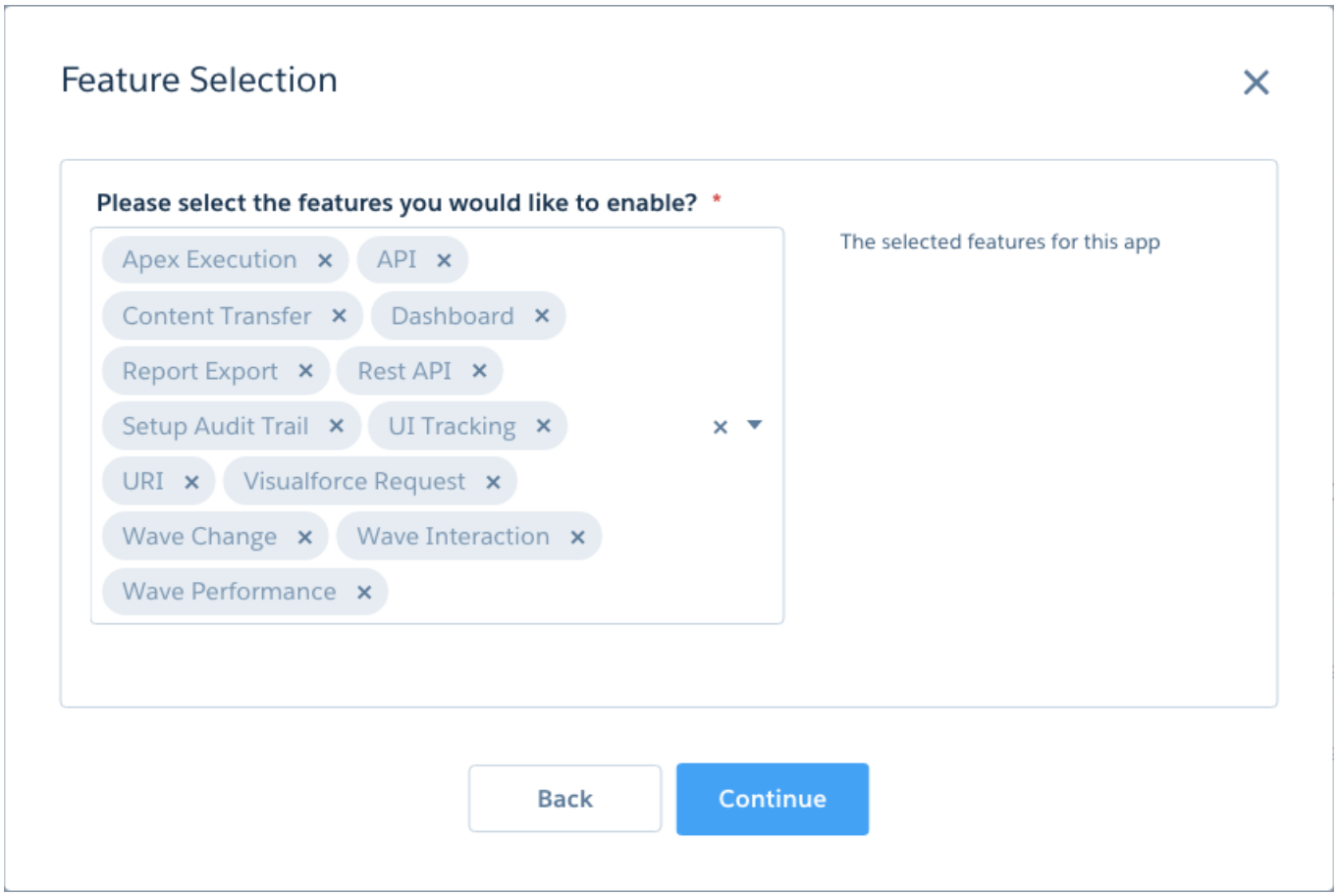
  - `rules.json`

```
"value": "${array:forEach(Variables.Account_Fields_String,
 '{\"name\":
        \"${var}\"}')}"
```

- You can also use the multiselect widget to replace the current use of multiple `StringType` widgets with `"Yes"` and `"No"` selections. The following is an example using three `StringType` widgets on one wizard page:



You can replace this with a single `ArrayType` widget:



The JSON for this variable is:

```
"SelectedFeatures": {
    "label": "Please select the features you would like to enable?",
    "description": "The selected features for this app",
    "defaultValue": [
     "Apex Execution"
    ],
    "variableType": {
      "type": "ArrayType",
      "itemsType" : {
        "type" : "StringType",
        "enums" : [
         "Apex Execution",
         "API",
         "Content Transfer",
         "Dashboard",
         "Login",
         "Report Export",
         "Rest API",
         "Setup Audit Trail",
         "UI Tracking",
         "URI",
         "Visualforce Request",
```

```
          "Wave Change",
          "Wave Interaction",
          "Wave Performance"
        ]
      }
    },
    "required": true
  }
```

You can use the selected values to set constants in `rules.json`, which you can then reference in conditionals for actions or in `template-info.json`.

```
"constants" : [
 {"name":"hasApexExecution", "value": "${array:contains(Variables.SelectedFeatures,
'Apex Execution')}"},
 {"name":"hasAPI", "value": "${array:contains(Variables.SelectedFeatures, 'API')}"},
 {"name":"hasContentTransfer", "value": "${array:contains(Variables.SelectedFeatures,
'Content Transfer')}"},
 {"name":"hasDashboard", "value": "${array:contains(Variables.SelectedFeatures,
'Dashboard')}"},
 {"name":"hasLogin", "value": "${array:contains(Variables.SelectedFeatures, 'Login')}"},

 {"name":"hasReport", "value": "${array:contains(Variables.SelectedFeatures, 'Report
Export')}"},
 {"name":"hasRestApi", "value": "${array:contains(Variables.SelectedFeatures, 'Rest
API')}"}
]
```

`json` Functions

Use json functions in rules to manipulate JSON at runtime. For example, find all the JSON paths in a dashboard that have a link widget and update the links at runtime.

Function Name: `searchPaths`

Parameters: String `jsonPath`

Returns: `Collection<String>` See *Description* for detail.

Recursive: Not supported

Description: Searches the current JSON document and returns matching fully qualified JSON paths. If no results are found, returns an empty array. If more than one item is found, returns an array of JSON paths.

Example:

```
${json:searchPaths('$.state.widget..[?(@.type in [\"link\"])]')}
```

run on a dashboard returns all JSON paths in the dashboard with link widgets:

```
["$['state']['widgets']['link3']", "$['state']['widgets']['link4']",
"$['state']['widgets']['link1']", "$['state']['widgets']['link2']" ].
```

These paths can be used to make runtime updates to the links in the dashboard.

Function Name: `searchValues`

Parameters: String `jsonPath`

Returns: `Collection<Object>` See *Description* for detail.

Recursive: Not supported

Description: Searches the current JSON document and returns values of the matching nodes. If no results are found, returns an empty array. If more than one item is found, returns an array of objects.

Example:

```
${json:searchValues('$.path.to.json.object')}
```

If this function is called outside of rules execution, an error is thrown.

Static Functions

Static functions have specialized use in template rules.

Function Name: `math:pow`

Parameters: Double `a`, Double `b`

Returns: Double

Recursive: Not supported

Description: Returns the value of the first argument raised to the power of the second argument.

Function Name: `math:min`

Parameters: Double `a`, Double `b`

>Returns: Double

Recursive: Not supported

Description: Returns the smaller of the two values.

Function Name: `math:max>`

Parameters: Double `a`, Double `b`

Returns: Double

Recursive: Not supported

Description: Returns the greater of the two values.

Function Name: `math:round`

Parameters: Double `x`, Int `scale`

Returns: Double

Recursive: Not supported

Description: Rounds the given value to the specified number of decimal places.

Function Name: `math:round`

Parameters: Double `x`, Int `scale`

Returns: Double

Recursive: Not supported

Description: Rounds the given value to the specified number of decimal places.

Function Name: `sfdc_fma:checkInteger`

Parameters: String `apiName`

Returns: Int

Recursive: Not supported

Description: Returns the integer value of the LMO to subscriber FMA value for `checkPackageIntegerValue`. See FeatureManagement Class.

Function Name: `sfdc_fma:checkBoolean`

Parameters: String `apiName`

Returns: Boolean

Recursive: Not supported

Description: Returns the boolean value of the LMO to subscriber FMA value for `checkPackageBooleanValue`. See FeatureManagement Class.

Function Name: `sfdc_fma:checkDate`

Parameters: String `apiName`

Returns: Date

Recursive: Not supported

Description: Returns the date value of the LMO to subscriber FMA value for `checkPackageDateValue`. See FeatureManagement Class.

Macros in Rules

Use a macro when a single rule executes repeatedly on different JSON node paths.

Rule macros enable template developers to define repeatable, well-tested rule code units that can be called within the context of any JSON transformation, simplifying and making the rule definition source easier to maintain.

You invoke macros using standard function syntax. Reference them anywhere a traditional expression is resolved within a rule action. In template rules, macros can be referenced by an 'eval' action. Macros can include actions and can reference other macros, and macro recursion is supported. Template support a macro depth of 10; beyond that results in an overflow error and app creation failure.

For example, consider the following macro for multiplying two numbers:

```
{
            "namespace": "testMacroNamespace",
            "definitions": [
                {
```

```
                    "name": "multiplyTwoNumbers",
                    "parameters": [
                        "firstNumber",
                        "secondNumber"
                    ],
                    "returns": "${p.firstNumber * p.secondNumber}"
                },
                {
                    "name": "shouldExecuteAction",
                    "returns": true
                }
            ]
}
```

It can be invoked in rules actions as follows:

```
{
    "rules": [
        {
            "name": "Testing of example macro",
            "actions": [
                {
                    "action": "set",
                    "key": "macroResult",
                    "path": "$.path.to.a.number",
                    "value": "${testMacroNamespace:multiplyTwoNumbers(3,3)}"
                },
                {
                    "action": "put",
                    "condition": "${testMacroNamespace:shouldExecuteAction()}",
                    "path": "$.my.json.path",
                    "value": {
                        "foo": "bar",
                        "anotherNumber": "${testMacroNamespace:multiplyTwoNumbers(5,5)}"
                    }
                }
            ]
        }
    ]
}
```

Rule Macro And Macro Definition Attributes

A rule macro object contains a collection of macro definition objects. Macro definition objects define the logic of macros and their return values.

Macro Examples

The following examples show the structure and syntax of macros in rules.

Macro Use Case: Recursive Operations

Here's an example of a macro that deletes an `sfdcDigest` node. It also deletes the digest node and all other nodes that depend on that node.

Macro Use Case: Delete Workflow Nodes

Use this macro to delete multiple nodes from the workflow without having to write multiple actions.

Macro Use Case: Add an Array of SObject Names to Extract Workflow

Here's a macro that adds an array of `sObject` names to the extract workflow.

Rule Macro And Macro Definition Attributes

A rule macro object contains a collection of macro definition objects. Macro definition objects define the logic of macros and their return values.

Macro object attributes are as follows:

- namespace: A unique string identifier (to the template) used to scope the collection of macro definitions.
- definitions: Array of macro definition objects defined within the rule macro..

Macro definition object attributes are as follows:

- name: The name of the macro.
- description: A string that describes what the macro does.
- parameters: An array of zero or more parameter names that the macro expects.
- actions: An array of zero or more rule actions to perform as part of the macro.
- returns: An expression or literal value to be returned from the macro.

Macro Definition Parameters

The number of parameters passed in the macro function call must match the number of parameters defined in the macro. They're passed in the order defined in the macro definition. You can use a maximum of 10 parameters per macro.

Parameters can be referenced in the body of the macro using standard expression syntax: `${p.<parameter_name>}`. Parameters can be referenced in any action that accepts expressions. Macros in actions support `"path"` statements so that expressions can be used for more dynamic JSON path handling. See Macro Examples.

Parameters are scoped to the execution of the macro call and any references to `${p.<parameter_name>)` fails outside the scope of the macro.

Macro Examples

The following examples show the structure and syntax of macros in rules.

Here's an example macro showing the use of attributes, parameters, and `"path"` statements.

```
"macros":
 [

        {

            "namespace": "myMacroNamespace",

           "definitions": [

                {

                    "name": "multiplyTwoNumbers",

                   "parameters": [
```

```
                        "firstNumber",
                       "secondNumber"
                    ],
                   "returns": "${p.firstNumber * p.secondNumber}"
               },
               {
                    "name": "deleteWidget",
                   "description": "Deletes a widget and any references to the widget.",
                   "parameters": [
                        "widgetName"
                    ],
                   "actions": [
                        {
                            "action": "eval",
                           "key": "results",
                        "path": "${json:searchPaths(\"$.state.widget['p.widgetName']\")}"
                       },
                        {
                            "condition": "${!empty results}",
                           "action": "delete",
                           "path": "$.state.widgets['${p.widgetName}']"
                        },
                        {
                            "condition": "${!empty results}",
                           "action": "delete",
                           "path":
"$.state.gridLayouts..pages..widgets[?(@.name=='${p.widgetName}')]"
```

```
                        }

                    ]

                }

            ]

        }

    ],

    "constants": [...],

    "rules": [...]
}
```

The following example calls a macro with namespace `'myNS'` and macro name `'getSomething'` defined with no parameters. Results of the macro are stored in the template context, and can be referenced using this expression: `${Rules.Eval.macroResults}`.

```
{
  "action": "eval",
  "key": "macroResults",
  "value": "${myNS:getSomething()}"
}
```

This example sets the JSON node at path $.path.to.json.node with the result of macro `myNS:getJsonValue` defined with no parameters. This action will only be called if the macro `myNS:shouldExecute` (defined with one parameter) returns true.

```
{
  "action": "set",
  "condition": "${myNS:shouldExecute(true)}",
  "path": "$.path.to.json.node",
  "value": "${myNS:getJsonValue()}"
}
```

Macro Use Case: Recursive Operations

Here's an example of a macro that deletes an `sfdcDigest` node. It also deletes the digest node and all other nodes that depend on that node.

This macro deletes the specified node and dynamically queries the JSON document to delete each dependency of those nodes.

Consider the simplified JSON below. To delete 'node3' and its dependencies, the result should delete: node3, node4, node5 (depends on node4), node6 (depends on node5), node7, node8, node9.

Example: Example JSON

```
{
    "node1": { },
    "node2": {
        "dependsOn": "node1"
    },
    "node3": {
```

```
        "dependsOn": "node2"
    },
    "node4": {
        "dependsOn": "node3"
    },
    "node5": {
        "dependsOn": "node4"
    },
    "node6": {
        "dependsOn": "node5"
    },
    "node7": {
        "dependsOn": "node3"
    },
    "node8": {
        "dependsOn": "node3"
    },
    "node9": {
        "dependsOn": "node3"
    }
}
```

Example: Without Macro

Without using a macro (and associated enhancements), nodes would have to be deleted individually.

```
{
    "actions": [
        {
            "action": "delete",
            "path": "$.node3"
        },
        {
            "action": "delete",
            "path": "$.node4"
        },
       {
            "action": "delete",
            "path": "$.node5"
        },
       {
            "action": "delete",
            "path": "$.node6"
        },
        {
            "action": "delete",
            "path": "$.node7"
        },
        {
            "action": "delete",
            "path": "$.node8"
        },
        {
            "action": "delete",
```

```
            "path": "$.node9"
        }
    ]
}
```

This code is rigid and difficult to maintain because rules would need to be modified as node dependencies are added and removed from the JSON payload.

Example: Macro Source

```
{

    "namespace": "macroRecursion",

   "definitions": [

        {

            "name": "deleteNodeAndDependencies",

           "description": "Delete a node and its dependencies.  Returns an array of
json paths of all the nodes that were deleted.",

           "parameters": [

                "nodeName"

            ],

           "actions": [

                {"action": "eval", "key": "fullNodePath",  "value": "$.${p.nodeName}"
 },

               {"action": "eval", "key": "dependencies",  "value":
"${macroRecursion:getDependents(p.nodeName)}"},

               {

                  "action": "eval",

                 "value": "${array:forEach(Rules.Eval.dependencies,
'${macroRecursion:deleteNodeAndDependencies(macroRecursion:deleteSingleNodeByFullJsonPath(var))}')}"


                },

                {

                    "action": "eval",

                   "value":
"${macroRecursion:deleteSingleNodeByFullJsonPath(Rules.Eval.fullNodePath)}"
```

```
                }

            ]

        },

        {

            "name": "deleteSingleNodeByFullJsonPath",

           "description": "Deletes a node by full json path.",

           "parameters": [

                "fullJsonPath"

            ],

           "actions": [

               {"action": "eval", "key": "pathSegments",
  "value": "${string:match(p.fullJsonPath,'\\\\[\\'(.*?)\\'\\\\]')}"},

             { "action": "delete",  "path": "${p.fullJsonPath}"}

            ],

           "returns": "${array:last(Rules.Eval.pathSegments)}"

        },

        {

            "name": "getDependents",

           "description": "Returns the full json path to search results",

           "parameters": [

                "nodeName"

            ],

           "actions": [

               { "action": "eval", "key": "searchString", "value":
"$.*[?(@.dependsOn=='${p.nodeName}')]"},

             { "action": "eval", "key": "paths", "value":
"${json:searchPaths(Rules.Eval.searchString)}"}

            ],

           "returns": "${Rules.Eval.paths}"
```

```
        }

    ]
}
```

Example: Rule Calling the Macro

```
{
    "rules": [
        {
            "name": "Recursively delete node3 and dependencies",
            "actions": [
                {"action": "eval", "value":
"${macroRecursion:deleteNodeAndDependencies('node3')}" }
            ]
        }
    ]
}
```

Macro Use Case: Delete Workflow Nodes

Use this macro to delete multiple nodes from the workflow without having to write multiple actions.

