



Omnistudio



© Copyright 2000–2026 salesforce.com, inc. All rights reserved. Salesforce is a registered trademark of salesforce.com, inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

Omnistudio	1
Get Started with Omnistudio	2
Learn about Omnistudio	4
Understand Omniscripts	5
Understand Flexcards	7
When to Use Flexcards and Omniscripts	8
When to Use Omnistudio Data Mappers and Integration Procedures	9
Integration Procedure Features Not in Omniscripts	9
Use the Applications Together	10
Plan and Prepare for Your Omnistudio Implementation	12
Omnistudio Naming Conventions	14
Omnistudio sObject Descriptions	16
Newport Design System	17
Callable Implementations	20
Add Apex Class Permissions Checkers	21
View the Namespace and Version of Managed Packages	21
Standard Omnistudio Content and Runtime	23
Deployment of Omnistudio Components Between Orgs	26
Tool Considerations for Deploying Omnistudio Components	27
Omnistudio Standard Designer	28
Feature Support Changes with Standard Designers	30
Enable the Omnistudio Standard Designer	30
Best Practices for Using the Omnistudio Standard Designer	31
Hide the Omnistudio App While Using the Standard Designer	32
Access Omnistudio Designer Components from the List View	33
Omnistudio Frequently Asked Questions (FAQs)	34
Omnistudio Limits and Considerations	39
Recommended Limits	40
Security Checks for Omnistudio	42
Omniscripts	43
Working with Omniscripts	45
Create an Omniscript	48
Create Multi-Language Omniscripts	49
Build an Omniscript with Elements	66
Configure Omniscript Settings	92
Integrate DocuSign with Omniscripts	111
Customize Omniscript Behaviors, Style, and Elements	119
Preview and Test an Omniscript	159

Activate and Launch Omniscripts	161
Omniscript Element Reference	166
Omnistudio Flexcards	285
Working with Flexcards	290
Open a Flexcard in a Canvas Instead of the Record Page Detail	290
Create a Flexcard	291
Configure Flexcard Settings	292
Add Elements to a Flexcard	309
Set Up Actions on a Flexcard	325
Display Data on Flexcards Based on Conditions	344
Style a Flexcard	348
Preview and Debug a Flexcard	353
Activate and Publish a Flexcard	356
Export and Import Flexcards	360
Flexcards Context Variables	372
Flexcards Omni Interaction Access Configuration	376
Omnistudio Data Mappers.....	387
Working with Omnistudio Data Mappers.....	390
Build an Omnistudio Data Mapper.....	393
Invoke Omnistudio Data Mappers	434
Security for Omnistudio Data Mappers and Integration Procedures	456
Preview and Test Data Mappers.....	462
Examples of Omnistudio Data Mappers	463
Omnistudio Integration Procedures	482
Work with Integration Procedures.....	485
Build an Integration Procedure	487
Integration Procedures on Agentforce	562
Invoke an Integration Procedure	565
Long-Running Integration Procedures	592
Security for Omnistudio Data Mappers and Integration Procedures	608
Test Procedures: Integration Procedures for Unit Testing	614
Integration Procedure Actions	620
Integration Procedure Blocks	649
Omnistudio Design Assistant	653
Monitor and Manage Your Flexcards and Omniscripts	655
Use Custom Numbering with Omni Global Auto Number	656
Set Up Omni Global Auto Numbers	657
Automated Testing with UTAM for Omnistudio	659
UTAM Testing Workflow	661
How to Write End-to-End Tests with UTAM for Flexcards	661
How to Write End-to-End Tests with UTAM for Omniscripts	664
Agentforce for Omnistudio (Pilot).....	671
Agentforce Topics for Omnistudio (Pilot).....	672
Omnistudio Formulas and Functions	676
Supported Data Mapper and Integration Procedure Data Types.....	677

Supported Data Mapper and Integration Procedure Operators	685
Supported Data Mapper and Integration Procedure Functions.....	689
Sample Apex Code for Custom Functions.....	763
Using OmniAnalytics to Track User Engagement in Omniscripts and Flexcards ...	
768	
Enable OmniAnalytics and Store Tracking Data	771
Configure Internal OmniAnalytics.....	772
Configure OmniAnalytics with Google Analytics.....	779
Create Google Analytics Trusted URLs	780
Adding Ecommerce Data to the Example Omniscript for OmniAnalytics	780
Setting Up Third-Party Tracking and Event Types	785
Omnistudio Lightning Web Components	792
Set Up Lightning Web Components	794
Extend Omnistudio Lightning Web Components.....	794
Deploy Lightning Web Components.....	796
Base Omnistudio LWC ReadMe Reference	796
Omnistudio on Experience Cloud.....	797
Omnistudio Usage-Based Entitlements	799

Omnistudio

Omnistudio provides a suite of services, components, and data model objects that combine to create Industry Cloud applications. Create guided interactions using data from your Salesforce org and external sources. Omnistudio is available to select Industry Cloud customers.

This document applies to Omnistudio on standard runtime with the standard designer or managed package designer. Any differences in behavior or options are specified in the respective sections of the designers.

-  **Note** Omnistudio uses the standard runtime and objects. If you're using Omnistudio with custom objects and the managed package runtime, see [Omnistudio for Managed Packages](#).

What Can You Do with Omnistudio?

Contain the user-interaction logic with [Omniscripts](#).

Transfer and transform data between Salesforce and the Omniscripts, Flexcards, and Integration Procedures tools with [Omnistudio Data Mappers](#).

Bundle server-side data integration operations for efficiency and reuse with [Integration Procedures](#).

Display data and launch actions with [Flexcards](#).

Omnistudio

Omnistudio provides a comprehensive, low-code suite of services and components, designed to build engaging, industry-specific digital experiences. It accelerates the path from development to production, enabling you to create guided customer interactions by using data from Salesforce and external sources.

Use the Omnistudio designers to build, manage, preview, and optimize your apps. As a declarative environment, Omnistudio helps you focus on business logic rather than programming languages. You can create dynamic user flows with Flexcards and Omniscripts, and seamlessly integrate different data sources with Data Mappers and Integration Procedures. When combined with products like Business Rules Engine, Omnistudio enforces complex logic within your user flows. Built on a Lightning Web Component (LWC) runtime, your solutions are optimized for high performance and easy deployment.



Note Starting in Summer '25, if you're a new Omnistudio user, don't install the Omnistudio package.