Example: Macro

```
        {
            "name": "deleteWorkflowNode",
            "description": "Deletes a workflow node.",
            "parameters": [
                "nodeName"
            ],
            "actions": [
                { "action": "delete",  "path": "$.workflowDefinition.${p.nodeName}"}
            ]
        },
        {
            "name": "deleteArrayOfWorkflowNodes",
            "description": "Deletes a set of workflow nodes.",
            "parameters": [
                "nodeNameArray"
            ],
            "actions": [
                { "action": "eval",
                "value": "${array:forEach(p.nodeNameArray,
'${macros:deleteWorkflowNode(var)}')}"}
            ]
        }
```

Example: Rule Calling the Macro

```
{"action":"eval","key" :
"myArray","value":["Extract_Queue","Add_Fields_To_Queue","Append_Queue_User"]},
```

```
      {
           "action": "eval",
           "description": "remove multiple nodes in dataflow",
           "value": "${macros:deleteArrayOfWorkflowNodes(Rules.Eval.myArray)}"
         }
```

Macro Use Case: Add an Array of SObject Names to Extract Workflow

Here's a macro that adds an array of `sObject` names to the extract workflow.

Example: Macro

```
{
       "name" : "concatArrayFieldName",
       "description" : "Concatenates field names from an sobject array node",
       "parameters" :
       [
        "variable"
       ],
       "returns": "${array:concat(Rules.CurrentNode,
array:forEach(p.variable,'{\"name\":\"${var.fieldName}\"}'))}"

      },
      {
       "name" : "addToExtractCaseWorkflow",
       "description" : "Adds array fields to Extract Case node",
       "parameters" :
       [
        "variable"
       ],
       "actions": [
        {
         "action": "set",
             "description": "put selected values for sfdcDigest in dataflow",
             "path": "$.workflowDefinition.Extract_Case.parameters.fields",
             "value" : "${macros:concatArrayFieldName(p.variable)}"
            }
            ]
      }
```

Example: Rule Calling the Macro

```
{
           "action": "eval",
           "description": "put selected values for sfdcDigest in dataflow",
           "value": "${macros:addToExtractCaseWorkflow(Variables.CaseMoreDims)}"
         }
```

Use the Apex `WaveTemplateConfigurationModifier` Class

The `WaveTemplateConfigurationModifier` class checks an org's data. Also, to simplify app creation, it can modify the template configuration wizard accordingly.

Apex is the object-oriented Salesforce programming language that allows developers to execute flow and transaction control statements on the Lightning Platform server along with calls to the API.

Using `WaveTemplateConfigurationModifier` in a smart wizard simplifies standard app creation in these ways:

- Checks if the org has data required to create the app.
- Checks if security settings enable app creation. For example, it determines if the Analytics Integration User has access to all fields required to create the app.
- Checks the org's fiscal year start date. Based on the outcome, the wizard can use a default setting, such as January 1, or require the user to choose a fiscal year start date for use in the app.
- Hides or exposes a wizard question (variable) based on the org data. For example, say that your wizard includes a question about opportunities. The smart wizard can check an org for the Opportunity object and, if it detects use of the object., display the question.
- Sets the values included in a wizard question's answer field. The smart wizard can include or exclude specific fields and expand or limit the number of values displayed.

The `WaveTemplateConfigurationModifier` can also be used with embedded apps to check the org's data and security settings. In this use case, you can fail app creation with a clear message for the Auto-Install request. You can also set template variables based on org settings to be used in app creation.

Declare the class in `template-info.json`.

If you implement the modifier and unit test classes in your Visual Studio project, include your classes in the `classes` directory along with the class metadata files.

For complete details about Apex, see Apex Developer Guide.

WaveTemplateConfigurationModifier Interfaces
The main class is `WaveTemplateConfigurationModifier` with these supporting classes and their methods.

WaveTemplateConfigurationModifier Methods
Extend `WaveTemplateConfigurationModifier` with the following methods.

WaveTemplateConfigurationModifier Apex Examples
Examples include setting computed values for questions, hiding a question, managing enum values, working with array type variables, and more.

TemplateApexException Apex Examples
Use the `TemplateApexException` class to handle errors cleanly and provide clear information to template users.

TemplateInterruptException Apex Example
Use the `TemplateInterruptException` class to handle template runtime errors cleanly and provide clear information to template users.

Test WaveTemplateConfigurationModifier
Test the `WaveTemplateConfigurationModifier` before deploying a template that uses it.

`WaveTemplateConfigurationModifier` Interfaces

The main class is `WaveTemplateConfigurationModifier` with these supporting classes and their methods.

WaveTemplateInfo

Provides access to a template's release information, UI information, and variable definitions

Method	Description
`ReleaseInfo getReleaseInfo();`	Gets the release information for the template.
`UI getUI();`	Gets the UI information for the template configuration wizard.
`Map<String, VariableDefinition> getVariables();`	Gets the variable definitions for the template.

ReleaseInfo

The release information for a template.

Method	Description
`String getTemplateVersion();`	Gets the version of the template.

UI

The UI information used to dynamically build the configuration wizard for the template.

Method	Description
`List<UIPage> getPages();`	Gets a list of all the pages for the configuration wizard.

UIPage

The information for a page in the configuration wizard.

Method	Description
`String getTitle();`	Gets the title of the configuration wizard page.
`String getCondition();`	Gets the condition expression of the configuration wizard page.
`Map<String, Variable> getVariables();`	Gets the variables for the configuration wizard page.
`void setTitle(String title);`	Set the title for the configuration wizard page. This value overrides the value specified in the JSON file.

Answers

Contains a collection of all the answers to the variables.

Method	Description
`Object get(String name);`	Get the value (answer) of a specific variable (question).
`void put(String name, Object obj);`	Update the value (answer) for a specific variable (question).

Variable

Represents a template variable on a template configuration page.

Method	Description
`String getName();`	Get the name of the variable.
`String getVisibility();`	Get the visibility of the variable in the configuration wizard. Returns a string as the value can contain a ternary operator.
`void setVisibility(VisibilityEnum visibility);`	Set the visibility of the variable in the configuration wizard. This value overrides the value specified in the JSON file. Valid values are: • `Disabled` • `Hidden` • `Visible`

VariableDefinition

Represents a template variable definition.

Method	Description
`String getName();`	Get the name for the variable definition.
`String getLabel();`	Get the label for the variable definition.
`String getDescription();`	Get the description for the variable definition.
`Object getDefaultValue();`	Get the default value for the variable definition.
`VariableType getVariableType();`	Get the variable type information for the variable definition.
`Boolean isRequired();`	Indicates whether a variable value is required for this variable definition (`true`) or not (`false`).
`Boolean isExcludeSelected();`	Indicates whether values selected in other variables should be excluded from this variable definition (`true`) or not (`false`).
`Set<String> getExcludes();`	Return a list of variable values to be excluded from the pick list displayed to the user. This can be a single regex.
`void setLabel(String label);`	Set the label for the variable definition. This value overrides the label specified in the JSON file.

Method	Description
`void setDescription(String description);`	Set the description for the variable definition. This value overrides the description specified in the JSON file.
`void setComputedValue(String Object);`	Set the computed value for the variable definition. This value overrides the value specified in the JSON file and replaces the `defaultValue`.
`void setRequired(Boolean required);`	Set the `isRequired` value for the variable definition. This value overrides the value specified in the JSON file.
`void setExcludes(Set<String> excludes);`	Set the excludes list value for the variable definition. This value overrides the value specified in the JSON file.

VariableType

The variable type of a variable definition.

Method	Description
`VariableTypeEnum getType();`	Gets the variable type for the variable definition. Valid values are: • `BooleanType` • `StringType` • `NumberType` • `ArrayType` • `DateTimeType` • `ObjectType` • `SobjectType` • `SobjectFieldType` • `DatasetType` • `DatasetDimensionType` • `DatasetMeasureType` • `DatasetDateType`
`String getDataType();`	Gets the SOAP data type for a `SObjectFieldType` variable definition. Valid values are: • `xsd:int` • `xsd:string` • `xsd:datetime`, • `xsd:boolean` • `xsd:any`
`VariableTypeEnum getType();`	Gets the variable type for the variable definition. Valid values are: • `BooleanType`

Method	Description
	• `StringType` • `NumberType` • `ArrayType` • `DateTimeType` • `ObjectType` • `SobjectType` • `SobjectFieldType` • `DatasetType` • `DatasetDimensionType` • `DatasetMeasureType` • `DatasetDateType`
`VariableType getItemsType();`	Gets the variable type for each entry in an `ArrayType` variable definition.
`Set<Object> getEnums();`	Gets the enumerated values for a `StringType` or `NumberType` variable definition.
`void setEnums(Set<Object> enums);`	Set the enumerated values for a `StringType` or `NumberType` variable definition. These values overrides the values specified in the JSON file.

`WaveTemplateConfigurationModifier` Methods

Extend `WaveTemplateConfigurationModifier` with the following methods.

- `onConfigurationRetrieval`. Runs before the wizard starts. This method is for standard apps only. If the org lacks required data or settings, you can customize the wizard (variables) based on the org data check or disable app creation. Requires `WaveTemplateInfo`:

```
onConfigurationRetrieval(wavetemplate.WaveTemplateInfo waveTemplate)
```

- `beforeAppCreate`. Runs after completion of the wizard for standard apps or on app creation for embedded apps. Use this method to avoid errors during app creation as the result of invalid user selections or org settings. Requires `WaveTemplateInfo` and `Answers`:

```
beforeAppCreate(wavetemplate.WaveTemplateInfo template, wavetemplate.Answers answers)
```

- `beforeAppUpdate`. Runs before app updates based on new version of the template for standard and embedded apps. Use this method to validate that org data is compatible with new version. Requires `WaveTemplateInfo`, `Answers`, and the previous app version:

```
beforeAppUpdate(wavetemplate.WaveTemplateInfo template, String previousAppVersion,
wavetemplate.Answers answers)
```

The methods are optional and all of them can be overridden.

WaveTemplateConfigurationModifier Apex Examples

Examples include setting computed values for questions, hiding a question, managing enum values, working with array type variables, and more.

Example: **Create your class:**

```
public class ExampleWaveTemplateConfigurationModifier
    extends wavetemplate.WaveTemplateConfigurationModifier
{
   public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo template)

    {
        // Method implementation
    }
}
```

Example: **Set the computed value for one of the questions:**

```
public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo template)
{
    Map<string, wavetemplate.VariableDefinition> variables = template.getVariables();

    variables.get('Has_Product').setComputedValue('No');
}
```

Example: **Hide one of the questions:**

```
public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo template)
{
    template.getUI().getPages().get(0).getVariables()
        .get('Has_Product').setVisibility(wavetemplate.VisibilityEnum.Hidden);
}
```

Example: **Remove an enum value from the dropdown list box using the specified array index:**

```
public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo template)
{
    Map<string, wavetemplate.VariableDefinition> variables = template.getVariables();

    wavetemplate.VariableDefinition hasProduct = variables.get('Has_Product');
    List<object> enums = hasProduct.getVariableType().getEnums();
    enums.remove(1);
    hasProduct.getVariableType().setEnums(enums);
}
```

Example: **Populate the enum values in the dropdown list box (StringType variable):**

```
public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo template)
{
    Map<string, wavetemplate.VariableDefinition> variables = template.getVariables();

    wavetemplate.VariableDefinition caseRecordTypes =
        variables.get('IncludeCaseRecordTypes');
```

```
    List<object> enums = caseRecordTypes.getVariableType().getEnums();
    RecordType [] results = [SELECT Name FROM RecordType where sobjectType = 'Case'];

    for (RecordType record : results){
        //adds the record type names to the drop down list
        enums.add((string)record.get('Name'));
    }
    caseRecordTypes.getVariableType().setEnums(enums);
}
```

Example: **Given the following ArrayType variable:**

```
"OpptyRecordType":
{
    "label": "Select OpptyRecordTypes that you want to bring in to Salesforce",
    "description": "Choose OpptyRecordTypes that you want to bring in to Salesforce.",

    "required": false,
    "variableType": {
        "type": "ArrayType",
        "itemsType": {
            "type": "StringType",
            "enums": ["Test", "Test2"]
        }
    }
}
```

Add more values to the ArrayType variable enums:

```
    Map<string, wavetemplate.VariableDefinition> variables = template.getVariables();

    wavetemplate.VariableDefinition enumsTest = variables.get('OpptyRecordType');
    List<object> enums = enumsTest.getVariableType().getItemsType().getEnums();
    //create an empty list if you want to overwrite the original enums.
    RecordType [] results = [SELECT Name FROM RecordType where sobjectType
        = 'Opportunity' AND IsActive = true];
    for ( RecordType record : results ) {
        enums.add((string)record.get('Name'));
    }
    enumsTest.getVariableType().getItemsType().setEnums(enums);
```

Example: **Reduce the possible answers via the excludes:**

```
public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo template)
{
    Map<string, wavetemplate.VariableDefinition> variables = template.getVariables();

    wavetemplate.VariableDefinition hasProduct = variables.get('Has_Product');
    Set<string> excludes = new Set<string>();
    excludes.add('Foo');
    excludes.add('Bar');
    hasProduct.setExcludes(excludes);
}
```

Example: **Set a variable answer before the app creates. Updating answers overwrites any value the user selected in the wizard:**

```
public override void beforeAppCreate(wavetemplate.WaveTemplateInfo waveTemplate,
    wavetemplate.Answers answers)
{
    answers.put('Has_Product', 'No');
}
```

Example: **Set a variable answer before the app updates. Updating answers overwrites any value the user selected in the wizard:**

```
public override void beforeAppUpdate(wavetemplate.WaveTemplateInfo waveTemplate,
    String previousAppVersion, wavetemplate.Answers answers)
{
    answers.put('Has_Product', 'Yes');
}
```

TemplateApexException Apex Examples

Use the `TemplateApexException` class to handle errors cleanly and provide clear information to template users.

Note: Throwing an exception fails the configuration wizard. You can also create variables to track errors and report them to the user in the wizard without failing.

Example: **Report an error to the user with a TemplateApexException:**

```
public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo template)
{
    // Run a query if Forecast History is enabled and report failure if isn't
    List<User> usersWithForecast = Database.query('SELECT Id FROM User WHERE
ForecastEnabled = true LIMIT 1');
    if (usersWithForecast.isEmpty()) {
        // Report issue to user
        throw new TemplateApexException('Forecast history is not enabled. Please have
 your admin enable it.');
    }
}
```

Example: **Handle database exceptions with a TemplateApexException**

```
public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo template)
{
     String query = 'SELECT DeveloperName, MasterLabel, SplitField, SplitEntity FROM
OpportunitySplitType WHERE IsActive = true AND SplitEntity = \'Opportunity\'';
     try {
         List<SObject> activeSplitTypes = Database.query(query);
         if (activeSplitTypes.isEmpty()) {
           template.getVariables().get('Has_OpportunitySplits').setComputedValue('No');

         }
         else {
```

```
template.getVariables().get('Has_OpportunitySplits').setComputedValue('Yes');
          }
       }
       catch (Exception e) {
           throw new TemplateApexException('Database query to determine Opportunity
Splits failed: ' + e.getMessage());
       }
}
```

TemplateInterruptException Apex Example

Use the `TemplateInterruptException` class to handle template runtime errors cleanly and provide clear information to template users.

Note: Throwing an exception fails the app creation process.

Example: **Report an error to the user with a TemplateInterruptException:**

```
public override void beforeAppCreate(wavetemplate.WaveTemplateInfo template.
wavetemplate.Answers answers)
{
 // Validate that an answer is present and has a value.
 String includeCasesAnswer = (String)answers.get('IncludeCases');
 if (includeCasesAnswer.isEmpty() || includeCasesAnswer == null) {
  // Report issue to user
  throw new TemplateInterruptException('You didn't answer "yes" to the question that
asked if you use Cases. Go back to the wizard and answer this question.');
 }
}
```

Test `WaveTemplateConfigurationModifier`

Test the `WaveTemplateConfigurationModifier` before deploying a template that uses it.

Before you can deploy your template or package it for the Salesforce AppExchange that uses a `WaveTemplateConfigurationModifier` class, a unit test class must be present and all of the tests must complete successfully. The unit test is designed to call and run the modifier methods and nothing more. Use the following examples when creating your own unit test class.

Use the `wavetemplate.Test` helper methods for testing.

- `wavetemplate.Test.getWaveTemplateInfoForApexTesting`. Retrieves the runtime template information object for testing. Requires `templateName`:

  ```
  getWaveTemplateInfoForApexTesting(String templateName)
  ```

- `wavetemplate.Test.getDefaultAnswersForApexTesting`. Retrieves the variable answers set by user selections and/or the configuration modifier methods. Requires `templateName`:

  ```
  getDefaultAnswersForApexTesting(String templateName)
  ```

The following code examples show how to test the methods from the Apex Examples page.

Example: Generic Test Class Example

```
@isTest
private class ExampleWaveTemplateConfigurationModifierTest
{
    @isTest
    static testMethod void testModifier()
    {
        // Test modifier methods
    }
}
```

Example: `beforeAppCreate` Test Method Example

```
 @isTest static testMethod void testBeforeAppCreate()
{
    wavetemplate.WaveTemplateInfo template =
            wavetemplate.Test.getWaveTemplateInfoForApexTesting('my_template');
    wavetemplate.Answers answers =
            wavetemplate.Test.getDefaultAnswersForApexTesting('my_template');

    ExampleWaveTemplateConfigurationModifier mod =
        new ExampleWaveTemplateConfigurationModifier();

    //Test the beforeAppCreate method in your Apex Modifier Class
    System.assertEquals('Yes', answers.get(Has_Product));
    mod.beforeAppCreate(template, answers);
    System.assertEquals('No', answers.get(Has_Product));
}
```

Example: `beforeAppUpdate` Test Method Example

```
 @isTest static testMethod void testBeforeAppUpdate()
{
    wavetemplate.WaveTemplateInfo template =
            wavetemplate.Test.getWaveTemplateInfoForApexTesting('my_template');
    wavetemplate.Answers answers =
            wavetemplate.Test.getDefaultAnswersForApexTesting('my_template');

    ExampleWaveTemplateConfigurationModifier mod =
        new ExampleWaveTemplateConfigurationModifier();

    //Test the beforeAppUpdate method in your Apex Modifier Class
    System.assertEquals('No', answers.get(Has_Product));
    mod.beforeAppUpdate(template, '2.0', answers);
    System.assertEquals('Yes', answers.get(Has_Product));
}
```

Example: `onConfigurationRetrieval` Test Method Example

```
 @isTest static testMethod void testOnConfig()
{
    wavetemplate.WaveTemplateInfo template =
            wavetemplate.Test.getWaveTemplateInfoForApexTesting('my_template');
```

```
    ExampleWaveTemplateConfigurationModifier mod =
        new ExampleWaveTemplateConfigurationModifier();

    //Test the onConfigurationRetrieval method in your Apex Modifier Class
    mod.onConfigurationRetrieval(template);
    System.assertEquals('No',
        template.getVariables().get('Has_Product').getComputedValue());
}
```

For complete information, see Testing Apex.

Use the Apex `Access` Methods

The `Access` class provides utility methods to check Integration User access to sObjects and sObjectFields. Use these methods in your `WaveTemplateConfigurationModifier` implementation.

Apex is the object-oriented Salesforce programming language that allows developers to execute flow and transaction control statements on the Lightning Platform server along with calls to the API.

Using `Access` in a smart wizard simplifies standard app creation in these ways:

- Checks if the Integration User has access to sObjects.
- Checks if the Integration User has access to sObjectFields.
- Checks if the Integration User has access to enough records in an sObject to continue with template processing.

The `Access` methods can also be used with embedded apps to check if the org has given access to the Integration User for the necessary sObjects and sObjectFields. In this use case, you can fail app creation with a clear message for the Auto-Install request.

For complete details about Apex, see Apex Developer Guide.

Access Methods
Use the `Access` features with the following methods.

Access Apex Examples
Examples include checking the Integration User access inside your WaveTemplateConfigurationModifier implementation.

`Access` Methods

Use the `Access` features with the following methods.

- `checkIntegUserAccessToArrayOfSObjectFields`. Checks an array of sObjectFields from a template answer and returns a list of fields the Integration User doesn't have access to. Requires the variable name, `WaveTemplateInfo`, and `Answers`:

```
global static List<String> checkIntegUserAccessToArrayOfSObjectFields(String variableName,
 wavetemplate.WaveTemplateInfo template, wavetemplate.Answers answers)
```

- `getCountOfRecordsAsIntegUser`. Returns the number of records visible to the Integration User. Use this method to gauge if the number of records is high enough to continue with template processing. There's a limit of 10K rows. Requires the sObject name to use for the count:

```
global static Integer getCountOfRecordsAsIntegUser(String sObjectName)
```

- `integUserHasAccessToSObjectField`. Checks an sObjectField answer to verify that the Integration User has access to it. Requires the variable name, `WaveTemplateInfo`, and `Answers`:

```
global static boolean integUserHasAccesstoSObjectField(String variableName,
wavetemplate.WaveTemplateInfo template, wavetemplate.Answers answers)
```

- `integUserHasAccessToSObjectField`. Checks a single sObject and one of its fields to verify that the Integration User has access to it. Requires the sObject name and `WaveTemplateInfo`:

```
global static boolean integUserHasAccessToSObjectField(String sObjectName,
wavetemplate.WaveTemplateInfo template)
```

Access Apex Examples

Examples include checking the Integration User access inside your WaveTemplateConfigurationModifier implementation.

Example: **In the `onConfigurationRetrival` implementation, verify the Integration User access to the Account sObject. Add the access result to the template variables.**

```
public class ExampleWaveTemplateConfigurationModifier
    extends wavetemplate.WaveTemplateConfigurationModifier
{
    public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo
waveTemplate)
    {
        String accessResult = null;
       if (!wavetemplate.Access.integUserHasAccessToSObjectField('Account', 'Industry'))
 {
           accessResult = 'No Access';
        } else {
           accessResult = 'Access found';
        }
        template.getVariables().get('accountAccess').setComputedValue(accsssResult);
    }
}
```

Example: **In the `onConfigurationRetrival` implementation, verify the Integration User has access more than 100 records in the Account sObject. Add the result to the template variables.**

```
public class ExampleWaveTemplateConfigurationModifier
    extends wavetemplate.WaveTemplateConfigurationModifier
{
    public override void onConfigurationRetrieval(wavetemplate.WaveTemplateInfo
waveTemplate)
    {
        Boolean enoughAcctRecords = true;
        if (!wavetemplate.Access.getCountOfRecordsAsIntegUser('Account') > 100) {
            enoughAcctRecords = false;
        }

template.getVariables().get('sufficientAcctRecordsFound').setComputedValue(enoughAcctRecords);
```

```
    }
}
```

Example: **Before creating the app, verify Integration User access to an sObject field selected during template wizard processing:**

```
public class ExampleWaveTemplateConfigurationModifier
    extends wavetemplate.WaveTemplateConfigurationModifier
{
    public override void beforeAppCreate(wavetemplate.WaveTemplateInfo waveTemplate,
        String previousAppVersion, wavetemplate.Answers answers)
    {
        Boolean access =
wavetemplate.Access.integUserHasAccessToSObjectField('sObjectFieldExample', template,
 answers);

        if (!access) {
           throw new wavetemplate.TemplateInterruptException('Integ User doesn't have
 access to sObjectFieldExample');
        }
    }
}
```

Example: **Before creating the app, verify if there are any sObject fields the Integration User doesn't have access from an array of fields selected during template wizard processing:**

```
public class ExampleWaveTemplateConfigurationModifier
    extends wavetemplate.WaveTemplateConfigurationModifier
{
    public override void beforeAppCreate(wavetemplate.WaveTemplateInfo waveTemplate,
        String previousAppVersion, wavetemplate.Answers answers)
    {
        List<String> noAccessFields =
wavetemplate.Access.checkIntegUserAccessToArrayOfSObjectFields('sobjectFieldArray',
template, answers);

        if (!noAccessFields.isEmpty()) {
           throw new wavetemplate.TemplateInterruptException('Integ User doesn't have
 access to ' + noAccessFields.get(0));
        }
    }
}
```

STEP 4: DEPLOY THE WAVETEMPLATE OBJECT

After editing its JSON files, deploy the template object back to the development org to see the results of your work and for testing.

Deploying your edited files pushes the changes you've made to the development org for testing and collaborative development by the rest of your team. Before you deploy your changes, they reside only in your local file system. Deploying your changes overwrites any files in your development org. Retrieving them again from the development org overwrites your local files.

- Deploy with CLI. Use the `force:source:push` command to push them back to your scratch org.
- Deploy with Salesforce Extensions for Visual Studio Code. Use the Command Palette **Push Source from Default Scratch Org**

STEP 5: TEST THE TEMPLATE

Test the template throughout the development process. After creating the template and exporting its files, you edit them, and then deploy them for testing. Based on the results, reexport the files and edit, deploy, and test again until you've achieved the result that you want.

1. Create the source app.
2. Create the template object with CRM Analytics Studio, CLI, or VS Code.

   Test 1—Create an app from the unmodified template to make sure that everything works.

   Note: CRM Analytics usually tokenizes the template correctly, but there are occasionally problems that require manual editing. Incorrect tokens in the template cause app creation to fail because CRM Analytics doesn't know what to substitute for the incorrect token. When app creation fails, you usually see a message like the following: `Expression error occurred [Error processing expression "${App.Dashboards.SupportingDashboard..Name}". Variable part [SupportingDashboard] not found in context map, check for typo in EL Expression.]`. The message indicates that the referenced piece of the context doesn't exist. To fix that, edit `template-info.json` to add the context piece with the correct name.

3. Create an app from your template.
   a. For app templates, create the app from the template in Analytics Studio. You can view the app creation log after app creation completes. With the app open, select **Details**. The **App Log** shows each asset that was created in the order it that was created, the asset status, and the overall app status. If app creation fails, the error messages are logged here.
   b. For data templates, create the data assets from the template in Data Manager.
   c. For embedded app and data templates, the installation process can be monitored on the Auto-Installed Apps page. For more details on how to find and use this page, see Monitor, Update, and Delete Auto-Installed Apps
4. To inspect and debug the app in detail, use the App Install History page in Setup. This page contains detailed logs and the runtime state for each install task. For more details, see the Use the App Install History Page.
5. Retrieve the template object using CLI or VS Code.
6. Edit the JSON files.
7. Deploy the updates with CLI or VS Code.

   Test 2— To evaluate the wizard and the app assets that it creates, create an app from the updated template.

8. Most likely you'll make many changes, mostly to the template assets, but sometimes also to the source app.
   a. Make changes in the source app to the dashboards, lenses, or dataset. Update the template object by using Analytics Studio to update the template, with CLI (`analytics:template:update`), or the VS Code **Update Analytics Template From App** command. These changes probably require more changes to the wizard or other template assets, so export the template files again and edit them. Update the template object again.

      Run **Test 2** again.

      OR

   b. Make changes to the wizard, rules, or variables. Update the template object again.

      Run **Test 2** again.

9. If you're using Smart Wizard functionality, test the Apex class after you have a fully functional template wizard. Test it to make sure that the Apex class is working as expected; giving variables programmatically assigned values, hiding or changing the visibility of variables and UI pages, giving feedback to the user, and so forth.

   Save your template in the source control system of your choice, such as GitHub.

Use the App Install History Page
Use the App Install History page to monitor installations in progress, track the assets created for the app, and view logs to troubleshoot any installation issues.

Debug the Template
CRM Analytics provides debug logs for templated apps to inspect the app creation lifecycle. These debug logs give you visibility into asset creation and the processing of any rules or Apex configuration modifier classes that you created. Use the debug logs during template testing when app creation isn't working as expected.

Reading the Template Debug Logs
Each `Analytics Templated App Poller` log contains multiple log lines generated from different log events.

Use the App Install History Page

Use the App Install History page to monitor installations in progress, track the assets created for the app, and view logs to troubleshoot any installation issues.

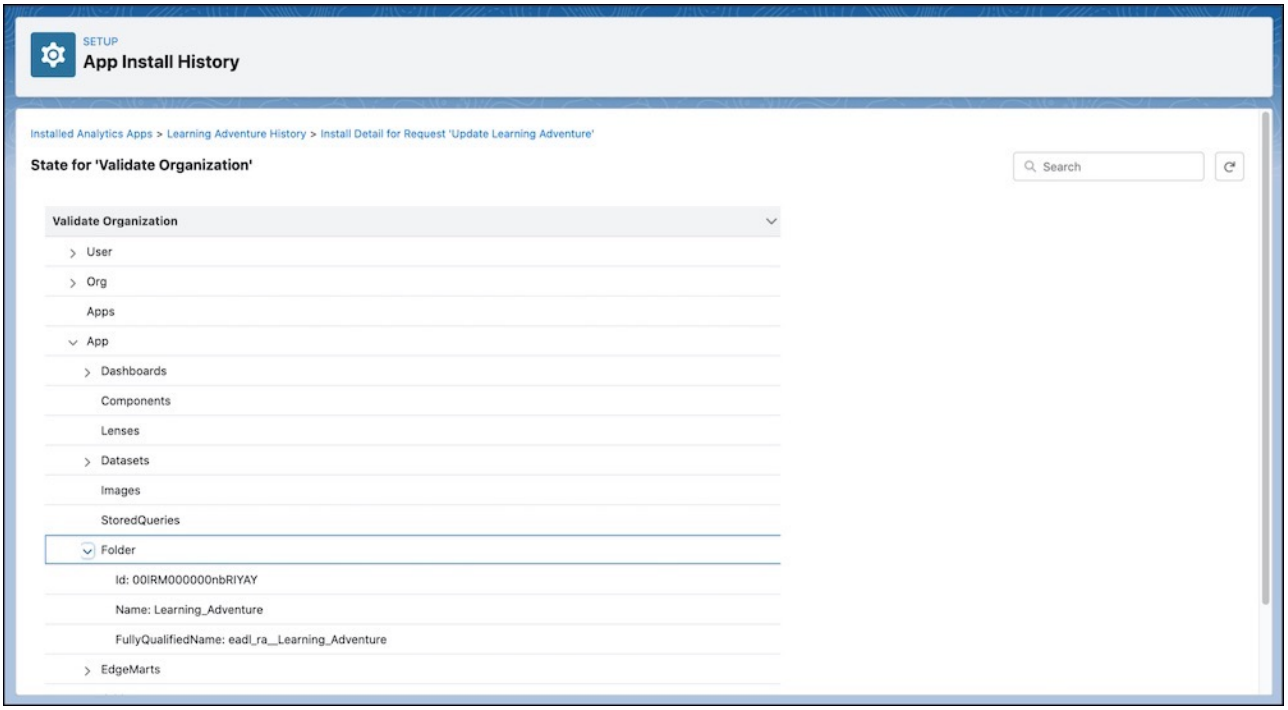
For an overview of the App Install History page, see Monitor and Troubleshoot Templated App Installations. The default view on this page is for App Templates. Use the filter to select Data Templates.

USER PERMISSIONS

To manage installed apps:
- Manage Analytics Templates

Templates use tokens to assign IDs, names, and labels at runtime, along with replacing variables and constants. The app state is the runtime state of the templated app during creation or upgrade. It stores the context information of the tokens used to create and update the app assets, along with data prep uploads and job runs.

The state for each install task detail record contains context information about the user, org, apps, app, variables, and constants, represented as a tree. For each item, click › to expand the elements of the tree.



User: The user context information for installing the templated app, which includes user ID, username, first name, and last name.

Org: The org context information for installing the templated app, which includes the org ID, the org name, and the org namespace.

Apps: The dependent app context information. This information includes the template name for dependent apps, the assets in those apps, and the asset source names.

App: The asset context information for the app. The list of assets updates with each installation step, as the runtime determines which assets are created or updated and then processes the assets. Possible assets for a templated app include:

- Dashboards: a list of dashboards with ID, name, and label
- Components: a list of components with ID, name, and label
- Lenses: a list of lenses with ID, name, and label
- Datasets: a list of datasets with ID, version ID, alias, fully qualified name, name, and label
- EdgeMarts: same as the list of datasets (Deprecated)
- Folder: the folder ID, name, and fully qualified name
- Images: a list of images with ID, name, and namespace
- Dataflows: a list of dataflows with ID
- Recipes: a list of recipes with ID, name, and label
- Connections: a list of connections with name

Variables: The runtime variables that are used to generate the templated app. The variables are a list of name and value pairs to display the value of each variable at runtime.

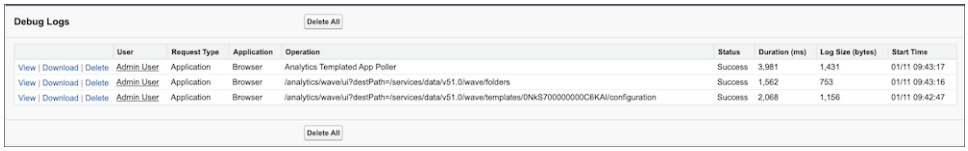
Constants: The runtime constants that are used to generate the templated app. The constants are a list of name and value pairs to display the value of each constant at runtime.

Debug the Template

CRM Analytics provides debug logs for templated apps to inspect the app creation lifecycle. These debug logs give you visibility into asset creation and the processing of any rules or Apex configuration modifier classes that you created. Use the debug logs during template testing when app creation isn't working as expected.

These steps cover setting up your trace flag to create your debug logs, creating your app to generate debug logs, and viewing the generated debug logs.

1. From Setup, in the **Quick Find** box, enter Debug Logs, then click **Debug Logs**.
2. Click **New**.
3. Select the entity to trace. For apps created in CRM Analytics Studio, the entity type must be `User` and the entity name must be your username. For apps created by the auto-install process, the entity type must be `Automated Process`.
4. The default time period to collect logs is set for 30 minutes. If a different or longer time period is required, update the **Start Date** or the **Expiration Date** values.
5. To create your debug level, click **New Debug Level**. Find the `Wave` debug log category and set it to `Finest`. Save your changes. You can reuse this debug level each time you create a trace flag.
6. To activate your trace flag, click **Save**.
7. From Analytics Studio, create an app from your template or kick off the automated process that creates the templated app. When app creation begins, debug logs are generated.
8. On the `Debug Logs` page, all debug logs are listed. To see your generated debug logs, refresh the page.



9. View the `Analytics Templated App Poller` log for details on the templated app creation process. Other logs listed from the templated app process include REST API calls to load the template wizard configuration and create the app folder and any Apex WaveTemplateConfigurationModifier output. If your template loads external data files, this process creates multiple `Analytics Templated App Poller` logs.

   Download any logs to save past the specified expiration date. Open your downloaded logs in a syslog viewer to allows filtering and easier reading.

Reading the Template Debug Logs

Each `Analytics Templated App Poller` log contains multiple log lines generated from different log events.

Multiple logs are written out during the templated app lifecycle of create, reset, or upgrade, usually 3–4 separate logs containing different details from the processing. Each log is a `Analytics Templated App Poller` log.

Note: Each log entry has a 2-MB limit controlled by org limits and is truncated when the limit is exceeded.

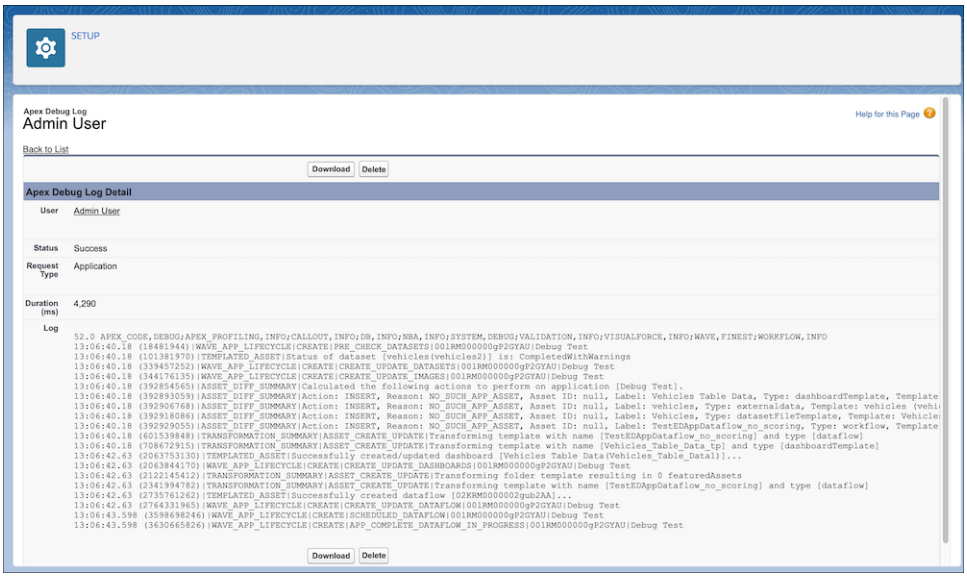
The `Analytics Templated App Poller` log includes these details.