 Get Oriented Get Started with Omnistudio Omnistudio Permission Sets Omnistudio SDLS 2 Theme	 Dive In: Learn About Recommended Core Features Accelerate Component Creation Using the Omnistudio Standard Designer Security Checks for Omnistudio Security for Omnistudio Data Mappers and Integration Procedures Automate Omnistudio UI Testing with UTAM Agentforce for Omnistudio	 Go Deeper: Learn About Features for Specific Business Needs Omnistudio Lightning Web Components Customize Omniscript Behaviors, Style, and Elements Configure Flexcard Settings Enhance Experience Cloud Sites with Omnistudio Create Custom Numbers with Omni Global Auto Number
		

Extend Further: Learn About Additional Capabilities & Add-Ons Get Real-time Design Feedback Using Omnistudio Design Assistant Omnistudio Formulas and Functions Salesforce Document Generation Business Rules Engine	Get Ready for Your Implementation Plan and Prepare for Your Omnistudio Implementation Integrate DocuSign with Omniscripts Deploy Omnistudio Components Between Orgs View Your Org's Omnistudio Usage-Based Entitlements	Know Your Resources Trailhead: Get Started with Omnistudio on the Salesforce Platform Developer Resources for Omnistudio
---	--	---

Omnistudio

Build Service Consoles and Industry Cloud Apps with the basic building blocks of Omnistudio.

Understand Omniscripts

An Omniscript guides users through complex processes with fast, personalized, and consistent responses.

Understand Flexcards

Flexcards provide tools for building customer-centric, industry-specific UI components and applications on the Salesforce platform. Flexcards are rich in information and actions relevant to the customer's context. Create your Flexcards in a declarative design tool and add them to your Lightning or Experience Cloud pages.

When to Use Flexcards and Omniscripts

Omniscripts and Flexcards have many basic similarities: Both are user interfaces, can pull data from multiple sources, and can be interactive. To decide when to use Flexcards or Omniscripts, consider if the purpose is to only show data or if users are likely to require complex interactive UIs.

When to Use Omnistudio Data Mappers and Integration Procedures

Omnistudio Data Mappers read or write Salesforce SObject data or perform single-step data structure transformations. Omnistudio Integration Procedures can interact with many types of data, including REST APIs and Apex classes, and process it in multiple steps. For some use cases, a single Data Mapper is sufficient. Integration Procedures usually call one or more Data Mappers and are more flexible and powerful. Use these guidelines to help you determine which to build.

Integration Procedure Features Not in Omniscripts

Integration Procedures and Omniscripts have differences and common features. Some important Integration Procedure features aren't present in Omniscripts.

Use the Applications Together

Use the Omnistudio applications together. Create guided interactions with Omniscripts; get and post data with Omnistudio Data Mappers and Integration Procedures; and display data and launch actions with Flexcards.

Understand Omniscripts

An Omniscript guides users through complex processes with fast, personalized, and consistent responses.

For example, create an Omniscript to guide:

- A customer service agent to add a customer
- An insurance agent to update a policy
- An end user to complete a self-service interaction such as troubleshooting

You can create a guided interaction to match the flow of your process. Omniscript is a declarative scripting tool, meaning you create it with clicks, not code. To create the structure of an Omniscript, you drag different types of elements to:

- Add actions such as extract data or send an email
- Group items together by creating a step or displaying a list of items the customer can select from
- Create a function such as a formula
- Add input fields and lookups for the user to enter data
- Refine the display by using a headline or text block
- Create branches that dynamically adjust the controls and enable or disable steps depending on choices the user makes in the guided process
- Configure calculations and messages that provide immediate feedback and error checking to the user

Templates control both the style and appearance of Omniscripts. You can customize whether your guided interaction has a horizontal or vertical mode, branding, and any other aspects you wish to change.



Templates

You can launch an Omniscript from anywhere, including:

- An action button such as the one shown here on the account page
- An action link on a card

		Apple iPhone X With iPhone X, the device is the display. An all-new 5.8-inch Super Retina screen fits the hand and dances to the eyes.	\$999.99	\$0.00	One Time	Monthly
		Apple iPhone 8 A beautiful mind. iPhone 8 introduces an all-new glass design. The	\$699.99	\$0.00		

- Note** The generated Omniscript is a Lightning Web Component. Based on the business logic, some

users modify the generated Omniscript.

Understand Flexcards

Flexcards provide tools for building customer-centric, industry-specific UI components and applications on the Salesforce platform. Flexcards are rich in information and actions relevant to the customer's context. Create your Flexcards in a declarative design tool and add them to your Lightning or Experience Cloud pages.

Customer Context

Each customer is linked to multiple aspects of your company's products and services. Customers have accounts, preferred methods to receive bills, preferred means of contact, and a history of the products they have purchased as well as their interactions with the company. When agents must use different systems to gather contextual information about the customer, it affects customer service. Use Flexcards to streamline customer engagement.

Designer

The Flexcard designer is a declarative tool to create UI components. Build dynamic customer-centric UIs without code from a drag interface with [WYSIWYG](#) editing. See [Omnistudio Flexcards](#).

Flexcards

A Flexcard is a block that contains a combination of pertinent information and links to processes within a specific context. For example, an account card can include unique account information, such as:

- Status
- Priority or service level agreement
- Creation date

Actions on an account card might include:

- Closing a case
- Opening a new case
- Creating a new task

In the Flexcard Designer, you can create an action that launches or updates an Omniscript, navigates to a web page or application, displays a flyout, fires an event, update field values, and more. See [Set Up Actions on a Flexcard](#).

 **Note** The generated Flexcard is a Lightning Web Component. Based on the business logic, some users modify the generated Flexcard.

Design and Layout

Style individual elements or add custom CSS directly on an element within the Flexcard Designer. You can also make elements responsive. See [Style a Flexcard](#).

Data Sources

Select from multiple data source options to retrieve data to display on your Flexcards. For a list of data sources available in the Flexcard Designer, see [Set Up a Data Source on a Flexcard](#).

When to Use Flexcards and Omniscripts

Omniscripts and Flexcards have many basic similarities: Both are user interfaces, can pull data from multiple sources, and can be interactive. To decide when to use Flexcards or Omniscripts, consider if the purpose is to only show data or if users are likely to require complex interactive UIs.

Use Flexcards to show data such as account information, contact details, and opportunity details, without requiring multi-step actions from users. Flexcards can also launch Omniscripts, new windows, or let users perform simple actions such as updating an address. Flexcards can use conditional logic to show or hide elements based on certain conditions.

Use Omniscripts to show data and collect information from users in a step-by-step process. For instance, collect a user's medical history or vehicle details by using guided, interactive flows and response formats, such as radio buttons or text. Omniscripts can use Flexcards for showing visualizations of data. Omniscripts can use complex conditional logic for showing one set of actions over another.

Use Case for Flexcards

Let's say you need to design an app for an Experience Cloud page that your marketing agents use. When marketing managers log in to their dashboards, they want to see a list of all open opportunities assigned to them, starting with the oldest open opportunity.

In this situation, you can create a Flexcard to pull data from the Opportunity object and show it on a card in a table on your Experience Cloud page. Users can perform basic interactions on the card such as sorting the table to show the newest records first or filtering for a specific type of open opportunity such as Value Proposition.

Use Case for Omniscripts

Let's say you need to design an app for a Lightning page that collects an employee's medical history. The goal of the app is to propose a company-approved insurance plan best suited to the employee's needs.

For this use case, you can create one or more Omniscripts that take your employees through a step-by-

step questionnaire. The questionnaire asks them about their age, prior medical issues, prior insurance claims, and other factors. Use Omniscripts to design an application that has several screens that show dynamic content based on user inputs.

When to Use Omnistudio Data Mappers and Integration Procedures

Omnistudio Data Mappers read or write Salesforce SObject data or perform single-step data structure transformations. Omnistudio Integration Procedures can interact with many types of data, including REST APIs and Apex classes, and process it in multiple steps. For some use cases, a single Data Mapper is sufficient. Integration Procedures usually call one or more Data Mappers and are more flexible and powerful. Use these guidelines to help you determine which to build.

Use a single Data Mapper if:

- You read data from SObjects or write data to SObjects, but not both.
- The SObjects you read from or write to have a defined relationship. For example, Accounts and Contacts have a relationship because a Contact can have an AccountId value.
- You only have to work with JSON or XML data. No SObjects are involved.
- You can perform any needed filtering, calculation, or reformatting of data values using one or a series of formulas.
- You can make any needed changes to the data structure by mapping input JSON nodes to output JSON nodes.
- You don't read from or write to CSV files, Apex classes, REST APIs, or external objects.
- You don't send email, merge lists, or handle errors.

Use an Integration Procedure if:

- You must both read from and write to one or more SObjects, which means you must call at least two Data Mappers.
- The SObjects you read from or write to have no defined relationship.
- Transforming your data can't be done using formulas alone. For example, different conditions determine whether some filtering or calculations are performed at all.
- JSON node mappings aren't straightforward or require a series of steps.
- You read from or write to multiple data source types, such as SObjects, CSV files, external objects, Apex classes, or REST APIs.
- You perform actions such as sending emails, merging lists, or handling errors.

Integration Procedure Features Not in Omniscripts

Integration Procedures and Omniscripts have differences and common features. Some important Integration Procedure features aren't present in Omniscripts.

Unlike Omniscripts, Integration Procedures have no end-user UI, run on the server, and can run either synchronously or asynchronously.

Like Omniscripts, Integration Procedures can call Omnistudio Data Mappers, Integration Procedures, REST endpoints, and Apex methods. Also, like Omniscripts, Integration Procedures can send emails and DocuSign envelopes and manipulate data using functions and variables.

However, Integration Procedures have some useful features that Omniscripts lack:

- Caching of part or all of the Integration Procedure for performance
- Unit testing of entities the Integration Procedure calls
- Chainable and queueable chainable settings for avoiding governor limits
- Complex list processing and merging
- Invocation of Integration Procedures from Apex, REST, Flow, and Scheduled Jobs

Blocks specific to Integration Procedures are:

- Cache Block – Actions within are cached for performance.
- If-Else Conditional Block – Actions within have mutually exclusive conditions.
- Loop Block – Actions within run one time for each item in a list.
- Try-Catch Block – If any action within causes an error, the error is handled, and the Integration Procedure keeps running.

Actions specific to Integration Procedures are:

- Assert Action – Declares an expected result for unit testing.
- Batch Action – Runs a Scheduled Job.
- Chatter Action – Creates a Chatter post.
- List Action – Merges two lists based on matching key values.
- Response Action – Returns data to the entity that called the Integration Procedure.

Use the Applications Together

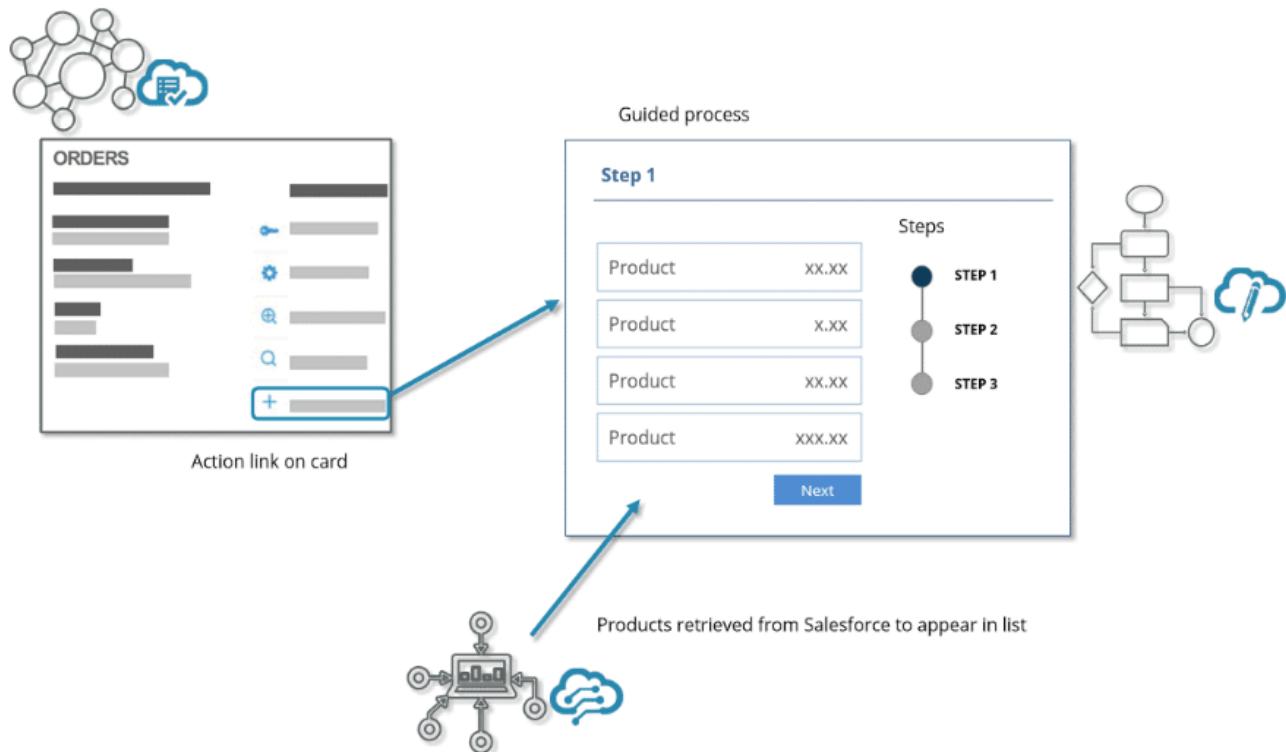
Use the Omnistudio applications together. Create guided interactions with Omniscripts; get and post data with Omnistudio Data Mappers and Integration Procedures; and display data and launch actions with Flexcards.