Log Level	Apex Log Event	Details	Example Log Line
Info	WAVE_APP_LIFECYCLE	The app action, app lifecycle phase, app id, and app name.	14:04:00.399 (51399277642) |WAVE_APP_LIFECYCLE|CREATE |CREATE_UPDATE_DASHBOARDS |001RM000000gP2LYAU|Sales Analytics Test
Error	TEMPLATE_PROCESSING_ERROR	Displays any template processing errors.	13:06:42.63 (2341994782) |TEMPLATE_PROCESSING_ERROR|CREATE |Application Sales Analytics Test (001RM000000gP2LYAU) failed
Fine	APP_CONTAINER_INITIATED	The template values and variables.	14:01:55.50 (15050452403) |APP_CONTAINER_INITIATED [redacted]
Fine	TEMPLATED_ASSET	The app asset status, which is created, updated, or deleted.	14:03:24.627 (15627342211) |TEMPLATED_ASSET|Successfully created/updated dashboard [Executive Overview(Exec_Overview_Pipeline_Performance1)]...
Fine	TRANSFORMATION_SUMMARY	The template asset name and type being transformed to an app asset.	14:03:25.684 (16692521613) |TRANSFORMATION_SUMMARY |ASSET_CREATE_UPDATE|Transforming template with name [Account_Summary] and type [dashboardTemplate]
Fine	RULES_EXECUTION_SUMMARY	The rule name, appliesTo types, actions, and any conditions.	14:03:15.266 (7081461905) |RULES_EXECUTION_SUMMARY |ASSET_CREATE_UPDATE, Processing action [Action {action='delete', path='$.workflowDefinition..[?(@.name=='HasRevenueSchedule')]', value='null', description='delete node from workflow'}]
Fine	ASSET_DIFF_SUMMARY	The summary of differences for an asset on an app reset or upgrade.	15:09:24.682 (9748463760) |ASSET_DIFF_SUMMARY|Action: UPDATE, Reason: DIFFERENCES_DETECTED, Asset ID: 0FKRM0000000boU4AQ, Label: Trending, Type: dashboard, Template: Trending (Sales_Rep_Pipeline_Changes)
Fine	JSON_DIFF_SUMMARY	The summary of differences for JSON on an app reset or upgrade.	15:09:21.493 (6865203874) |JSON_DIFF_SUMMARY|ASSET_DIFF |Found [106] differences between existing dashboard and new

Log Level	Apex Log Event	Details	Example Log Line
			`dashboard [0FKRM0000000boN4AQ] [Summary of Account]`
`Finer`	`RULES_EXECUTION_DETAIL`	The rule results for each action of a rule.	`14:03:35.872 (26905199214) |RULES_EXECUTION_DETAIL |ASSET_CREATE_UPDATE, set, $['state']['gridLayouts'][0]['pages'][0]['widgets'][33]['column'], 20`
`Finest`	`JSON_DIFF_DETAIL`	The detailed difference for JSON on an app upgrade or reset.	`15:09:24.682 (9729246304) |JSON_DIFF_DETAIL|ASSET_DIFF |Action {action='set', path='$.Extract_Account.parameters.fields[8].name', value='BillingPostalCode', description='Change 11: 'String values are not equal: new [BillingPostalCode] old [BillingCountry]''}`

Setting the `Wave` category log level to `Finest` includes all the log events in the log file.

Example: A sample `Analytics Templated App Poller` log file:



Note: The templated app debug log files are also visible via the Salesforce Dev Console after the debug trace flags are created.

STEP 6: SHARE THE TEMPLATE

Distribute the template to others in your Salesforce org or other orgs by deploying to the Metadata API. Others with the correct permissions can then retrieve the template object from the Metadata API and make their own contributions. See the Prerequisites section in this document.

- Create and install a first or second-generation package with `WaveTemplate` as the component type.
- For templates that create embedded apps via the auto-install framework, to use the install hooks, you must include the `auto-install.json` file in the managed package with the other template files. When your managed package is deployed in a customer org, the app is automatically created from your packaged template with an auto-install request.

  Each org can only allow 10 auto-install requests to run at one time. If an org has 10 requests that are in progress, any subsequent requests are marked as new and must be manually started from the Auto-Installed Apps setup page. Use this page to monitor auto-install requests and app creation statuses and possible failures.

  Embedded apps may also be installed using custom Apex pages to generate auto-install requests for app creation.

SEE ALSO:

First-Generation Managed Packages

Second-Generation Managed Packages

STEP 7: CREATE NEW (DOWNSTREAM) APPS FROM THE TEMPLATE

Users can now create apps using your template as follows:

1. Create a Standard App
   a. In CRM Analytics Studio, select **Create-->App**.
   b. Select the template you shared in the template picker.
   c. Follow the configuration wizard!

      If there were changes made to the template since the downstream app's last use, the user receives a prompt to upgrade the app anytime they open the app. It's important to accept this prompt to keep their downstream apps on the upgrade path and in sync with the source app and the template.

2. Create an Embedded App
   a. Install your managed package in a customer org.
   b. Verify the embedded app installed in the customer org using the Auto-Installed Apps setup page. Reference details in the Monitor, Update, and Delete Auto-Installed Apps help.
   c. To give users access to the embedded apps dashboard(s), use the Lightning App Builder and CRM Analytics Dashboard component to embed the dashboards(s) inside Salesforce. For details, see Embed CRM Analytics Dashboards in Lightning Pages.

STEP 8: UPDATE AN EXISTING TEMPLATE

Update a template after changing the source app's dashboards, lenses, and datasets in CRM Analytics Studio.

- Using Analytics Studio. Update the template. For more information, see Managing Your Templates.
- Using Analytics CLI. Use the *analytics template update* command.
- Using VS Code. Use the **Update Analytics Template** command.

Important: You update a template only after changing the source app in Analytics Studio. When you edit template JSON files in your IDE, you deploy those changes to the template using the CLI or VS Code *push* command. See Step 4, Deploy the WaveTemplate Object.

When you update the template, it's saved to your development org with a new version. For example, the version 1.0 becomes 2.0. You can then update the existing managed package for the template with the new version and deploy it. Admins managing standard apps based on the previous version of the template are prompted to upgrade their apps.

Upgraded Apps From New Versions Overwrite Customizations to Downstream Apps

Note: Overwriting customizations applies to standard apps only. Embedded apps can't be customized by users.

If the admin upgrades the app from a new template version, any customizations made to downstream apps are overwritten. To preserve customizations, we advise admins to save customized app assets before upgrading so they can copy the customizations into the new version.

Admins can elect to not upgrade from a new version and preserve the original downstream app with its customizations. To help admins determine the value of upgrading, provide details about the new template version in the `releaseNotes.html` file referred to by `template-info.json`.

Updating the Asset Version

When a template is created from an app, each asset is represented as a JSON file, generated at the current API version. The API version is stored as the `assetVersion` in `template-info.json`. When the template is installed in an org and used, in each subsequent release the Analytics assets are created using the template `assetVersion`, not the current API version. Updating the `assetVersion` allows the template to provide the latest features for the assets in the app.

Note: Updating a template without specifying the assetVersion downgrades assets to the `assetVersion` stored in `template-info.json`, even if the asset was created or updated at the current API version. Recipes are the only assets that are always created and updated with the current API version.

To update the API version of the template assets, the `assetVersion` must be specified on the template update call via the CLI. The `assetVersion` takes an `Integer` that is a valid Salesforce API version.

For the CLI, use the `-v` parameter to specify the `assetVersion`. For example, `analytics:template:update -v 54.0`.

Specifying the `assetVersion` on a template update impacts all the template assets, including dashboards, datasets, and dataflows. Recipe assets are always created with the current API version, regardless of the `assetVersion`.

We recommend template developers complete these steps when updating the `assetVersion`.

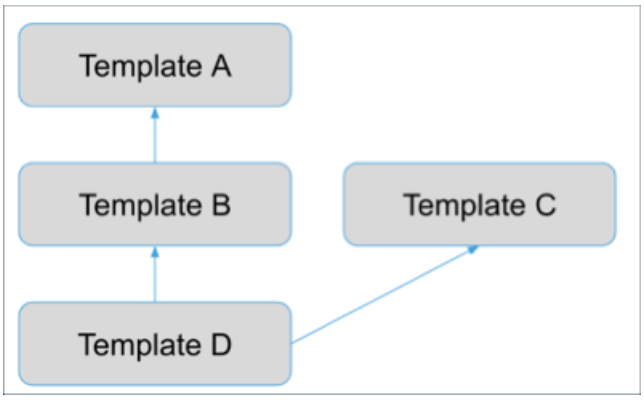
- Retain a copy of the template before using the update call, in case template assets fail to update as expected.
- Test the template update by creating apps in a test org. Verify that the updated assets work and have the new features for the version.

- If there are template rules, it's possible the JSON path attributes must be corrected to reflect any changes made by the `assetVersion` update. Test that all rules and any conditionals still work.

SHARE CRM ANALYTICS ASSETS BETWEEN APPS

Use the Template Dependencies feature to reuse CRM Analytics dashboards, datasets, and lenses across apps.

Create dependencies between app templates with Template Dependencies. Create the dependency from one app template (the dependent app template, Template D in the image) to assets in another (the supporting app template, Template A in the image) that reside in the same org. Apps created from the dependent template can include dashboards, datasets, and lenses from the supporting template. A template (Template B) can be both a dependent template and a supporting template at the same time.



Important: Template dependencies can't be used with templates packaged for automatic install. The creation of auto-install requests can't guarantee dependent templates are installed in the correct order.

While a template declares dependencies on other templates, it is really the apps that get created from the templates that have dependencies on each other. If Template B declares a dependency on Template A, it means that to create an app from Template B, you must first create one or more apps from Template A. The dependent and supporting apps must both reside in the same org.

Note: You can only create dependencies on apps created from templates. That's because the dependency refers to the template. There is template-to-template dependency in creating token references and app-to-app dependency when app assets are created from the dependent template.

Create Dependent Templates

Follow these steps to create templates with dependencies.

1. Create supporting app template. Supporting apps' dashboards, lenses, and datasets can be reused by other apps.
2. Add a line to your dependent app templates' `template-info.json` file that refers to templates to include in the dependency:

```
"templateDependencies" : [{
      "name" : "<Template_A_devname>",
      "namespace" : "<Template_A_mynamespace>",
      "templateVersion" : "1.0",
      // optional condition in which to include this dependency.
      "condition" : "${Variables.dependOnAnotherTemplate}"
    }]
```

3. In your templatized asset JSON files (`dashboard.json` and so on) based on the dependent template, use the supporting app's assets in context with the following:

```
${Apps.<templatename>.<Dashboards|Lenses|Datasets>.<assetSourceName>.<Name|Label|Alias|Id>}:
```

   Examples:

```
"${Apps.namespacedone__SupportingDashboard.Dashboards.Comparision_tp.Name}"
 "${Apps.namespacedone__SupportingDashboard.Datasets.MyDataset.Alias}"
```

4. Load all templates into the same org.
5. Create apps from the supporting templates.
6. Create the app from the dependent template. The requirements for creating a dependent app are:
   - The supporting app templates must exist in the org, and the user creating the dependent app must have access to them.
   - At least one app from the supporting template at the right version and namespace must exist and have a Completed status.

Template Dependency Syntax

The syntax you use to specify dependencies in the `template-info.json` file for `templateVersion` is flexible. The value can take on any of the following.

- Choose any version.
- A specific version.
- Versions greater than or less than the one indicated.

Here is the syntax:

1. If no version is specified, any version found is used. If a version is specified, but no operator (for example, 3.0), only that version is used.
2. If the operator is "=" (=3.0), only that version is used.
3. If the operator is ">" (>3.0), any app with a version greater than the specified version is used
4. If the operator is "<" (<3.0), any app with a version less than the specified version is used.
5. If the operator is ">=" (>=3.0), any app with a version greater than or equal to the specified version is used.
6. If the operator is "<=" (<=3.0), any app with a version less than or equal to the specified version is used.

Best Practices

When creating template dependencies, follow these guidelines.

1. Make the `templateVersion` as broad as possible.
2. Always include namespace.
3. For ease of maintenance, keep your template dependencies simple. There's no technical limitation to the number of dependencies you can create. But in practice, it's best to avoid creating multiple dependencies from one template to others. Also, avoid making supporting templates dependent on other templates.

Creating a Template from a Dependent App

Apps with dependencies on assets on other apps can be templatized using the same template creation process as standalone templates.

To create a template with dependencies on other apps, the supporting apps must be templatized first. If they aren't, the template creation process returns an error that reminds you to create a template for the supporting app. Aside from that consideration, use the same steps to templatize an app that has dependencies as a template that has no dependencies.

FEATURES NOT SUPPORTED IN THIS RELEASE

The following Analytics features are not supported by the Analytics Templates framework.

- Custom Maps and Charts
- Einstein Discovery count analysis prediction type

ANALYTICS TEMPLATES RELEASE NOTES

Use the Salesforce Release Notes to learn about the most recent updates and changes to Analytics Templates.

Analytics Templates use several different tool sets, including the CRM Analytics Connect REST API, the Salesforce Analytics CLI Plugin, Apex classes, and Lightning Web Components.

For a list of all current developer changes, see CRM Analytics in the Salesforce Release Notes.

Note: If the Analytics Development section in the Salesforce Release Notes isn't present, there aren't any updates for that release.

API END-OF-LIFE POLICY

Salesforce is committed to supporting each API version for a minimum of three years from the date of first release. In order to mature and improve the quality and performance of the API, versions that are more than three years old might cease to be supported.

When an API version is to be deprecated, advance notice is given at least one year before support ends. Salesforce will directly notify customers using API versions planned for deprecation.

Salesforce API Versions	Version Support Status	Version Retirement Info
Versions 31.0 through 64.0	Supported.	
Versions 21.0 through 30.0	As of Summer '25, these versions are retired and unavailable.	Salesforce Platform API Versions 21.0 through 30.0 Retirement
Versions 7.0 through 20.0	As of Summer '22, these versions are retired and unavailable.	Salesforce Platform API Versions 7.0 through 20.0 Retirement

If you request any resource or use an operation from a retired API version, REST API returns `410:GONE` error code.

To identify requests made from old or unsupported API versions of REST API, access the free API Total Usage event type.

APPENDIX

The reference section of this document contains detailed examples, attribute lists and descriptions, and other in-depth reference materials that build on the concepts we have discussed up to this point.

Add a Recipe to a CRM Analytics Template

Data prep recipes are a unique CRM Analytics asset, as they aren't stored in folders like dashboards, lenses, and datasets. The only way to connect a recipe to an app is via the relationship of the datasets in the folder to the recipes that consume or output those datasets.

Configure Recipe Execution

Data Prep recipes can be complex and add to app creation time if they sync and execute while the templated app is generating. Recipes can also change the shape of your existing data. The `executeCondition` attribute for recipe entries allows you to specify if a templated recipe syncs and executes during templated app creation or not.

Add an Einstein Discovery Story to a CRM Analytics Template

Einstein Discovery stories can be added to an app template by referring to the story in the `template-info.json` file.

Add a Remote Connector to a CRM Analytics Template

Add remote connectors to the digest node of a template dataflow by referencing connectors in `variables.json` and editing `ui.json` and `rules.json`.

Use Live Datasets in a CRM Analytics Template

Use live datasets in your CRM Analytics template with the `liveConnection` attribute to bring in live connections to Snowflake source tables. Creating a template from a source app with live datasets generates the dataset information, but there are considerations for successful app creation for your users.

Work with Embedded App Auto-Install Requests

Embedded app templates can create apps without any custom code, but you can add customizations to your embedded app lifecycle. Create Apex classes that use the CRM Analytics auto-installer to create and manage auto-install requests for your embedded app.

Feature Parameters for Analytics Templates

Feature parameters are available to control behavior in the template-to-app process for CRM Analytics apps. Feature parameters are defined as metadata that is deployed as part of the managed package. When feature parameters are deployed with the template, the parameter values are set using the LMA application.

template-info.json Attributes

The `template-info.json` file is the main file that describes the template. It includes or references all the information required to create a downstream app.

template-info.json Example

Here are two example `template-info.json` files.

ui.json Attributes

The `ui.json` file attributes are:

variables.json Attributes

Named nodes, each representing a single variable. Each node contains the following attributes:

org-readiness.json Attributes

The `org-readiness.json` file attributes are:

folder.json Attributes

The `folder.json` file attributes are:

auto-install.json Attributes

The `auto-install.json` file attributes are:

rules.json Example

Refer to this example of the `rules.json` file.

rules.json Attributes

The `rules.json` file attributes are:

Rules Testing with jsonxform/transformation endpoint

Test the results of a rule before deployment by calling the `jsonxform/transformation` endpoint.

VisualForce Events for Customizing the Wizard UI

This reference section explains the details of using VisualForce for customizing the Wizard user interface.

Add a Recipe to a CRM Analytics Template

Data prep recipes are a unique CRM Analytics asset, as they aren't stored in folders like dashboards, lenses, and datasets. The only way to connect a recipe to an app is via the relationship of the datasets in the folder to the recipes that consume or output those datasets.

When a recipe creates a dataset for a CRM Analytics app, the dataset connects the recipe to the folder ID for the app. These recipes are automatically added to the app template when it's created from the app. Only recipes that run successfully and create a dataset are added to the app template.

If a recipe doesn't contain a dataset, like a recipe that reads and writes from a connector, there's no way for the app framework to determine which app the recipe belongs to. To solve this issue, in v55.0 and above, the Connect REST API for creating and updating a template supports a list of recipe IDs in the request.

Note: Only the REST APIs and the Analytics CLI support creation with recipe IDs. Users can't specify recipe IDs when creating a template from an app in Analytics Studio.

When the app framework adds recipes to a template, it doesn't know the order in which the recipes must be run. On template creation, the app framework first adds any recipes specified in the request. Then the framework finds recipes used to create datasets for this app and adds them to the end of the list. If a duplicate recipe ID is specified, the app framework ignores it.