Here's an example of Flexcards, Omniscripts, and Data Mappers used together to:

- Display and provide a context for information and actions
- Complete a guided process
- Retrieve the correct data for the process

The agent views order-related information and actions contained in the card. When the agent clicks the new order link on the card, an Omniscript launches to begin a guided process for purchasing products. Omniscript uses a Data Mapper to get a list of products for display. Then the agent can choose one or

more products for purchase. When the purchasing process ends, the Omniscript returns the agent to the view that includes the Orders card.



Omnistudio

Plan and prepare for Omnistudio.

Now that you've learned what Omnistudio does, let's walk you through some key concepts before you begin your implementation.

[Omnistudio Naming Conventions](#)

When creating an Omnistudio component, such as an Omnistudio Data Mapper, Flexcard, Integration Procedure, or an Omniscript, follow the naming conventions on this page.

[Omnistudio sObject Descriptions](#)

The table on this page lists Omnistudio sObjects, their descriptions, and which objects you must grant read access to so that customers can view Omnistudio components.

[Newport Design System](#)

Use the Newport Design System to design, customize, and preview your Omniscripts directly within the designer canvas without switching between design and preview modes. The system offers Newport and Newport Storybook themes in addition to the existing Lightning theme. The Newport theme is available in the designer canvas and preview alongside the Lightning theme. The Newport Storybook theme is available only in the designer preview, alongside the Newport and Lightning themes. Previously, the Omniscript Designer canvas supported only the Lightning theme, while preview mode supported both Newport and Lightning.

[Callable Implementations](#)

Vlocity Apex classes and the Remote Actions of Omniscripts and Integration Procedures support the *Callable* interface.

[Add Apex Class Permissions Checkers](#)

Enable specific access to the classes used by remote action APIs for each user profile, permission set, or permission set group. Configure Apex class permissions checkers to ensure that users require explicit access to the Apex class that administers the remote action called from an Omniscript, Flexcard, Integration Procedure, or REST API.

[View the Namespace and Version of Managed Packages](#)

Omnistudio can be installed as a managed package. The package might be for a Cloud product (such as Insurance or Communications) or Omnistudio Foundation. You can check the namespace and version of your managed package, which you might need for planning upgrades or troubleshooting.

Standard Omnistudio Content and Runtime

With the Omnistudio permission set license, standard Omnistudio content runs in standard runtime. From the Lightning App and Experience Builders, add standard content to a Lightning or Experience Cloud site with standard Omniscript and Flexcard components. Standard Omnistudio components built by Salesforce Industries, such as Health Cloud and Financial Services Cloud, are available. You can also use standard components instead of custom-generated components to run custom content in the standard runtime.

Omnistudio Naming Conventions

When creating an Omnistudio component, such as an Omnistudio Data Mapper, Flexcard, Integration Procedure, or an Omniscript, follow the naming conventions on this page.

Product	sObject	Unique Name
Integration Procedures	Omni Process	<p>Type_SubType: The Type and SubType are joined by an underscore to create a unique identifier used to call the Integration Procedure. The Name Type and SubType can contain letters, numbers, and special characters, but no spaces.</p> <p>For example, <i>Type=Auto</i> and <i>SubType=CreateUpdateQuote</i> create a unique identifier <i>Auto_CreateUpdateQuote</i>.</p>
Data Mappers	Omni Data Transformation	<p>Interface Name: The Data Mapper Interface Name must be unique. The Interface Name can contain letters, numbers, and dashes, but no spaces.</p>
Omniscripts	Omni Process	<p>TypeSubtypeLanguage: Type, SubType, and Language combine to create a unique identifier that becomes the name of a compiled Omniscript's Lightning web component. The Type and SubType can contain letters and numbers, but no spaces or underscores and cannot exceed 104 characters.</p> <p>For example, an Omniscript where <i>Type=account</i>, <i>SubType</i></p>

Product	sObject	Unique Name
		=Create, and <i>Language=English</i> generates an LWC named <i>accountCreateEnglish</i> .
Flexcards	Omni Ui Card	Name + Author: The combination of the card Name and Author must be unique in your Org. Name and Author can only contain letters, numbers, and underscores, begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.

Reserved Words

These reserved words shouldn't be used in Flexcard names or element names:

- Action
- Data-element-label
- Data-action-key
- Data-element-label
- Data-action-element-class
- Flyout
- FlyoutType
- Tracking-obj
- Parent-Mergefields

Omnistudio Metadata API Support

If you enable Omnistudio Metadata API support through Setup under Omnistudio Settings, Omnistudio component names can only include letters and numbers and cannot contain spaces or special characters such as underscores. With Omniscripts, the name may contain special characters. However, the unique combination of the Type, Subtype, and Language can't contain special characters.

After you enable Metadata API support, you won't be able to create a component whose unique name contains special characters.

See Also

[Enable Omnistudio Metadata API Support](#)

Omnistudio sObject Descriptions

The table on this page lists Omnistudio sObjects, their descriptions, and which objects you must grant read access to so that customers can view Omnistudio components.

sObject	Description	Read Access Required
Omni UI Card	Flexcards	Yes
Omni Process	Omniscripts and Integration Procedures	Yes
Omni Process Element	Omniscript elements	Yes
Omni Process Compilation	Compiled Integration Procedures and Omniscripts	Yes
Omni Electronic Signature Template	DocuSign signature templates used in Omniscripts. See Integrate DocuSign with Omniscripts .	No
Omni Data Transformation	Omnistudio Data Mappers	Yes
Omni Data Transformation Item	Data Mapper metadata	Yes
Omni Process Transient Data	Temporarily stored data for Omniscripts' and Integration Procedures' long running processes.	No
Omniscript Saved Session	Omniscript saved sessions	Yes
Omni DataPack	Collection of Omnistudio components and related functionality packaged for deploying objects from one	No

sObject	Description	Read Access Required
	sandbox or dev org to another, such as a production org.	

Newport Design System

Use the Newport Design System to design, customize, and preview your Omniscripts directly within the designer canvas without switching between design and preview modes. The system offers Newport and Newport Storybook themes in addition to the existing Lightning theme. The Newport theme is available in the designer canvas and preview alongside the Lightning theme. The Newport Storybook theme is available only in the designer preview, alongside the Newport and Lightning themes. Previously, the Omniscript Designer canvas supported only the Lightning theme, while preview mode supported both Newport and Lightning.

Switch between the Lightning, Newport, and Newport Storybook themes in real time to get instant visual feedback in the designer preview, similar to the package designer. Additionally, override default styles and upload custom design files to align with your branding requirements.