During app creation, the recipes are run in the order in which they're defined in the `template-info.json` file. If the recipes must run in specific order, you must edit the `template-info.json` file manually to put the recipes in run order.

Example: Use the Connect REST API

Use the template REST API to create a template from an app and include 2 recipes. In the POST request body, use the `recipeIds` parameter to specify any additional recipes to add to the app template.

```
POST /wave/templates/
{
    "folderSource": {
        "id": "00lxx000000xxxxxx"
    },
    "name": "myAppTemplate",
    "label": "My App Template",
```

```
    "description": "My app template with recipes",
    "recipeIds": [
        "05vxx000000xxxxx1",
        "05vxx000000xxxxx2"
    ]
}
```

When updating an existing template to add recipes, specify the `recipeIds` in the PUT request body.

```
PUT /wave/templates/0Nk1k000000xxxxxx
{
     "folderSource": {
         "id": "00lxx000000xxxxxx"
     },
     "recipeIds": [
         "05vxx000000xxxxx3"
     ]
}
```

This request adds another recipe to the existing template. If the recipe exists in the template, the PUT request does a no-op.

Note: The PUT only adds recipes to an existing template. If you need to remove a recipe, you must download the template and modify the `template-info.json` file manually.

Example: Use the Analytics CLI Plugin

First, get a list of recipes in your org:

```
sf analytics recipe list
```

returning:

```
RECIPEID           NAME          NAMESPACE           LABEL          STATUS
_________________ _____________ ____________________ _____________ ________
05vxx000000xxxxx1 Recipe_One                       Recipe One     New
05vxx000000xxxxx2 Recipe_Two                       Recipe Two     New
```

Then, create a template from an app and include the recipe IDs:

```
sf analytics template create -f 00lxx000000xxxxxx -l "My App Template" -r
"05vxx000000xxxxx1, 05vxx000000xxxxx2"
```

Note: When specifying multiple recipe IDs, they must be enclosed in quotes.

To add a recipe to an existing template:

```
sf analytics template update -t 0Nk1k000000xxxxxx -f 00lxx000000xxxxxx -r
05vxx000000xxxxx3
```

Configure Recipe Execution

Data Prep recipes can be complex and add to app creation time if they sync and execute while the templated app is generating. Recipes can also change the shape of your existing data. The `executeCondition` attribute for recipe entries allows you to specify if a templated recipe syncs and executes during templated app creation or not.

To specify the execution selection for a recipe entry in `template-info.json`, add the `executeCondition` attribute. The valid values for this attribute are:

- `CreateOnly`—Create the recipe during templated app creation, but don't sync or execute it.
- `CreateExecute`—Create and execute the recipe during templated app creation, but don't sync it.
- `CreateSyncExecute`—(Default) Create, sync, and execute the recipe during templated app creation.

The `executeCondition` attribute is optional. When not specified, the default behavior during templated app creation is to create, sync, and execute the recipe.

After app creation, all recipes created are visible in the Data Manager. You can verify a recipe was created correctly for your data, then configure it for data sync, execution, and scheduling.

Here are examples for recipe entries using `executeCondition` in `template-info.json`.

Example: Specify a recipe that is created at templated app runtime, but data sync and recipe execution aren't run.

```
"recipes": [
  {
    "label": "Sentiment Analysis Recipe",
    "name": "Sentiment_Analysis_Recipe",
    "file": "sentiment_analysis_recipe.json",
    "executeCondition": "CreateOnly"
  }
]
```

Example: Specify a recipe where the `executeCondition` is determined by an EL expression, like a variable value. The EL expression used must evaluate to one of the valid values.

```
"recipes": [
  {
    "label": "Contact Import Recipe",
    "name": "Contact_Import_Recipe",
    "file": "contact_import_recipe.json",
    "executeCondition": "${Variables.executionType}"
  }
]
```

Add an Einstein Discovery Story to a CRM Analytics Template

Einstein Discovery stories can be added to an app template by referring to the story in the `template-info.json` file.

Create the Story

Before adding the story to a template, create it using Einstein Discovery. (See Create a Story.) We recommend testing your story in CRM Analytics Studio before adding it to a template. Also, add the story JSON to the template. See the following section "Configure a Story" for instructions about editing the story JSON dynamically with a template rule.

Note: Templates only support Einstein Discovery stories with a numeric, binary, and multiclass analysis prediction types in the outcome. The count analysis prediction type isn't supported.

Reference the Story in `template-info.json`

To include the story in the template, add the following to `template-info.json`:

```
"extendedTypes":{
  "discoveryStories": [
    {
      "label": "AcquiredAccounts",
      "name": "acquired_accounts",
      "file": "acquired-accounts-story.json"
      "condition" : "${Variables.StringValueInclude == 'Yes'}"
    },
    {
      "label": "AcquiredAccounts2",
      "name": "acquired_accounts2",
      "file": "acquired-accounts-story2.json"
    }
  ]
}
```

To add multiple stories to a template, add a `discoveryStories` entry for each story.

Add a conditional to exclude or include the story based on a variable the user selects in the wizard, available org data, or some other condition.

Story Metadata Examples

Here's an example `acquired-accounts-story.json` file for a story with a non-Boolean outcome.

```
{
    "outcome": {
        "type": "Number",
        "label": "CLV",
        "name": "CLV",
        "goal": "Maximize"
    },
    "autopilot": {
        "enabled": true
    },
    "setup": {
        "dataset": {
            "id": "${App.Datasets.opportunities.Id}"
        },
        "outcome": {
            "type": "Number",
            "label": "CLV",
            "name": "CLV",
            "goal": "Maximize"
        },
        "fields": [{
                "type": "text",
                "name": "Account Id"
            },
            {
                "type": "text",
```

```
                "name": "BillingState"
            },
            {
                "type": "date",
                "name": "StartDate"
            },
            {
                "type": "number",
                "name": "example"
            }
            ...
        ]
    }
}
```

For a boolean outcome, use something like the following:

```
"outcome": {
     "failureValue": "False",
     "goal": "Maximize",
     "label": "IsWon",
     "name": "IsWon",
     "successValue": "True",
     "type": "Text"
},
"setup": {
    "outcome": {
        "type": "text",
        "name": "IsWon",
        "displayName": "IsWon",
        "success": "True",
        "failure": "False",
        "goal": "Maximize"
      },...
}
```

Syntax notes:

- The `id` value in `dataset` must match a dataset specified in `template-info.json`. You can only create one story per dataset in a template.
- Story metadata is case-sensitive. Capitalize the first letter in outcome, and use lowercase for fields.
- All fields are required in the `outcome` object.
- `outcome` and `setup.outcome` must be the same.
- The outcome name is the variable we want the story to identify. For example, in the above, we want to learn if the outcome is `IsWon`.
- Number of fields can be a minimum of 2 and maximum of 50
- Fields are the columns of data analyzed by Einstein Discovery and included in the story.
- The `autopilot` object is optional. The default value is `false`.

Configure a Story

You can add filters to fields in the story. In the following example, we only consider rows where IsClosed = true.

```
"fields": [
    {
       "type": "text",
       "name": "IsClosed",
       "includeOther": false,
       "values":
       [
         {
           "name" : "true",
           "ignored":false
         },
         {
           "name" : "false",
           "ignored":true
          }
       ]
    }
]
```

Use caution when adding date fields. Adding a field that shows individual dates (for example, 2019-07-01), isn't helpful in the story. Instead, it makes more sense to show months.

```
{
    "type": "date",
    "name": "CloseDate",
    "periodicIntervals": [
        "MONTH_OF_YEAR"
    ],
    "interval": "NONE"
}
```

Multivalue fields can't be included in a story. They're excluded from the list in the story UI and must not be added to the story json.

Create or alter the story JSON using template rules. For example, the following rule loops through a measures variable and adds each field to the story JSON:

```
{
  "condition": "${!empty(Variables.Dataset_Measures[0].fieldName)}",
  "action": "set",
  "description": "put selected fields in ED Story json",
  "path": "$.setup.fields",
  "value": "${array:union(Rules.CurrentNode,
array:forEach(Variables.Dataset_Measures,'{\"type\":\"number\", \"name\":
\"${var.fieldName}\"}'))}"
}
```

Here's a rule that conditionally adds a competitor name if it's selected in the wizard:

```
{
  "condition": "${!empty(Variables.Competitor.fieldName)}",
  "type": "text",
  "name": "${Variables.Competitor.fieldName}"
}
```

Here's another that adds only month of the year values to the story:

```
{
  "condition": "${!empty(Variables.Dataset_Dates[0].dateAlias)}",
  "action": "set",
  "description": "put selected fields in ED Story json",
  "path": "$.setup.fields",
  "value": "${array:union(Rules.CurrentNode,
array:forEach(Variables.Dataset_Dates,'{\"type\":\"date\", \"name\": \"${var.dateAlias}\",
 \"periodicIntervals\": [\"MONTH_OF_YEAR\"], \"interval\": \"NONE\"}'))}"
}
```

Add a Remote Connector to a CRM Analytics Template

Add remote connectors to the digest node of a template dataflow by referencing connectors in `variables.json` and editing `ui.json` and `rules.json`.

Using the `ConnectorType` Variable in `variables.json`

If your app includes a connection to a remote data source, add the connection to the template using a `ConnectorType` variable to `variables.json`. `ConnectorType` is a variation of the `StringType` variable with an added `connectorType` attribute. The attribute lets you specify types of connectors to add to the template dataflow, as follows:

```
"ExternalConnector" : {
  "label" : "my connector",
  "description" : "This will be the connector.",
  "variableType" : {
      "type" : "ConnectorType",
       "connectorType" : "SalesforceExternal"
  },
  "required" : true
}
```

The `connectorType` attribute must be a string. It corresponds to values in `WaveDataConnectorTypeEnum`. The value is a "starts with" instead of an "equals," so using Salesforce includes any connector beginning with that string, such as `SalesforceExternal`, `SalesforceMarketingCloud`, and `SalesforceMarketingCloudOAuth2`. Leave the `connectorType` value blank to include all defined connectors in the org. You can also use a `ConnectorType` variable as the `itemType` in an `ArrayType` variable.

After using the `ConnectorType` variable in `variables.json`, add the variable to `ui.json` so a user creating an app can choose connectors in the template wizard. The user's org must have at least one connector of the specified type already authenticated, but that connector doesn't need existing replicated objects. The wizard displays all remote connectors that match the `connectorType` attribute's criteria, as shown here.

Add a Rule to Replace the `connectionName` in the Dataflow

Use the `ConnectorType` variable in a rule to update the dataflow's digest node `connectionName` at app creation time. Here's an example rule:

```
{
"rules" : [
  {
     "name" : "UpdateConnectors",
     "appliesTo" : [ {
         "type" : "workflow",
         "name" : "*"
     }],
     "actions" : [
     {
         "action" : "set",
         "description" : "Use connector for digest node",
         "path" : "$..workflowDefinition.digest_Account.parameters.connectionName",
         "value" : "${Variables.ExternalConnector}"
     }]
  }]
}
```

Avoid using `${Variables.ExternalConnector}` directly in the template workflow JSON. Every time the template is updated from the source app, this variable token is overwritten and removed.

Use Live Datasets in a CRM Analytics Template

Use live datasets in your CRM Analytics template with the `liveConnection` attribute to bring in live connections to Snowflake source tables. Creating a template from a source app with live datasets generates the dataset information, but there are considerations for successful app creation for your users.

Live Datasets in template-info.json

Live datasets are generated when a template is created from the source app containing a live connection to Snowflake. Like a standard SFDC dataset, a live dataset contains a `name` attribute and an optional `label` attribute. Live datasets also contain a `liveConnection` attribute with the Snowflake source table information.

```
"datasetFiles": [
    {
        "name": "my_live_data",
        "label": "My Live Dataset",
        "liveConnection": {
            "connectionName": "myuniquelongconnectionname",
            "sourceObjectName": "Account"
        }
    }
]
```

Live Datasets in Templated Assets

When you create your template from your source app with a live dataset, the template generation tokenizes the live dataset, just as it does for standard SFDC datasets. Here's an example of a tokenized live dataset in a dashboard template file.

```
"connectionName" : "${App.Connections.myuniquelongconnectionname.Name}",
"datasets" : [
    {
        "name" : "${App.Datasets.my_live_data.Name}"
    }
],
```

Considerations for Templates with Live Datasets

For a user to create an app from your template, their consumer org must have an existing connection to Snowflake. This Snowflake connection must have a developer name with the exact value that is specified for the `connectionName` of the dataset entry in the template-info.json file. As a template developer, you must document that your template relies on a Snowflake connection with a specific name in order for app creation from the template to succeed. Prompt your users to verify a Snowflake connection on a custom Visualforce page that you can create and use in the configuration wizard.

Warning: If a Snowflake connection developer name doesn't match the `connectionName` in the consumer org, app creation from the template fails. The app creation notification email notifies the user of the failure.

To clarify that a Snowflake connection is for use in a templated app, it's recommended that you create a unique, longer `connectionName` value. The previous examples highlight using a unique and long connection name.

When an app instance is created from your template, the connection information is cloned for each instance. When an app instance is deleted, the cloned connection associated with that app is also deleted, as long as no other live datasets are referencing the connection. The original Snowflake connection created to support templated app instances isn't deleted, in case other app instances are using the connection.

SEE ALSO:

Create an App

Work with Embedded App Auto-Install Requests

Embedded app templates can create apps without any custom code, but you can add customizations to your embedded app lifecycle. Create Apex classes that use the CRM Analytics auto-installer to create and manage auto-install requests for your embedded app.

What Is the CRM Analytics Auto-Installer?

The CRM Analytics auto-installer is a metadata-based API that manages the lifecycle of a CRM Analytics templated app. No user access to CRM Analytics Studio and the template configuration wizard is required.

The auto-installer provides management functionality for embedded apps. Creating customized runtime functionality requires knowledge of Apex. Auto-install requests can also be managed via Analytics CLI Plugin commands or using Connect API REST calls.

Limits and Restrictions

Concurrent Auto-Install Requests

The auto-installer uses asynchronous processing that allows a maximum of 10 `Enqueued`, `InProgress`, or `AppInProgress` requests at a time. Until existing requests complete their cycle or their `RequestStatus` is updated to `Cancelled`, no more requests can transition from `New` to `Enqueued`.

App-Specific Concurrent Requests

No app-specific auto-install `WaveAppDelete`, `StartDataflow`, `WaveAppUpdate` requests can be `Enqueued` until existing requests that are in progress are completed or canceled with the same folder id.

Cancellation Policy

If an auto-install request is canceled, the auto-installer attempts to abort the request, but there's no guarantee when or if the request can be aborted.

Auto-Installer Use Cases

These use cases show how to create customized auto-install app management using Apex code. Examples include enabling CRM Analytics in the install org, configuring the app creation settings, and creating and updating embedded apps.

CRM Analytics Enablement

Any use of a CRM Analytics templated app requires the CRM Analytics preference to be enabled before an app can be created and managed.

The CRM Analytics auto-installer can automatically enable this preference so any other auto-installer task can be taken. The auto-installer can enable or disable any of these preferences based on the Auto-Installer configuration specified by the API caller:

- CRM Analytics Replication
- Sharing Inheritance

Apex Example

```
Map<String, Object> configuration = new Map<String, Object>();
Map<String, Object> waveEnableConfiguration = new Map<String,Object>();
waveEnableConfiguration.put('replicationEnabled', true);
waveEnableConfiguration.put('templatesEnabled', true);
waveEnableConfiguration.put('inheritedSharingEnabled', true);
configuration.put('waveEnableConfiguration', waveEnableConfiguration);
WaveAutoInstallRequest newRequest = new WaveAutoInstallRequest();
newRequest.put('Name', 'Test WaveEnable request');
newRequest.put('RequestStatus', 'Enqueued');
newRequest.put('RequestType', 'WaveEnable');
newRequest.put('Configuration', JSON.serialize(configuration));
insert newRequest;
```

For information on auto-install requests and auto-install configuration metadata, see:

- `WaveAutoInstallRequest` reference
- `AutoInstallAppConfigurationRepresentation` reference.

App Create

The `WaveAppCreate` auto-install task creates a CRM Analytics app based on the `TemplateApiName` and values specified by the caller.

After the app creation process is complete, the auto-install request is updated automatically by the app template framework. If `deleteAppOnConstructionFailure` is set to true and the app fails to create, the app is automatically deleted.

```
Map<String, Object> configuration = new Map<String, Object>();
Map<String, Object> values = new Map<String, Object>();
values.put('myVariableName1','someValue');
Map<String, Object> appConfiguration = new Map<String,Object>();
appConfiguration.put('deleteAppOnConstructionFailure', true);
appConfiguration.put('values', values);
configuration.put('appConfiguration', appConfiguration);
WaveAutoInstallRequest newRequest = new WaveAutoInstallRequest();
newRequest.put('Name', 'Test WaveAppCreate request');
newRequest.put('TemplateApiName', 'template_test');
newRequest.put('RequestStatus', 'Enqueued');
newRequest.put('RequestType', 'WaveAppCreate');
newRequest.put('Configuration', JSON.serialize(configuration));
insert newRequest;
```

App Update

The `WaveAppUpdate` auto-install task performs an upgrade or reset on an existing CRM Analytics app based on the `TemplateApiName`, the `FolderId` of the app, and values specified by the caller.

After the app update is completed, the app template framework automatically updates the auto-install request. If `deleteAppOnConstructionFailure` is set to true and the app fails to update, the app is automatically deleted.

```
Map<String, Object> configuration = new Map<String, Object>();
Map<String, Object> values = new Map<String, Object>();
values.put('myVariableName1','someValue');
Map<String, Object> appConfiguration = new Map<String,Object>();
appConfiguration.put('deleteAppOnConstructionFailure', true);
appConfiguration.put('values', values);
configuration.put('appConfiguration', appConfiguration);
WaveAutoInstallRequest newRequest = new WaveAutoInstallRequest();
newRequest.put('Name', 'Test WaveAppUpdate request');
newRequest.put('TemplateApiName', 'template_test');
newRequest.put('RequestStatus', 'Enqueued');
newRequest.put('FolderId', '<folder_id>');
newRequest.put('RequestType', 'WaveAppUpdate');
newRequest.put('Configuration', JSON.serialize(configuration));
insert newRequest;
```

App Delete

The `WaveAppDelete` auto-install task simply deletes an existing CRM Analytics app based on the `TemplateApiName` and `FolderId` that the caller specifies.