The Newport Design System is designed for both designers and web developers. It provides a centralized framework with custom, optimized CSS files for consistent rebranding and restyling of Omniscripts. Use this integration to streamline your design process.

- For designers: Use the new Design System to review all components and download a Sketch library to accelerate your design process.
- For developers: Visit the [Vlocity Github repository](#) to customize and rebrand all Vlocity Newport-based templates in one place.

The Newport design system offers these features:

- Real-time theme preview in the Omniscript designer canvas, with the ability to switch between Lightning, Newport, and Newport Storybook themes
- Seamless customization of Omniscripts and Flexcards, with support for overriding default styles to ensure consistent branding
- Consistent styling of components, with asynchronous style loading for a smooth design experience
- Streamlined design process, reducing the need to toggle between design and preview modes

[Use the Newport Design System](#)

The Newport Design System allows you to customize the appearance and behavior of Omniscript elements. Newport includes Storybook.js, a browser-based preview tool, that enables you to see your design changes in action.

Use the Newport Design System

The Newport Design System allows you to customize the appearance and behavior of Omniscript elements. Newport includes Storybook.js, a browser-based preview tool, that enables you to see your design changes in action.

Before You Begin

If you're unfamiliar with using command-line interfaces, refer to the [Command Line 101 guide](#).

System requirements:

- Git. See [Git documentation](#)
- Node.js v12 or higher. See [Node documentation](#).
- Gulp CLI: `Install via npm install --global gulp-cli`

Install the Newport Design System:

1. Clone the `vlocityinc/newport-design-system` repository from [GitHub](#) using a command-line interface. To see what components are provided by Newport, check the contents of the `UI/components` directory.
2. Change your current directory to `newport-design-system`:

```
cd newport-design-system
```

3. To work with the correct version of the Salesforce package, switch to the corresponding branch:

```
git checkout release_version
```

4. Install the dependencies:

```
npm install
```

5. Launch the Storybook.js preview:

```
npm start
```

6. Use the Storybook preview to review the available components and configurations.
7. Add customizations to your theme. See [Customize Your Omniscripts Using the Newport Design System](#).
8. Apply your customizations either to individual Omniscripts, or globally across your organization. See [Deploy and Apply Global Styling Changes using the Newport Design System](#).

Customize Your Omniscripts Using the Newport Design System

To customize the appearance of your Omniscripts, edit the CSS and HTML files in the `UI` subdirectory of the directory where you cloned Newport.

Themes are stored in the design-tokens folder. To create a new theme for your Omniscripts:

1. Copy the vlocity-newport-skin folder to a new folder.
2. Edit the files in the new folder as required.
3. To change the color scheme, modify the `color.yml` files. Colors are defined using hex RGB codes or tokens that resolve to hex values, which are defined in the `aliases/color.yml` file.

For example, to change the color of a specific component, such as the input element's orange underline:

- a. Edit `design-tokens/vlocity-newport-skin/background-color.yml`.
- b. Update `COLOR_BORDER_INPUT_ACTIVE` to a different color token defined in `/aliases/color.yml`. When you change colors, the Newport components are recompiled, so there is a slight delay before your changes are reflected in the Previewer. To verify your change, view the input component using the Previewer.

For optimal rendering on all platforms, set the control width to 6. Smaller widths may not render correctly on every platform.

4. To preview the effect of Newport Design System styles on an Omniscript:
 - a. Edit the Script Configuration and paste the path to the top-level CSS file (by default, `xxx.css`) into the **CUSTOM HTML TEMPLATES** field.
The Newport Previewer allows you to see how components render across desktops, tablets, and smartphones.
 - b. To preview an element or utility, choose it from the lists on the left pane. To view the code behind it, click `</>`.
 - c. To navigate the component hierarchy, use the restriction tree, which shows element definitions from base to derived implementations.
 - d. To view a specific definition, click the corresponding node in the restriction tree.

Deploy and Apply Global Styling Changes using the Newport Design System

Apply modifications to the Newport Design System by adding a static resource that overrides the Newport theme wherever it's available. After it's uploaded, the changes are applied to all Omniscripts that are on the Newport Design System.

1. Download the Newport Design System. See [Apply Global Branding to Omniscripts](#).
2. Create a distribution folder containing your changes by issuing this command: `npm run-script dist`
3. In versions before Vlocity Spring '19, change (cd) to the dist folder and zip its contents. (Do not zip the top-level dist folder: cd into it and zip its contents.)
4. In Salesforce, upload the zip file as a static resource.
5. In your Salesforce org, go to **Setup** and, in the Quick Find box, enter **Custom Settings**.
6. Find **UISettings**, click **Manage > New**.
7. Configure these settings:
 - **Name:** Enter `newportZipUrl` for the changes to be applied.
 - **Key:** Enter a custom name.

- **Value:** Enter the relative URL for the static resource containing your custom styles. To get this URL, go to **Static Resource** (see 4 in the Before You Begin section), hover over the **View File** link, right-click and copy the URL. Enter the relative URL after the domain name, removing the query string (question mark) at the end, leaving just the relative path. For example, the bold text would be copied in this URL: https://MyDomainName-c.vf.force.com/resource/16679825000/vloc_customNewport?
8. To enable Newport or Newport Storybook theme on an Omniscript, open an Omniscript and select **Newport** or **Newport Storybook** from the dropdown on the designer canvas or Preview page.

After selecting the Newport theme in the designer canvas, the Omniscript automatically appears in the Newport theme when previewed. Switching to Lightning in the preview page displays the Omniscript in the Lightning theme in the designer canvas.

-  **Note** When creating a new version of an Omniscript originally built with the Newport theme, the new version opens in the designer canvas with the Lightning theme selected by default.

Callable Implementations

Vlocity Apex classes and the Remote Actions of Omniscripts and Integration Procedures support the *Callable* interface.

Although the `VlocityOpenInterface` and `VlocityOpenInterface2` interfaces enable flexible implementations with a uniform signature, they reside in the Vlocity managed package and aren't truly Salesforce standards. However, classes that implement `VlocityOpenInterface` or `VlocityOpenInterface2` also implement the [Callable Interface](#), which is a Salesforce standard.

In addition, Remote Actions of Omniscripts and Integration Procedures can invoke any class that implements *Callable*, regardless of whether it implements `VlocityOpenInterface` or `VlocityOpenInterface2`. You specify the **Remote Class**, **Remote Method**, and **Additional Input** properties in the same way for classes that implement any of these interfaces.

To change custom classes that implement `VlocityOpenInterface` or `VlocityOpenInterface2` to *Callable* implementations, convert the signature with a few lines of code:

```
global with sharing class ClassName implements Callable
{
    public Object call(String action, Map<String, Object> args) {

        Map<String, Object> input = (Map<String, Object>)args.get('input');
        Map<String, Object> output = (Map<String, Object>)args.get('output');
        Map<String, Object> options = (Map<String, Object>)args.get('options');
    }
}
```

```
        return invokeMethod(action, input, output, options);
    }

    private Object invokeMethod(String methodName, Map<String, Object> inputMa
p, Map<String, Object> outMap, Map<String, Object> options) {
        ...
    }
    ...
}
```

For a full example, see [Make a Long-Running Remote Call Using Omnistudio.OmniContinuation](#).

Add Apex Class Permissions Checkers

Enable specific access to the classes used by remote action APIs for each user profile, permission set, or permission set group. Configure Apex class permissions checkers to ensure that users require explicit access to the Apex class that administers the remote action called from an Omniscript, Flexcard, Integration Procedure, or REST API.

For example, after you create a Lightning Platform site, publicly available APIs are enabled for the Site User profile based on the profile's Apex class access to some Apex classes. When you add an Apex class permissions checker, you ensure that unauthorized users, such as a guest user, can't access classes through the `ApexRemote` call implementing the `Callable` interface.

- ! **Important** Enable the `ApexClassCheck` setting to ensure the principle of least privilege and that unintentional unauthorized access isn't provided to guest users.
- ! **Note** Users who use Remote Actions in Integration Procedures must have access to the object and record they're invoking through the action. If they receive an access error, they must be granted object and record access at the profile, permission set, or permission set group level.

Enable the `ApexClassCheck` setting.

1. From Setup, in the Quick Find box, enter *Omni Interaction Configuration*, then select **Omni Interaction Configuration**.
2. Click **New**.
3. For Name and Label, enter `ApexClassCheck`.
4. For Value, enter `true`.
5. Save your changes.

- ! **Important** During the week of February 2, 2026, Salesforce enables the `ApexClassCheck` setting by default to enhance org security. Review and prepare your configuration for a seamless transition and to prevent potential service interruptions. See [Security Checks for Omnistudio](#).

View the Namespace and Version of Managed Packages

Omnistudio can be installed as a managed package. The package might be for a Cloud product (such as Insurance or Communications) or Omnistudio Foundation. You can check the namespace and version of your managed package, which you might need for planning upgrades or troubleshooting.

1. Go to **Setup**.
2. In the **Quick Find** box, enter *Installed Packages* and click **Installed Packages**.

Name	Type
Vlocity CPQ	Custom App
Language	Custom Field Definition
Vlocity OmniScript	Custom Object Definition
General Settings	Custom Setting
Service Assets (CardFramework)	Visualforce Page
AccountApplications	Visualforce Page
Account Record Page	Lightning Page
Testing Site Guest User	User

3. Verify that the latest version of the package is listed on the Installed Packages page.
4. On the Installed Packages page, click your current Omnistudio Vlocity package.

Action	Package Name	Publisher	Version Number	Namespace Prefix	Status	Allowed Licenses	Used Licenses
Uninstall Manage Licenses	Vlocity GMT	Vlocity, Inc	900.136	vlocity_cmt	Active	1	1

The resulting page displays the Omnistudio Vlocity package details including namespace, version number, and license information.

Package Details
Vlocity CMT (Managed)
« Back to List: Installed Package

Installed Package Detail

Package Name	Vlocity CMT	Version Number	900.136
Language	English	First Installed Version Number	900.73
Version Name	Winter 2018	Package Type	Managed
Namespace Prefix	vlocity_cmt	Allowed Licenses	1
Publisher	Vlocity, Inc.	Used Licenses	1
Status	Active	Modified By	Vlocity CME v101 Admin, 6/5/2018 11:44 AM
Expiration Date	Does not Expire		
Description		Tabs	76
Installed By	Vlocity CME v101 Admin, 6/5/2018 11:44 AM	Objects	265
Count Towards Limits	<input type="checkbox"/>		
Apps	2		

Standard Omnistudio Content and Runtime

With the Omnistudio permission set license, standard Omnistudio content runs in standard runtime. From the Lightning App and Experience Builders, add standard content to a Lightning or Experience Cloud site with standard Omniscript and Flexcard components. Standard Omnistudio components built by Salesforce Industries, such as Health Cloud and Financial Services Cloud, are available. You can also use standard components instead of custom-generated components to run custom content in the standard runtime.

Accessing Standard Content

Standard Omnistudio content is visible in the Omnistudio product home screens along with your custom Omnistudio content. However, we recommend that you use the standard Flexcard and Omniscript components to a Lightning page or an Experience Builder page after activation.

Disabling Managed Package Runtime

We recommend rebuilding Omnistudio for Managed Package components using Salesforce standard components for the following reasons:

- Components created in Omnistudio for Managed Package don't always work as expected in Omnistudio Standard.
- If you edit Omnistudio for Managed Package components after switching to the standard runtime, your changes aren't reflected in your site.

After activation, use the Lightning App Builder or Experience Builder to add the standard Flexcard or Omniscript component to a Lightning page or Aura-based Experience site. You can now add Omniscripts to LWR sites. See [Considerations for Using Omniscripts with Lightning Web Runtime Sites](#).

! **Important** To embed a Flexcard in an LWR Experience site, keep the Managed Package Runtime setting enabled and use the generated LWC.

To maintain backward compatibility with existing pages referencing Omniscript and Flexcard LWCs, LWCs

that existed before disabling Managed Package Runtime (formerly enabling Standard Omnistudio Runtime) will run in standard runtime the next time they're reactivated.

Viewing and Modifying Standard Content

- To let end users view a standard Flexcard or Omniscript in a Lightning page or Experience site, disable the Managed Package Runtime setting and confirm you have the Omnistudio permission set license.
- As an admin, to view and modify standard content in the designers, disable the Managed Package Runtime setting and install the Winter '23 or later managed package.

To modify standard components that ship with your Salesforce Industries license, you must create a version. Release updates are pushed to Version 1 of the components and are never editable. Your created version doesn't update with new releases.

 **Note** Omnistudio doesn't support the **Load Lightning Experience in LWR** checkbox in Setup > User > Edit. If you use Omnistudio in your org, disable this setting. Disabling this setting doesn't affect Omniscripts and Flexcards embedded on your LWR sites.

[View, Modify, and Enable Fast Activation on Omnistudio Content](#)

When you disable the Managed Package Runtime setting, you can view and modify standard Omnistudio content that ships with your Salesforce Industries license from the designers. You also enable faster Omniscript and Flexcard activation.