No auto-install requests can be in progress for the specified folder id when the delete request is enqueued.

```
WaveAutoInstallRequest newRequest = new WaveAutoInstallRequest();
newRequest.put('Name', 'Test WaveAppDelete request');
newRequest.put('RequestStatus', 'Enqueued');
newRequest.put('FolderId', '<folder_id>');
newRequest.put('RequestType', 'WaveAppDelete');
// Not required, but best practice is to add the attribute so that related
// requests are aggregated together by template api name and folder.
```

```
newRequest.put('TemplateApiName', 'template_test');
insert newRequest;
```

Start Dataflow

The `StartDataflow` auto-install task executes a dataflow for the specified folder. Dataflow execution is only allowed for apps that are based on an Embedded App template.

After the dataflow is completed, the templating framework automatically marks the auto-install request as completed.

```
WaveAutoInstallRequest newRequest = new WaveAutoInstallRequest();
newRequest.put('Name', 'Test StartDataflow request');
newRequest.put('RequestStatus', 'Enqueued');
newRequest.put('FolderId', '<folder_id>');
newRequest.put('RequestType', 'StartDataflow');
insert newRequest;
```

Guaranteed Execution Order

The CRM Analytics Auto-Install Request framework is dependent on asynchronous processing, making it impossible to assure ordering of multiple auto-install requests.

To create request ordering, that auto-installer uses auto-install request dependencies, which guarantee that one auto-install request runs and finishes before another related auto-install request executes. Auto-install request dependencies don't guarantee that dependencies are executed before other enqueued requests that aren't part of the dependency chain.

Auto-install requests that have dependencies on other requests must have a status of `New` to be auto-enqueued by the framework. Dependencies with any other status are ignored.

If any auto-install request in the request hierarchy fails for any reason (including canceled), subsequent related requests auto-fail as well.

Request dependencies have these limits:

- A single auto-install request can depend on up to 10 other auto-install requests, directly or indirectly through the dependency chain.
- All parent auto-install requests must exist and be in `New` status before the child can declare a dependency on a parent request.

```
//Create first request with a new status (do not enqueue yet)
sObject firstRequest = Schema.getGlobalDescribe().get('WaveAutoInstallRequest').newSObject();
firstRequest.put('Name', requestName);
firstRequest.put('RequestStatus', 'New');
firstRequest.put('RequestType', 'WaveAppCreate');
firstRequest.put('TemplateApiName', 'template_test');
insert firstRequest;

//Create second request with a new status. This is enqueued after the first request is
done.
sObject secondRequest =
Schema.getGlobalDescribe().get('WaveAutoInstallRequest').newSObject();
secondRequest.put('Name', 'Dependant Request');
secondRequest.put('RequestStatus', 'New');
secondRequest.put('RequestType', 'WaveAppCreate');
secondRequest.put('TemplateApiName', 'template_test');

//Set up configuration for the second request, containing the id from the first
Map<String, Object> configuration = new Map<String, Object>();
configuration.put('parentRequestIds', new List<String>{firstRequest.Id});
secondRequest.put('Configuration', JSON.serialize(configuration));
```

```
insert secondRequest;

//update the first request to enqueue.
firstRequest.put('RequestStatus', 'Enqueued');
update firstRequest;
```

Monitoring Auto-Install Requests

To monitor auto-install requests, use the Auto-Installed App page in Setup. For more information, see Monitor, Update, and Delete Auto-Installed Apps.

Working with Auto-Install Requests

You can also create and manage auto-install requests using the Salesforce Analytics Plugin CLI and CRM Analytics REST API endpoints.

To use the Analytics Plugin auto-install commands, see the autoinstall Commands reference.

To use the CRM Analytics REST API, see the Auto-Install Requests List Resource and the Auto-Install Requests Resource references.

Feature Parameters for Analytics Templates

Feature parameters are available to control behavior in the template-to-app process for CRM Analytics apps. Feature parameters are defined as metadata that is deployed as part of the managed package. When feature parameters are deployed with the template, the parameter values are set using the LMA application.

For more information on feature parameters, see Feature Parameter Metadata Types and Customs Objects in the First-Generation Managed Packaging Developer Guide.

For implementation examples, view and clone the Project Force App in Github.

Analytics Template Metadata

For Analytics templates, the managed package component is `WaveTemplateBundle`. The template metadata reads feature parameter values set in the LMA application as EL expressions and uses them to control the behavior of app creation. The supported types of feature parameter values are `Boolean`, `Date`, and `Integer`.

Example: Use an `Integer` Feature Parameter Value in Dashboard Creation

For this example, the feature parameter is deployed using `/featureParameters/FinancingAmount.featureParameterInteger-meta.xml`.

```
<FeatureParameterInteger xmlns="http://soap.sforce.com/2006/04/metadata">
   <dataflowDirection>LmoToSubscriber</dataflowDirection>
   <masterLabel>Amount Financed</masterLabel>
   <value>250000</value>
</FeatureParameterInteger>
```

The feature parameter metadata is referenced in the `template-to-app-rules.json` file:

```
{
  "constants":[
    {
      "name": "FinancingAmount",
```

```
      "value": "${sfdc_fma:checkInteger('[namespace__FinancingAmount')}"
    }
  ],
  "rules": [
    {
      "name": "Use Feature Parameter",
      "appliesTo": [
        {
          "type": "dashboard"
        }
      ],
      "actions": [
        {
          "action": "set",
          "path": "$.state..text_1.parameters.content.displayTemplate",
          "value": "${Constants.FinancingAmount}"
        }
      ]
    }
  ]
}
```

template-info.json Attributes

The `template-info.json` file is the main file that describes the template. It includes or references all the information required to create a downstream app.

Attribute	Required	Description
`name`	Yes	The unique developer name of the template. If changed, it creates a new template; if it's changed to an existing template `devName`, it overwrites the existing template.
`label`	Yes	The template name visible to users in the Analytics template picker and wizard.
`description`	No	The template description. Shown in the app creation wizard template picker.
`icons`	No	Three types available: • `templateBadge` defines the small icon that appears in the app picker and at the top of the template details window. **Don't** use file extension with `name` attribute, for example `"learn"`, **not** `"learn.png"`. • `appBadge` defines the icon shown by Analytics Studio for the app after creation. Must be `"1.png"` through `"21.png"`. Use file extension with `name` attribute, for example `"16.png"`, **not** `"16"` • `templatePreviews` defines preview images displayed on the template details window during the app creation process. Supports multiple images. **Don't** use file extension with `name` attribute, for example `"preview_education"`, **not** `"preview_education.png"`. Name of the icon image must use all lower case letters.

Attribute	Required	Description
		If images are stored as static resources and org uses namespace, must also specify a `namespace` attribute for each icon. For example, `"appBadge": { "name": "15.png" "namespace": "${Org.Namespace}" }`
`tags`	No	Array of tags used in template search and shown on template details page.
`assetVersion`	Yes	Refers to the numbered API version to use when creating Analytics assets, such as dashboards, datasets, XMDs, and dataflows. Recipes are always on the current API version. To update this version, see Update an Existing Template.
`releaseInfo`	Yes	Container attribute only.
`releaseInfo template Version`	Yes	The version of the template, tracked for the upgrade process. Calling wave/templates PUT automatically increments this number, but it can be changed manually as well. The format is `##.##`, although it's a string.
`releaseInfo notesFile`	No	A template dev defined HTML file to describe new release details. Basic HTML formatting is supported, but not links or javascript. Defaults to using the template description as release notes if file isn't specified.
`rules`	No	Container attribute only - can contain 1 or more rule file definitions (loaded first to last and order matters for dependencies).
`rules type`	Yes, if `rules` is used.	appToTemplate - rules to apply when updating the template from the source app (Rest API PUT call). templateToApp - rules to apply when creating an app from a template.
`rules file`	Yes	Path to the rules file.
`templateType`	Yes	Can specify either `app`, `embeddedapp`, `data`, or `dashboard`.
`uiDefinition`	No	Path the `ui.json` file.
`variable Definition`	No	Path to the `variables.json` file.
`folder Definition`	No	Path to the `folder.json` file.
`autoInstall Definition`	Yes for auto-installed templates, No for templates that use the configuration wizard	Path to the `auto-install.json` file. Only specify this file for auto-install templates.
`readiness Definition`	No	Path to the `orgReadiness.json` file.

Attribute	Required	Description
`apexCallback`	No	Container attribute only. Registers an Apex class that runs on wizard load, app create, and app upgrade. The Apex class examines the org and sets variable values. For example, the Sales Analytics template uses an Apex class to check whether the org uses the Products object. If it doesn't, the `has_product` variable is set to `no`. **Important:** Make sure the Apex class can be accessed by users in the organization with Manage Analytics Templated Apps user permission. Otherwise, trying to create an app from your template results in an error. In Setup, go to **Apex Classes**, click **Security** next to the class, then add the profile with the Manage Analytics Templated Apps user permission.
`apexCallback namespace`	Yes, if `apexCallback` is used and org uses namespace.	The namespace for the org, if it has one. Required for Apex classes in the development org or packaged with the template.
`apexCallback name`	Yes	The name of the Apex class to be used.
`custom Attributes`	No	Specifies information describing your template in the app creation wizard, such as Salesforce objects used or notable features. Takes a label and an array of values. Example: `customAttributes" : [ { "label" : "Features", "values" : ["CUSTOM OBJECTS", "MOBILE READY"] }, {"label": "Salesforce objects", "values" : ["ACCOUNT","PRODUCT","LEADS",”CASES”] }],`
`dashboards`	No	Container attribute only. Array of dashboard JSON files to be used, can be empty if there are no dashboard files.
`dashboards file`	Yes, if `dashboards` is used. Otherwise no.	Path to the dashboard file.
`dashboards label`	Yes, if `dashboards` is used. Otherwise no.	The dashboard label. Must match the “label” attribute in the dashboard file
`dashboards name`	Yes, if `dashboards` is used. Otherwise no.	The dashboard name. Must match the “name” attribute in the dashboard file.
`dashboards condition`	No	If condition is met, then the dashboard is created in the app; if the condition isn't met, the dashboard isn't created. Can include `Overrides`, which are defined in `variables.json`. See Complex `variables.json` Variable Types on page 31.

Attribute	Required	Description
`dashboards → overwriteOn Upgrade`	No	Indicates if asset is overwritten during the reset/upgrade process. Uses `Always`, `IfDifferent`, `Never`, or an expression that evaluates to one these values. Case sensitive. Example: `"overwriteOnUpgrade": "${Variables.DontOverwriteQuota ? 'Never' : 'IfDifferent'}",`
`lenses`	No	Container attribute only - array of dashboard JSON files to be used, can be empty if there are no dashboard files.
`lenses file`	Yes, if `lenses` is used. Otherwise no.	Path to the lens file.
`lenses label`	Yes, if `lenses` is used. Otherwise no.	The lens label. Must match the "label" attribute in the lens file.
`lenses name`	Yes, if `lenses` is used. Otherwise no.	The lens name. Must match the "name" attribute in the lens file.
`lenses condition`	No	If the condition is met, then the lens is created in the app; if the condition isn't met, the dashboard isn't created. Can include `Overrides`, which are defined in `variables.json`. See Complex `variables.json` Variable Types on page 31
`lenses → overwriteOn Upgrade`	No	Indicates if asset is overwritten during the reset/upgrade process. Uses `Always`, `IfDifferent`, `Never`, or an expression that evaluates to one these values. Case sensitive. Example: `"overwriteOnUpgrade": "${Variables.DontOverwriteQuota ? 'Never' : 'IfDifferent'}",`
`eltDataflows`	No	Container attribute only – array of dataflow JSON files to be used, can be empty if there are no dataflow files.
`eltDataflows file`	Yes, if `eltDataflows` is used. Otherwise no.	Path to the dataflow file.
`eltDataflows label`	Yes, if `eltDataflows` is used. Otherwise no.	The dataflow label. Must match the `sourceLabel` attribute in dataflow file.
`eltDataflows name`	Yes, if `eltDataflows` is used. Otherwise no.	The dataflow name. Must match the `name` attribute in the dataflow file.
`eltDataflows condition`	No	If condition is met, then the dataflow is created in the app; if the condition isn't met, the dataflow isn't created. Can include `Overrides`, which are defined in `variables.json`. See Complex `variables.json` Variable Types on page 31

Attribute	Required	Description
`eltDataflows overwriteOnUpgrade`	No	Indicates if asset is overwritten during the reset/upgrade process. Uses `Always`, `IfDifferent`, `Never`, or an expression that evaluates to one these values. Case sensitive. Example: `"overwriteOnUpgrade": "${Variables.DontOverwriteQuota ? 'Never' : 'IfDifferent'}",`
`recipes`	No	Container attribute only – array of recipes JSON files to be used, can be empty if there are no recipe files.
`recipes file`	Yes, if `recipes` is used. Otherwise no.	Path to the recipe file.
`recipes label`	Yes, if `recipes` is used. Otherwise no.	The recipe label.
`recipes name`	Yes, if `recipes` is used. Otherwise no.	The recipe name.
`recipes condition`	No	If condition is met, then the recipe is created in the app; if the condition isn't met the recipe isn't created. Can include `Overrides`, which are defined in `variables.json`. See Complex `variables.json` Variable Types on page 31
`recipes overwriteOn Upgrade`	No	Indicates if asset is overwritten during the reset/upgrade process. Uses `Always`, `IfDifferent`, `Never`, or an expression that evaluates to one these values. Case sensitive. Example: `"overwriteOnUpgrade": "${Variables.DontOverwriteQuota ? 'Never' : 'IfDifferent'}",`
`externalFiles`	Yes	Container attribute only - array of external files, CSV and descriptive files, to be used, can be empty if there are no external files.
`externalFiles file`	No	Path to the CSV file.
`externalFiles name`	Yes, if `externalFiles` is used. Otherwise no.	The externalFile name.
`externalFiles rows`	No	The number of rows to stub out in the dataset created from the CSV, before data upload. If not specified, defaults to 5. By default, a file has 5 rows until the user uploads their own version of the file.
`externalFiles schema`	No	Path to the schema file (describes the format of the data)
`externalFiles userXmd`	No	Path to the user XMD file (describes the display format of the data).
`externalFiles type`	Yes	Must be `CSV`.

Attribute	Required	Description
`externalFiles condition`	No	If condition is met, then the external file is created in the app; if the condition isn't met, the external file isn't created. Can include `Overrides`, which are defined in `variables.json`. See Complex `variables.json` Variable Types on page 31
`externalFiles overwriteOn Upgrade`	No	Indicates if asset is overwritten during the reset/upgrade process. Uses `Always`, `IfDifferent`, `Never`, or an expression that evaluates to one these values. Case sensitive. Example: `"overwriteOnUpgrade": "${Variables.DontOverwriteQuota ? 'Never' : 'IfDifferent'}",`
`datasetFiles`	No	Container attribute only – array of SFDC dataset to be used, can be empty if there are no SFDC dataset.
`datasetFiles userXmd`	No	Path to the user XMD file, if it exists.
`datasetFiles label`	Yes	The dataset label.
`datasetFiles name`	Yes	The dataset name.
`datasetFiles condition`	No	If condition is met, then the dataset is created in the app; if the condition isn't met, the dataset isn't created. Can include `Overrides`, which are defined in `variables.json`. See Complex `variables.json` Variable Types on page 31
`datasetFiles live Connection`	No	The container attribute for Snowflake connection live dataset settings.
`datasetFiles live Connection connection Name`	Yes, if `liveConnection` is used. Otherwise no.	The connection name for the Snowflake connector. This name must match the developer name of the connector in the consumer org.
`datasetFiles live Connection sourceObject Name`	Yes, if `liveConnection` is used. Otherwise no.	The Snowflake source table name.
`imageFiles`	No	Container attribute only: array of app images to be used.
`imageFiles label`	Yes, if `imagesFiles` is used. Otherwise no.	A label for the image.
`imageFiles name`	Yes, if `imagesFiles` is used. Otherwise no.	The image name.

Attribute	Required	Description
`imageFiles condition`	No	If the condition is met, then the image is uploaded into the app; if the condition isn't met, the image isn't uploaded.
`imageFiles namespace`	Yes, if org uses a namespace.	Includes reference to namespace for static resources, for example `${Org.Namespace}`.
`extendedTypes`	No	Add extended type to a template.
`extendedTypes discovery Stories`	No	Add Einstein Discovery story extended type to template. Uses `label`, `name`, `file`, `condition`. See Add an Einstein Discovery Story to an Analytics Template
`template Dependencies`	No	Array of other templates the template is dependent on. See Share Analytics Assets Between Apps on page 103
`template Dependencies → name`	Yes, if `dependencies` is used. Otherwise no.	Developer name of the template.
`template Dependencies → namespace`	No	Namespace of the org, if used.
`template Dependencies → templateVersion`	No	Number of the template version.
`template Dependencies → condition`	No	Condition in which to include this dependency.
`components`	No	Container attribute only. Array of dashboard component JSON files to be used, can be empty if there are no dashboard components files.
`components file`	Yes, if `components` is used. Otherwise no.	Path to the dashboard component file.
`components label`	Yes, if `components` is used. Otherwise no.	A label for the dashboard component.
`components name`	Yes, if `components` is used. Otherwise no.	The dashboard component name.

Attribute	Required	Description
`components condition`	No	If the condition is met, then the dashboard component is created in the app. If the condition isn't met, the dashboard component isn't created.

SEE ALSO:

Edit template-info.json

template-info.json Example

template-info.json Example

Here are two example `template-info.json` files.

Example:

```
{
 "templateType": "app",
 "label": "Generic Analytics",
 "name": "GenericAnalytics",
 "description": "The Generic Analytics template creates an app that visualizes business
 data. v1.0 #160",
 "assetVersion": 43,
 "releaseInfo": {
  "templateVersion": "1.0",
  "notesFile": "releaseNotes.html"
 },
 "variableDefinition": "variables.json",
 "uiDefinition": "ui.json",
 "readinessDefinition": "org-readiness.json",
 "apexCallback": {
  "namespace": "wavetemplate",
  "name": "GenericConfigurationModifier"
 },
 "icons": {
  "appBadge": {
   "name": "15.png"
  },
  "templateBadge": {
   "name": "Generic.png"
  },
  "templatePreviews": [{
   "name": "preview_Generic.jpg",
   "label": "Generic Analytics",
   "description": "Explore impact to sales."
  }]
 },
 "tags": [
  "Marketing",
  "Sales"
 ],
 "customAttributes": [{
```

```
  "label": "Features",
  "values": [
   "Prebuilt Dashboard",
   "Mobile Ready",
   "Dataflow",
   "KPI Rich Datasets"
  ]
 },
 {
  "label": "Salesforce Objects",
  "values": [
   "Customizable Generic Influence"
  ]
 },
 {
  "label": "Publisher",
  "values": [
   "Analytics"
  ]
 }
],
"rules": [{
  "type": "templateToApp",
  "file": "template-to-app-rules.json"
 },
 {
  "type": "appToTemplate",
  "file": "app-to-template-rules.json"
 }
],
"externalFiles": [],
"lenses": [],
"dashboards": [{
 "file": "dashboard/Generic_Influence.json",
 "name": "Generic_Influence",
 "label": "Generic Influence"
}],
"eltDataflows": [{
 "label": "GenericAnalyticsDataflow",
 "name": "GenericAnalyticsDataflow",
 "file": "dataflow/GenericAnalyticsDataflow.json"
}],
"datasetFiles": [{
  "label": "GenericInfluence",
  "name": "GenericInfluence",
  "userXmd": "dataset_files/GenericInfluence_XMD.json"
 },
 {
  "label": "Generic",
  "name": "Generic",
  "userXmd": "dataset_files/Generic_XMD.json"
 }
],
"imageFiles": [{
```

```
  "name": "GenericLogo",
  "file": "images/Generic_logo.png"
  "namespace": "${Org.Namespace}" //if org is namespaced
 }]
}
```

Example:

```
{
 "templateType": "app",
 "label": "Corporate Sales Results",
 "name": "My_Sales",
 "description": "Create an app from this template to get actionable insights about the
 sales team.",
 "assetVersion": 46.0,
 "variableDefinition": "variables.json",
 "uiDefinition": "ui.json",
 "rules": [{
   "type": "templateToApp",
   "file": "template-to-app-rules.json"
  },
  {
   "type": "appToTemplate",
   "file": "app-to-template-rules.json"
  }
 ],
 "releaseInfo": {
  "templateVersion": "1.1",
  "notesFile": "releaseNotes.html"
 },
 "folderDefinition": "folder.json",
 "autoInstallDefinition": "auto-install.json",
 "externalFiles": [],
 "lenses": [],
 "dashboards": [{
   "label": "Sales",
   "name": "Sales_SAMPLE_tp",
   "condition": "${Variables.Overrides.createAllDashboards}",
   "file": "dashboards/DTC_Sales_SAMPLE.json"
  },
  {
   "label": "Opportunity Details",
   "name": "Opportunity_Details_tp",
   "condition": "${Variables.Overrides.createAllDashboards}",
   "file": "dashboards/Opportunity_Details.json"
  },
  {
   "label": "Regional Sales Results",
   "name": "Regional_Sales_SAMPLE_tp",
   "condition": "${Variables.Overrides.createAllDashboards}",
   "file": "dashboards/Regional_Sales_SAMPLE.json"
  },
  {
   "label": "Sales Performance with Selectable Measures",
```

```
  "name": "Sales_Performance_with_Selectable_Measures_Trailhead_tp",
  "condition": "${Variables.Overrides.createAllDashboards}",
  "file": "dashboards/Sales_Performance_with_Selectable_Measures_Trailhead.json"
 }
],
"recipes": [{
  "label": "Filter Accounts Recipe",
  "name": "Filter_Accounts_Recipe",
  "file": "recipes/Filter_Account_Recipe.json"
 }
],
"datasetFiles": [{
  "label": "Opportunity",
  "name": "Opportunity_SAMPLE_tp",
  "condition": "${Variables.Overrides.createAllDatasetFiles}",
  "userXmd": "dataset_files/Opportunity_SAMPLE_XMD.json"
 }
],
"components": [{
  "label": "Image Component",
  "name": "ImageComponent_tp",
  "condition": "${Variables.Overrides.createAllComponents}",
  "file": "components/ImageComponent.json"
 }
],
"imageFiles": [{
  "name": "LayingOutjpg",
  "condition": "${Variables.Overrides.createAllImages}",
  "file": "images/LayingOut.jpg"
  "namespace": "${Org.Namespace}" //if org is namespaced
 },
 {
  "name": "png1",
  "condition": "${Variables.Overrides.createAllImages}",
  "file": "images/head.png"
  "namespace": "${Org.Namespace}" //if org is namespaced
 }
],
"extendedTypes": {},
"templateDependencies": [],
"icons": {
 "appBadge": {
  "name": "3.png"
 },
 "templateBadge": {
  "name": "dtc_sales"
 },
 "templatePreviews": [{
   "name": "home_dashboard",
   "label": "The Sales Home Dashboard",
   "description": "Get an overview of your sales team and dig into the key performance
indicators that matter most to you."
  },
  {
```

```
    "name": "sales_results",
    "label": "Sales Performance with Selectable Measures",
   "description": "Key metrics about your sales team's performance, including details
 for each team member."
   }
 ]
},
"tags": [
 "Sales",
 "Team leaders",
 "Operations"
],
"customAttributes": [{
  "label": "Features",
  "values": [
   "Prebuilt apps",
   "External CSV data",
   "Mobile ready"
  ]
 },
 {
  "label": "Data Source",
  "values": [
   "External CSV data",
   "Salesforce Opportunities Object"
  ]
 },
 {
  "label": "Developer",
  "values": [
   "Manny"
  ]
 }
]
}
```

SEE ALSO:

template-info.json Attributes

ui.json Attributes

The `ui.json` file attributes are:

Attribute	Required	Description
`pages`	No	Container attribute only - array of UI pages to be displayed.
`pages title`	Yes	String title to be displayed at the top of the page.

Attribute	Required	Description
`pages variables`	Yes	Array of variables to display on a page, 1 variable is necessary.
`pages variables name`	Yes	Variable name, must match name of variable defined in variables.json.
`pages variables visibility`	No	Variable visibility (visible, disabled, hidden) defined by conditional.
`pages condition`	No	If condition is met, then the page will display; if the condition is not met, the page will not display.
`pages helpUrl`	No	URL provided by template dev to additional content to help user with creating app.
`pages vfPage`	No	Container attribute only: defines the VisualForce Page to customize this page.
`pages vfPage name`	Yes	The name of VisualForce class to use.
`pages vfPage namespace`	Yes, if org uses namespace.	The namespace for the org.
`displayMessages`	No	Container attribute only - array of custom display message, currently only 1 `displayMessage` is supported.
`displayMessages text`	Yes	Text to display on the app creation progress landing page (running Astro page).
`displayMessages location`	Yes	`"AppLandingPage"` is only supported location.

SEE ALSO:

Edit ui.json

variables.json Attributes

Named nodes, each representing a single variable. Each node contains the following attributes:

`variable` Attributes

Attribute	Required	Description
`computedValue`	No	The computed value for the variable. The computed value is set by the `WaveTemplateConfigurationModifier` implementation.

Attribute	Required	Description
`description`	No	Additional text that appears under the label for more information.
`defaultValue`	No	The default value for the variable, if the user doesn't change the value in the wizard. Can be the most likely value a user would use.
`excludeSelected`	No	Indicates whether values selected for another variable are excluded from the list of values for this variable () or not (). For use with variable types like `SObjectType` or `SObjectFieldType`. The default is `false`.
`excludes`	No	Allows specific values to be excluded from the picker selections
`label`	No	The visible label for the variable in the UI wizard. The format is typically a question like "Do you use product?".
`required`	No	Indicates whether the variable value required in the UI. The default is `false`.
`variableType`	No	The variable type.

`variableType` Attributes

Attribute	Required	Description
`dataType`	No	Specifies the data type for a FieldType variable types. The data type value format for `SObjectFieldType` is `xsd:string`. See Primitive Data Types in the *Salesforce Object Reference* for more values. For `DataLakeObjectFieldType` and `DataModelObjectFieldType`, valid values are `string`, `number`, `date`, and `date_time`.
`fieldType`	No	Specifies the field type for a `CalculatedInsightsFieldType` variable. Valid values are `dimension` and `measure`.

Attribute	Required	Description
`enums`	No	Specifies a list of values for an enumerated variable type for a `StringType` or `NumberType` variable type.
`enumsLabels`	No	Specifies a list of labels for the enumerated values of a `StringType` or `NumberType` variable only.
`itemsType`	No	Specifies the items type for an `ArrayType` variable. The items type must be a valid variable type.
`max`	No	Specifies the maximum value for a `NumberType` variable only.
`min`	No	Specifies the minimum value for a `NumberType` variable only.
`type`	Yes	The type of the variable. Valid types are: • `ArrayType` (Enumerated) • `BooleanType` (Enumerated) • `CalculatedInsightFieldType` (Object) • `CalculatedInsightType` (Object) • `ConnectorType` (String) • `DataLakeFieldObjectType` (Object) • `DataLakeObjectType` (Object) • `DataModelFieldObjectType` (Object) • `DataModelObjectType` (Object) • `DatasetAnyFieldType` (Object) • `DatasetDateType` (Object) • `DatasetDimensionType` (Object) • `DatasetMeasureType` (Object) • `DatasetType` (Object) • `DateTimeType` • `NumberType` (Enumerated) • `SObjectFieldType` (Object) • `SObjectType` (Object)

Attribute	Required	Description
		• `StringType` (Enumerated)

SEE ALSO:

Edit variables.json

org-readiness.json Attributes

The `org-readiness.json` file attributes are:

Org Readiness Attributes

Attribute	Required	Description
`values`	No	The variable values to use for readiness checks. Values provided by the API call override these values.
`templateRequirements`	Yes	An array that defines the requirements for the template to validate successfully.
`definition`	Yes	The requirement definitions used by the template requirements.

`values` Attributes

Each entry must match a variable from the `variables.json` file, with a defined value to use for the readiness validation. `values` can contain 0 or more variables.

This example illustrates how to define values for different variable types.

```
"values": {
  "booleanVar" : true,
  "stringVar" : "My string",
  "numericVar" : 10,
  "sobjectFieldVar" : {
    "sobjectName": "Opportunity",
    "fieldName": "StageName"
  }
}
```

`templateRequirements` Attributes

An array of requirements to use for validating the template. A requirement uses a `definition` to evaluate at runtime. The optional `type`, `image`, `tags`, and `message` attributes define how the requirement results are displayed to end users.

Attribute	Required	Description
`expression`	Yes	The expression to evaluate for validation.
`type`	No	The readiness type of the requirement, which must match the `definition` type. In this context, it's used by the UI for badging purposes.
`image`	No	An image that best represents the template requirement. The image can be a static resource or a standard CRM Analytics image.
`successMessage`	No	The message to display when the expression evaluates to `true`. EL expressions can be used to create dynamic messages.
`failMessage`	No	The message to display when the expression evaluates to `false`. EL expressions can be used to create dynamic messages.
`tags`	No	A collection of strings to use as categories to describe the template requirement. The tags are user-defined and are used to group multiple template requirements together into common themes.

This example illustrates how to define requirements of different complexity.

```
”templateRequirements” : [
  {
    "expression" : "${Readiness.numberOfAccounts > Variables.numericValue}",
    "type" : "SobjectRowCount",
    "image" : {
      "namespace" : "my_namespace",
      "name" : "accounts_required.jpg"
    },
    "successMessage" : "Your org contains enough accounts to proceed.",
    "failMessage" : "You only have ${Readiness.numberOfAccounts} accounts. You must have
${Variables.numericValue} or more accounts to proceed."
    "tags" : ["Accounts", "Cases"]
  },
  {
    "expression" : "${Readiness.numberOfCases > 5}",
    "failMessage" : "You must have 5 or more cases to proceed."
  }
]
```

`definition` Attributes

Each entry defines a readiness check. There must be at least 1 `definition` that is used in the `templateRequirements`. The evaluated result of each definition is stored in the template context as `Readiness['<definition_name>']`.

These examples illustrate definitions for different requirement types.

The `OrgPreferenceCheck` validates one or more org preferences.

```
"Example Org Readiness Check": {
  "type" : "OrgPreferenceCheck",
  "names" : [ "DashboardSavedViewEnabled" ]
}
```

```
EL Expression: ${Readiness['Example Org Readiness Check'].DashboardSavedViewEnabled}
```

The `SobjectRowCount` performs a SOQL query to retrieve a row count for the specified sObject and optional filters. Both the `sobject` and `filter` attributes can contain EL expressions that are resolved at runtime. This example validates that there are a specific number of accounts with an annual revenue greater than the minimum value specified at runtime. This check is performed before app creation and the results are available for `templateReadiness`.

```
"Count of Accounts Revenue" : {
  "type" : "SobjectRowCount",
  "sobject" : "Account",
  "filters" : [
    {
      "field" : "AnnualRevenue",
      "operator" : "GreaterThan",
      "value" : "${Variables.MinimumAnnualRevenue}"
    }
  ]
}
```

```
EL Expression: ${Readiness['Count of Accounts Revenue'] >= 500}
```

The `OrgDatasetRowCount` performs a SQL query on an existing dataset to determine row count based on the dataset id or name and optional filters. This check can query datasets outside of the scope of the current or future app, which is useful for templates that have template dependencies on other apps. This check is performed before app creation and the results are available for `templateReadiness`.

```
"Query Parent App Dataset" : {
  "type" : "OrgDatasetRowCount",
  "dataset" : "${Apps.parentTemplateApp.Datasets.quota.Id}",
  "filters" : [
    {
      "field" : "QuotaDollarValue",
      "operator" : "LessThan",
      "value" : "100000"
    }
  ]
}
```

```
EL Expression: ${Readiness['Query Parent App Dataset'] >= 500}
```

The `AppDatasetRowCount` performs a SQL query on an existing dataset to determine row count based on the dataset id or name and optional filters. This check queries datasets that are created by the template. It validates if a dataset dependent asset, such as a dashboard or Einstein Discovery story can or should be created. This check is performed during app construction, after the datasets are

created and recipes are run to populate the datasets. The results of this check can't be referenced by `templateRequirements`, but they can be referenced by template asset conditions in `template-info.json`.

```
"Query App Dataset" : {
  "type" : "AppDatasetRowCount",
  "dataset" : "${App.Datasets.quota.Id}",
  "filters" : [
    {
      "field" : "QuotaDollarValue",
      "operator" : "LessThan",
      "value" : "100000"
    }
  ]
}
```

```
EL Expression: ${Readiness['Query App Dataset'] >= 500}
```

The `ApexCallout` performs a check using a custom Apex method in the template's implementation of `WaveTemplateConfigurationModifier`. The method must be a non-static, global method with an arbitrary number of arguments. The method return value is also arbitrary or void and is place into the template context for reference in the `templateRequirements` or in rules processing.

```
"Global Apex Method Readiness Check" : {
  "type" : "ApexCallout",
  "method" : "checkReadiness",
  "arguments" : {
    "booleanArg" : true
  }
}
```

```
EL Expression: ${Readiness['Global Apex Method Readiness Check'] == true}
```

Example Apex method for the readiness check

```
global Boolean checkReadiness(Boolean booleanArg) {
  if (booleanArg) {
    return true;
  }
  return false;
}
```

The `DataCloudRowCount` performs a SQL query on an existing Data Cloud object, DLO, DMO, or Calculated Insight, to determine row count based on the object name or id and optional filters. This check can query Data Cloud objects outside of the scope of the current or future app. This check is for templates that have dependencies on the existence and status of specific Data Cloud objects. This check is performed before app creation and the results are available for `templateReadiness`.

```
"Row Count of DMO" : {
  "type" : "DataCloudRowCount",
  "object" : "ssot__Account__dlm",
  "filters" : [
    {
      "field" : "Id__c",
      "operator" : "In",
      "value" : ["001RO00004KkxxLxxx", "001RO00004KkxxMxxx"]
    }
```

```
  ]
}
```

```
EL Expression: ${Readiness['Row Count of DMO'] >= 0}
```

SEE ALSO:

Edit org-readiness.json

folder.json Attributes

The `folder.json` file attributes are:

Attribute	Required	Description
`name`	Yes for auto-installed templates, No for templates that use the configuration wizard	Folder developer name. For apps created from auto-installed templates, the folder developer name is required for the creation of app.
`label`	No	Folder label.
`description`	No	Folder description.
`featuredAssets`	Yes	Container for assets; this can be empty if no `featuredAssets` are present.
`featuredAssets default`	No	Container for assets.
`featuredAssets default assets`	No	Container for array of assets.
`featuredAssets default assets id` element	No	Tokenized ID of dashboard asset.
`featuredAssets default assets name`	No	Asset name.
`featuredAssets default assets namespace`	No	Asset namespace.
`featuredAssets default assets type`	No	Asset type, such as `dashboards`, `lenses`, or `eltDatflows`. Can be `null`.
`shares`	No	Sharing rules for the folder.
`shares sharedWithID`	Yes, if `shares` is used.	User ID or Group ID to share this folder with.
`shares accessType`	Yes, if `shares` is used.	Access type for the sharing rule. Valid values are `EDIT`, `MANAGE`, and `VIEW`.
`shares shareType`	Yes, if `shares` is used.	The sharing type for the sharing rule. Valid values are `Group`, `Organization`,

Attribute	Required	Description
		`Role`, `RoleAndSubordinates`, `RoleAndSubordinatesInternal`, `User`, `PortalRole`, `PortalRoleAndSubordinates`, `AllCspUsers`, `AllPrmUsers`, `PartnerUsers`, and `CustomerPortalUser`.