See Also

[Disable the Managed Package Runtime and Deploy Custom Lightning Web Components](#)

[Add a Flexcard to a Lightning Page](#)

[Add a Flexcard to an Experience Builder Page](#)

View, Modify, and Enable Fast Activation on Omnistudio Content

When you disable the Managed Package Runtime setting, you can view and modify standard Omnistudio content that ships with your Salesforce Industries license from the designers. You also enable faster Omniscript and Flexcard activation.

For functions and features not supported in Flexcard and Omniscript standard components, see [Omnistudio and Omnistudio for Managed Packages Features Support](#).

How Do I View and Modify Standard Content?

- As an admin, to view and modify Omnistudio content from the designers, disable Managed Package Runtime (formerly enable Standard Omnistudio Runtime) and install the Winter '23 or later package. Until you upgrade to the Winter '23 or later package, activating an Omnistudio component in a Designer generates an LWC. Therefore, you don't benefit from fast activation. Without the Winter '23 or later package, you can't modify the standard content that ships with your Salesforce Industries license. To modify standard content, you must create a version of the

component. Release updates to a standard component are pushed to Version 1 of the component, which is never editable. Your created versions don't update with new releases.

- As an end user, to view standard Flexcards or Omniscripts on a Lightning page or Experience site, disable Managed Package Runtime (formerly enable Standard Omnistudio Runtime) and confirm you have the Omnistudio permission set license.

Which Runtime Am I Using?

- If my Flexcard or Omniscript LWCs are published to Lightning or Experience pages...
 - and I disable package-based runtime: My content uses whichever runtime was enabled when the Flexcard or Omniscript was activated. If I deactivate and then reactivate my content, my LWCs automatically use the standard runtime.
 - and package-based runtime is enabled: My content uses the managed package runtime.
- If I have inactive Flexcards or Omniscripts...
 - and I disable package-based runtime: In Preview, my content uses the standard runtime. After activation, no LWCs are generated and my content uses the standard runtime. I must add Flexcard and Omniscripts to Lightning pages and Experience sites with the standard component in the Lightning App and Experience Site Builders.
 - and package-based runtime is enabled: In Preview, my content uses the managed package runtime. After activation, LWCs are generated and my content uses the managed package runtime. I must add Flexcard and Omniscripts to Lightning pages and Experience sites with a custom component in the Lightning App and Experience Site Builders.

See Also

[Disable the Managed Package Runtime and Deploy Custom Lightning Web Components](#)

Omnistudio

For your projects that use Omnistudio tools, and when you're on Omnistudio standard runtime with standard objects, use Salesforce CLI to move the components and metadata from one org to another org.

To learn more about how to install and configure Salesforce CLI, see [Salesforce CLI Developer Guide](#).

Use DevOps methods to create solutions with Omnistudio in a dev org. Then deploy those Omnistudio components to a test or integration org for testing purposes. Finally, deploy those components to your production org.



Note Deployment of components between orgs and migration from Omnistudio for Managed Packages to Omnistudio are different processes. See [Deploying Omnistudio Components Between Orgs or Migrating to Omnistudio Standard](#).

If you're using Omnistudio for Managed Packages, you must use different deployment tools. See [Deploy Omnistudio for Managed Packages Components Between Orgs](#).

[Tool Considerations for Deploying Omnistudio Components](#)

Select the tool for deploying components based on what you're deploying, the data model used in the source and target orgs, and the Omnistudio runtime used in the source and target orgs. The source and target orgs can be sandboxes, development orgs, or production orgs.

Tool Considerations for Deploying Omnistudio Components

Select the tool for deploying components based on what you're deploying, the data model used in the source and target orgs, and the Omnistudio runtime used in the source and target orgs. The source and target orgs can be sandboxes, development orgs, or production orgs.

We recommend that you use Salesforce CLI for deploying Omnistudio components between orgs when you're on Omnistudio standard runtime with standard objects.

Salesforce CLI is a Salesforce standard deployment tool. It caters across:

- Omnistudio components such as Omniscripts, Flexcards, Integration Procedures, and Data Mappers.
- Other Salesforce components such as Business Rules Engine.
- Metadata such as FlexiPage, PermissionSet, Omnistudio Lightning Web Components, and entities.

Salesforce CLI works only with standard objects in the source org and with standard objects and standard runtime in the target org. To learn more about the differences between Salesforce CLI and Build Tool, see [Deployment of Omnistudio for Managed Packages Components Between Orgs](#).



Note You can't deploy setup objects and non-setup objects together. Setup objects include FlexiPage, Profile, PermissionSet, and CustomField. Non-setup objects include Omnistudio entities such as OmniProcess and OmniUICard.

Deployment using Salesforce CLI provides these benefits:

- **Consistency:** One standard deployment tool for all your deployment needs. Ensures consistent deployments across different environments, minimizing discrepancies.
- **No learning curve:** For users familiar with command-line interfaces, Salesforce CLI provides an intuitive way to manage deployments. Users aren't required to learn complex UIs or tools.

Omnistudio

In addition to the managed package designer, Omnistudio offers a standard designer for Flexcards, Omniscripts, Integration Procedures, and Data Mappers in the standard runtime. The standard designer provides ease of navigation and improved performance. All the existing elements and configurations from the managed package designer are available in the standard designer.

-  **Note** The standard designer is available with Omnistudio only on the standard runtime. For new users, the standard designer is enabled by default. For existing users using the standard runtime, the standard designer is enabled by default after upgrading to Summer '25. If you enabled an Omnistudio license in Spring '25 without a package installation, you can get the Omnistudio standard designer by requesting for it with Salesforce Customer Support. Alternatively, you can install a Summer '25 managed package in your org and enable the relevant Omnistudio settings.

Omnistudio standard designer provides these benefits for each component, as compared to the managed package designer.

- Flexcard: Use this designer to create Flexcards in fewer steps. Drag elements to the canvas and edit their properties simultaneously, without switching between tabs. Dragging elements to the canvas is now twice as fast. Flexcard activation is up to 7 times faster.


- Omniscript: Drag elements to the canvas twice as fast and edit their properties simultaneously, without switching between tabs. Omniscript activation is up to 6 times faster.


- Integration Procedure: Create Integration Procedures up to 2.5 times faster than before. Access Integration Procedure settings in a single click and configure Integration Procedures in fewer steps. Easily insert, edit, or rearrange elements between blocks by using connectors. Drag elements to the canvas up to 7 times faster than before.


- Data Mapper: Create Data Mappers up to 6 times faster than before. Effortlessly access Data Mapper settings and visualize connections between objects.


-  **Important** The performance figures in this document are provided for informational purposes only and are based on internal validation and testing under specific conditions. Actual results may vary depending on your component design, production environment, and other factors. These figures provide general guidance and aren't a guarantee of performance.