SEE ALSO:

Edit folder.json

auto-install.json Attributes

The `auto-install.json` file attributes are:

Attribute	Required	Description
`hooks`	No	Container for hook types.
`hooks type`	Yes	The type of auto-install hook. Valid values are: `PackageInstall`.
`hooks requestName`	No	The name for the auto-install request.
`configuration appConfiguration`	No	The configuration to use for creating the app.
`configuration appConfiguration autoShareWithLicensedUsers`	No	Indicates whether the app is automatically shared with users that have the AnalyticsViewOnlyEmbeddedApp license (`true`) or not (`false`). The default value is `false`.
`configuration appConfiguration autoShareWithOriginator`	No	Indicates whether the app is automatically shared with the user who initiated the auto-install request (`true`) or not (`false`). The default value is `true`.
`configuration appConfiguration deleteAppOnConstructionFailure`	No	Indicates whether the app is deleted when the auto-install request fails due to a construction error (`true`) or not (`false`). The default value is `false`.
`configuration appConfiguration failOnDuplicateName`	No	Indicates whether the app creation fails when an existing app or asset exists with the same developer name (`true`) or not (`false`). The default value is `false`.

Attribute	Required	Description
`configuration appConfiguration values`	No	A map of variable values to use when creating the app. The variable names must correspond to variables defined in the `variables.json` file

SEE ALSO:

Edit auto-install.json

rules.json Example

Refer to this example of the `rules.json` file.

Example:

```
{
   "constants":[

           {
          "name":"ChartType",

 "value":"pie"
      }
   ],
    "rules":[  w

           {
          "name":"rule1",
          "appliesTo":[
             "type":"dashboard",
             "name":"dashboardOne",

          ],
          "actions":[
             {
                "action":"put",
                "description":"Put a section back in, but with different attributes",
                "path":"$.state.widgets.chart_1",
                "key":"pos",
                "value":{
                   "w":"501",
                   "y":"41",
                   "h":"121",
                   "x":"51"
                }
             },
             {
                "action":"delete",
                "description":"Delete a section",
                "path":"$.state.widgets.chart_2"
```

```
            }
        ]
```

The corresponding `dashboardOne.json` file that uses the above rules files looks like:

```
{
    "_type": "dashboardTemplate",
    "name": "Dashboard From Template With Rules",
    "edgemarts": {
        "${Variables.Dataset1.datasetAlias}": {
            "_type": "edgemart",
            "_uid": "${Variables.Dataset1.datasetId}"
        },
        "${Variables.Dataset2.datasetAlias}": {
            "_type": "edgemart",
            "_uid": "${Variables.Dataset2.datasetId}"
        }
    },
    "folder": {
        "_type": "folder",
        "_uid": "${App.Folder.Id}"
    },
    "tags": [

    ],
    "state": {
        "widgets": {
            "chart_1": {
                "params": {
                    "chartType": "${Constants.ChartType}",
                    "minColumnWidth": 30,
                    "maxColumnWidth": 200,
                    "legend": false,
                    "selectMode": "single",
                    "sqrt": false,
                    "legendHideHeader": false,
                    "legendWidth": 145,
                    "step": "step_1"
                },
                "type": "ChartWidget",
                "pos": {
                    "w": "500",
                    "y": 60,
                    "h": "120",
                    "x": 40
                }
            },
            "chart_2": {
                "params": {
                    "chartType": "${Constants.ChartType}",
                    "minColumnWidth": 30,
                    "maxColumnWidth": 200,
                    "legend": false,
                    "selectMode": "single",
                    "sqrt": false,
```

```
                    "legendHideHeader": false,
                    "legendWidth": 145,
                    "step": "step_2"
                },
                "type": "ChartWidget",data
                "pos": {
                    "w": "500",
                    "y": 190,
                    "h": "120",
                    "x": 40
                }
            }
        },
        "steps": {
            "step_1": {
                "isFacet": true,
                "start": null,
                "query": {
                    "measures": [
                        [
                            "count",
                            "*"
                        ]
                    ]
                },
                "selectMode": "single",
                "useGlobal": true,
                "em": "${Variables.Dataset1.datasetId}",
                "type": "aggregate",
                "isGlobal": false
            },
            "step_2": {
                "isFacet": true,
                "start": null,
                "query": {
                    "measures": [
                        [
                            "count",
                            "*"
                        ]
                    ]
                },
                "selectMode": "single",
                "useGlobal": true,
                "em": "${Variables.Dataset2.datasetId}",
                "type": "aggregate",
                "isGlobal": false
            }
        },
        "type": "hbar"
    }
}
```

When the rules are applied to the dashboard, the `chart_1` widget will have a chartType of "pie" and the "pos" will be updated to w=501, y=41, h=121, and x=51. The `chart_2` widget will be removed.

rules.json Attributes

The `rules.json` file attributes are:

Attribute	Required	Description
`constants`	No	Container attribute only - array of constants to be defined, can be empty if no constants are needed
`constants --> name`	No	Unique name for a constant, used to reference the constant in other places
`constants --> value`	No	Value of the constant
`rules`	No	Container attribute only - array of rules to be defined, can be empty if no rules are needed
`label`	No	
`rules --> name`	Yes	Unique name for a rule
`rules --> condition`	No	If condition is met, then the rule will run, if the condition is not met, the rule will not run.
`rules --> appliesTo`	Yes	Container attribute only - array of assets the rule should be be applied to (dashboards, dataflow, recipes, schemas, XMDs)
`rules --> appliesTo --> name`	No. Best practice is to specify `name`. If null, defaults to `*`, and rule applies to all template assets.	Name of asset (can be * in case of dashboards for all dashboards); must match the "name" defined in the template-info for the asset
`rules --> appliesTo --> type`	No. Best practice is to specify `type`. If null, defaults to `*`, and rule applies to all asset types*	Type of asset (dashboard, workflow, lens, schema, xmd)
`rules --> actions`	Yes	Container attribute only - array of actions the rule should invoke (one or more action is necessary)
`rules --> actions --> action`	Yes	Type of action - set, add, delete, put, replace
`rules --> actions --> description`	No	Description of the rule action - for maintainability
`rules --> actions --> path`	Yes	JSONPath to take the action on

Attribute	Required	Description
`rules --> actions --> value`	No	Value needed for set/add/put, defines the value to put in the JSONPath
`rules --> actions --> index`	No	Index needed for add action only
`rules --> actions --> key`	No	Key needed for put action only

SEE ALSO:

Edit Rules Files

Rules Testing with jsonxform/transformation endpoint

Test the results of a rule before deployment by calling the `jsonxform/transformation` endpoint.

Process for Using `jsonxform/transformation`

Use cURL or Postman to execute a `Post` call on the `/services/data/v51.0/jsonxform/transformation` REST endpoint. See the examples for starting points on what the POST request body must contain.

If the rule works, the POST response contains the transformed JSON results. If the rule doesn't work, the POST response contains any errors with guidance to fix the rule.

The following examples show how you can take advantage of the built-in tokens we provide.

Example: Post Transform

A "put" rule that adds a `name/value` pair to the JSON.

```
{
 "document": {
  "user": {
   "firstName": "${User.FirstName}",
   "lastName": "${User.LastName}",
   "userName": "${User.UserName}",
   "id": "${User.Id}",
   "hello": "${Variables.hello}"
  },
  "company": {
   "id": "${Org.Id}",
   "name": "${Org.Name}",
   "namespace": "${Org.Namespace}"
  }

 },
 "values": {

  "Variables": {
   "hello": "world"
  }
```

```
 },
 "definition": {
  "rules": [{
   "name": "Example",
   "actions": [{
    "action": "put",
    "description": "add hobby to user",
    "key": "hobby",
    "path": "$.user",
    "value": "mountain biking"
   }]
  }]
 }
}
```

Example: Constant Array

Tests an array function (`contains`) to determine the right value to add to the final JSON.

```
{
 "document": {
  "user": {
   "firstName": "${User.FirstName}",
   "lastName": "${User.LastName}",
   "userName": "${User.UserName}",
   "id": "${User.Id}",
   "hello": "${Variables.hello}"
  },
  "company": {
   "id": "${Org.Id}",
   "name": "${Org.Name}",
   "namespace": "${Org.Namespace}"
  }

 },
 "values": {
  "Variables": {
   "hello": "world",
   "choices": ["Products", "Cases"]
  }
 },
 "definition": {
  "constants": [{
   "name": "ConstantTest",
   "value": "${array:contains(Variables.choices, 'Products') ? 'Products' : 'No Products'}"

  }],
  "rules": [{
   "name": "Example",
   "actions": [{
    "action": "put",
    "description": "add hobby to user with constant",
    "key": "hobby",
    "path": "$.user",
```

```
    "value": "${Constants.ConstantTest}"
   }]
  }]
 }
}
```

Example: Macro Transform

Tests a macro with a rule.

```
{
 "document": {
  "user": {
   "firstName": "${User.FirstName}",
   "lastName": "${User.LastName}",
   "userName": "${User.UserName}",
   "id": "${User.Id}",
   "favorite": null
  }
 },
 "values": {},
 "definition": {
  "macros": [{
   "namespace": "jsonTest",
   "definitions": [{
    "name": "setNodeByPath",
    "parameters": [
     "jsonPath",
     "value"
    ],
    "actions": [{
     "action": "set",
     "path": "${p.jsonPath}",
     "value": "${p.value}"
    }]
   }]
  }],
  "rules": [{
   "name": "Example",
   "actions": [{
     "action": "put",
     "description": "add hobby to user",
     "key": "hobby",
     "path": "$.user",
     "value": "mountain biking"
    },
    {
     "action": "eval",
     "description": "Set the value of 'favorite' with the value from 'hobby'",
     "path": "$.user.hobby",
     "value": "${jsonTest:setNodeByPath('$.user.favorite', Rules.CurrentNode)}"
    }

   ]
```

```
  }]
 }
}
```

For reference on the `jsonxform/transformation` REST endpoint, see *JsonXform Transformation Resource* in the Analytics REST API Developer Guide.

VisualForce Events for Customizing the Wizard UI

This reference section explains the details of using VisualForce for customizing the Wizard user interface.

Table 1: List of Events to Which You Can Subscribe or Publish

Event Name	Subscribe/Publish	Description	Response
`ready`	publish	Fired from client indicating they are ready to receive events.	Response contains all the metadata: page, variableDefinitions, initialValues needed to render the questions and possible answers for the page.
`update`	publish	Fired from client when ready to update one or more variables.	Variables with errors messages.
`visibility`	publish	Fired from client to check 'visibility' of one or more questions.	Variables and their visibility enum.
`values`	publish	Fired from client to request the most up-to-date values of each question.	The values of all variables in the entire wizard.
`next`	subscribe	Fired from parent when user clicks "continue" button. May or may not advance the page. If there are errors in the variable values, the page will not advance.	A list of variables that are required with no values.
`resize`	publish	Fired from client to change the size of the VisualForce page.	None.
`options`	publish	Fired from client to get a list of 'options' for filling in drop-down list boxes.	List will be filtered for "queryable," "excludes," "exclude selected." "Tags" will be applied to indicate which option values are "default," "recommended," or "previous" values. You can use this for sObjects, sObjectFields, string enums, and number enums.

Event Name	Subscribe/Publish	Description	Response
`computed`	publish	Fired from client to set the computed (suggested) values for a variable.	Variables with success or error messages.
`close`	publish	Fired from client to close the wizard. No actions are taken as part of the close.	None.
`buttons`	publish	Change the text of the Back and Next button. Can also enable or disable the Back or Next button.	
`create`	publish	Fired from the client to create an app before the final screen. Calls `POST /wave/folder` and returns results.	The status of the POST call. Success if app creation is kicked off successfully (not final app status), otherwise Fail.
`dataset`	COMING SOON	Dataset	*Coming soon; not currently supported in this release. We intend to implement this as an 'options' event.*
`dataset fields`	COMING SOON	Dimensions, Measures tied to parent datasets	*Coming soon; not currently supported in this release. We intend to implement this as an 'options' event.*

Event: wizard.ready

- **Subscribe/Publish:** publish
- **Input/Multiple:** optional
  - **Size:** small, medium, large
  - **Banner:** visible, title, progress

    Note: for VisualForce pages, the banner is hidden by default.
- **Response:** Yes
- **Normalized Input:** N/A
- **Input Examples:**
  - `{}`
  - `{size : "large" }`
  - `{banner : visible : true, title : "My Custom title", progress : "Almost done"}}`
  - `{size : "small", banner : {visible : false}}`
- **Output Example:**

```
{
  "page": {
```

```
  "condition": null,
  "helpUrl": null,
  "title": "Page 1",
  "variables": [
    {
      "name": "stringExample",
      "visibility": "Visible"
    },
    {
      "name": "visibilityControl"
      "visibility": "Visible"
    },
    {
      "name": "visibilityTest",
      "visibility": "Hidden"
    }
  ]
},
"variableDefinitions": {
  "stringExample": {
    "label": "What is the value of the string?",
    "description": "The String.",
    "defaultValue": null,
    "variableType": {
      "type": "StringType",
      "enums": [
        "foo",
        "bar",
        "baz",
        "testStr"
      ]
    }
  },
  "stringExample2": {
    "label": "What is the value of this string?",
    "description": "The String2.",
    "defaultValue": null,
    "variableType": {
      "type": "StringType"
    }
  },
  "visibilityControl": {
    "label": "Do you want to answer more questions?",
    "description": "Check to test visibility",
    "variableType": {
      "type": "BooleanType"
    },
    "defaultValue": false,
    "required": true
  },
  "visibilityTest": {
    "label": "I should only be visible when visibilityControl is checked",
    "variableType": {
      "type": "StringType",
```

```
        "enums": [
          "aaa",
          "bbb",
          "ccc"
        ]
      }
    },
    "visibilityTest2": {
      "label": "I should only be enabled when visibilityControl is checked",
      "variableType": {
        "type": "StringType"
      }
    },
    "datasetExample": {
      "label": "Pick a dataset",
      "description": "Interesting dataset",
      "variableType": {
        "type": "DatasetType"
      }
    },
    "dimensionExample": {
      "label": "Pick a dimension",
      "description": "Dim testing",
      "defaultValue": {
        "datasetId": "{{Variables.datasetExample.datasetId}}",
        "fieldName": "test"
      },
      "variableType": {
        "type": "DatasetDimensionType"
      },
      "required": false
    }
  },
  "initialValues": {
    "stringExample": null,
    "stringExample2": null,
    "visibilityControl": false,
    "dimensionExample": {
      "datasetId": "{{Variables.datasetExample.datasetId}}",
      "fieldName": "test"
    }
  }
}
```

Event: wizard.update

- **Subscribe/Publish:** publish
- **Input/Multiple:** yes
- **Response:** yes
- **Normalized Input:** yes
- **Input Examples:**

  - `{name : "myVariable", value : "some value" }`
  - `[{name : "myVariable", value : "some value" }]`
  - `[{name : "myVariable", value : "some value" }, {name : "myNumber", value : 123}]`
  - `{myVariable : "some value", myNumber: 123, myBoolean : true}`
  - `{name : "mySObject", value : {sobjectName : "Account"}}`
- **Output Example:** always an array

```
[
    {name : "myVariable": value : "some value", valid : true },
    {name : "myNumber": value : null, valid : false, errorMessage : "Not a valid value"}

    {name : "mySObject": value : {sobjectName : "Account"}, valid : true}
]
```

Event: wizard.visibility

- **Subscribe/Publish:** publish
- **Input/Multiple:** yes
- **Response:** yes
- **Normalized Input:** yes
- **Input Examples:**
  - `["myVariable", "myNumber"]`
  - `[{name : "myVariable"}]`
  - `{name : "myVariable"}`
  - `"myVariable"`
- **Output Example:** always an array

```
[

  {name : "myVariable": visibility : "Hidden"},

  {name : "myNumber": visibility : "Visible"},

  {name : "myOtherVariable": visibility : "Disabled"}

]
```

Event: wizard.values

- **Subscribe/Publish:** publish
- **Input/Multiple:** n/a
- **Response:** yes
- **Normalized Input:** n/a
- **Input Examples:**

  - `{}`
- **Output Example:**

```
{
    values : {
        myVariable: "foo",
        myNumber: 123,
        myBoolean: true,
        myDimension: {fieldName : "days"}
    }
}
```

Event: wizard.next

- **Subscribe/Publish:** subscribe
- **Input/Multiple:** n/a
- **Response:** yes
- **Normalized Input:** n/a
- **Output Example:** Page will not advance if required fields are not met.
  - ```
    {
            "myVariable": {
                  errorMessage: "Not a valid value",
                  isRequired : true,
                  variableName : "myVariable"
            }
    }
    ```
  - `{}`

Event: wizard.resize

- **Subscribe/Publish:** publish
- **Input/Multiple:** n/a
- **Response:** no
- **Normalized Input:** n/a
- **Input Examples:**
  - `{size : "small"}`
  - `{size : "medium"}`
  - `{size : "large"}`

Event: wizard.options

- **Subscribe/Publish:** publish
- **Input/Multiple:** no
    ```

- **Response:** yes
- **Normalized Input:** n/a
- **Input Examples:**
  - `{name : "mySObjectVariableName"}`
  - `{name : "mySObjectFieldVariableName"}`
  - `{name : "myStringVariableName"}`
  - `{name : "myNumbertVariableName"}`
- **Output Example:**

```
{
 "options": [
   {
     "label": "Account",
     "value": "Account",
     "tags": [
       {
         "tag": "Default",
         "toolTip": "Default answer"
       }
     ]
   },
   {
     "label": "Oppty",
     "value": "Opportunities"
   }
 ]
}
```

Event: wizard.buttons

- **Subscribe/Publish:** publish
- **Input/Multiple:** n/a
- **Response:** no
- **Normalized Input:** n/a
- **Input Examples:**

- ```
  {buttons : {
      next : {disabled : true, text : “Fix error before continue”},
      back : {disabled : false, text : “Go Back”}
    }
  }
  ```
- ```
  {  buttons : {
      back : {disabled : true}
    }
  }
  ```
- ```
  {  buttons : {
      next: {“text” : “Proceed at Own Risk”}
    }
  }
  ```

- **Output Example:**

  None. Buttons should update accordingly.

Event: wizard.create

- **Subscribe/Publish:** publish
- **Input/Multiple:** no
- **Response:** yes
- **Normalized Input:** yes
- **Input Examples:**
  - ```
    {folder : {label : "vfTest", description : "my desc"}}
    ```