All elements from the managed package designer are supported in the standard designer. Due to some considerations, you might still want to switch to the managed package designer after using the standard designer. Note that after you switch back to the managed package designer, you can't view or edit components made in the standard designer. However, you can still reference these components in runtime.

For a complete list of supported features across both managed package and standard designers, see [Supported Features in Omnistudio and Omnistudio for Managed Packages](#).

For a focussed list of feature support changes with the standard designer, see [Feature Support Changes with Standard Designers](#).

[Feature Support Changes with Standard Designers](#)

This table provides major feature support differences. It doesn't cover all supported features across both Omnistudio managed package and standard designers.

[Enable the Omnistudio Standard Designer](#)

The standard designer is available to new and existing users by default, depending on the Omnistudio license and version in your Salesforce org. If necessary, you can switch to the managed package designer. However, if you switch to the managed package designer, you can't edit components that are created or modified using the standard designer.

[Best Practices for Using the Omnistudio Standard Designer](#)

To troubleshoot an issue or to test new workflows, we recommend that you use a cloned Salesforce org. To directly use the standard designer in your sandbox and production environments, create a new version of your Omnistudio components and validate them in the standard designer. When you use the standard designer in your test orgs for the first time, open all Omnistudio components from the list view page.

[Hide the Omnistudio App While Using the Standard Designer](#)

To enable better performance and faster load times for list views, hide the Omnistudio app from App Launcher and make component search the default option for your org. This way, users can search for Flexcards, Omniscripts, Integration Procedures, or Data Mappers directly, instead of searching for the Omnistudio app.

[Access Omnistudio Designer Components from the List View](#)

View your Flexcards, Omniscripts, Integration Procedures, and Data Mappers on the list view pages, and open them in the standard designer. The list view page for the standard designer offers an enhanced user experience and better performance, when compared to the managed package designer.

[Omnistudio Frequently Asked Questions \(FAQs\)](#)

Get answers to common questions about using the Omnistudio standard designer.

See Also

[Export and Import Omniscripts](#)

[Export and Import Flexcards](#)

[Export or Import an Omnistudio Data Mapper](#)

[Export or Import an Omnistudio Integration Procedure](#)

[Differences Between Lightning and TinyMCE Rich Text Editors](#)

Feature Support Changes with Standard Designers

This table provides major feature support differences. It doesn't cover all supported features across both Omnistudio managed package and standard designers.

For a complete list of supported features across both designers, see [Supported Features in Omnistudio and Omnistudio for Managed Packages](#).

Feature	Applicable for	Supported on Standard designer?	Notes
Omniout	Flexcards, Omniscripts, Data Mappers, Integration Procedures	No	
In-Product Help	Flexcards, Omniscripts, Data Mappers, Integration Procedures	No	
Export and Import	Flexcards, Omniscripts, Data Mappers, Integration Procedures	Yes	Use Salesforce CLI.
Streaming API data source	Flexcards	Yes	Supported during run time, but not supported in the designer.
Rich text editor with TinyMCE	Flexcards and Omniscripts	Yes	<p>The standard designer integrates TinyMCE with these elements:</p> <ul style="list-style-type: none"> • Omniscript Step • Omniscript Disclosure • Omniscript Text Block • Flexcard Display Text
Lightning Rich Text Editor	Flexcards and Omniscripts	Yes	See Differences Between Lightning and TinyMCE Rich Text Editors .

Enable the Omnistudio Standard Designer

The standard designer is available to new and existing users by default, depending on the Omnistudio license and version in your Salesforce org. If necessary, you can switch to the managed package designer. However, if you switch to the managed package designer, you can't edit components that are created or modified using the standard designer.

Before you begin, make sure that you're on Omnistudio standard runtime.

To check whether the standard designer is available by default in your org, see [Omnistudio Standard Designer](#).

1. In Setup, find and select **Omnistudio Settings**.
2. Turn off the **Managed Package Designer** setting.

If an Omnistudio component is open in the managed package designer when you switch to the standard designer, an error appears.

[Retain the Designer Setting on Org Upgrade](#)

When you upgrade the package installation in your org, if you're using the standard runtime and package designer, the designer switches to standard designer after upgrade. To retain the designer of your choice, enable the `RetainDesignerSettingOnUpgrade` Omni Interaction Configuration.

Retain the Designer Setting on Org Upgrade

When you upgrade the package installation in your org, if you're using the standard runtime and package designer, the designer switches to standard designer after upgrade. To retain the designer of your choice, enable the `RetainDesignerSettingOnUpgrade` Omni Interaction Configuration.

To set the `RetainDesignerSettingOnUpgrade` Omni Interaction Configuration, perform these tasks.

1. In Setup, find and select **Omni Interaction Configuration**.
2. In the Label field, type `RetainDesignerSettingOnUpgrade`.
3. In the Value field, type `true`.
4. Save your configuration.

Best Practices for Using the Omnistudio Standard Designer

To troubleshoot an issue or to test new workflows, we recommend that you use a cloned Salesforce org. To directly use the standard designer in your sandbox and production environments, create a new version of your Omnistudio components and validate them in the standard designer. When you use the standard designer in your test orgs for the first time, open all Omnistudio components from the list view page.

Create a Test Org by Cloning Your Org

If you have a frequently used Salesforce org, test the standard designer in a different environment by cloning your existing org. This ensures that the test environment has the same setup and data as the original org.

1. If the standard designer is enabled by default in your existing org, enable the managed package designer by turning on the Managed Package Designer setting in Omnistudio Settings.
2. Clone your existing org.
3. To enable the standard designer in the cloned org, turn off the Managed Package Designer setting.
4. Validate your workflows in the test org.
5. If the validation is successful, enable the standard designer in your existing org.

Create New Component Versions

If you've activated versions of Omnistudio components, we recommend that you create a new version of the component before opening it in the standard designer. After you open a component in the standard designer, you can't edit it in the managed package designer. Create versions of Omnistudio components to track changes and maintain consistency when you switch between designers.

1. If the standard designer is enabled by default in your existing org, enable the managed package designer by turning on the Managed Package Designer setting.
2. From the list view page of the managed package designer, open the activated version of the component and create a new version.
3. Switch to the standard designer by turning off the Managed Package Designer setting.
4. From the list view page of the standard designer, open the new version of the component in the standard designer.
5. Validate the component in the standard designer.
6. Activate the component to use it in the Lightning and Experience sites.

Hide the Omnistudio App While Using the Standard Designer

To enable better performance and faster load times for list views, hide the Omnistudio app from App Launcher and make component search the default option for your org. This way, users can search for Flexcards, Omniscripts, Integration Procedures, or Data Mappers directly, instead of searching for the Omnistudio app.

Your org must have and use the Omnistudio standard designer.

As an admin, you can hide the Omnistudio app from specific user profiles. Once done, these users can search for the component they want to use and load the list view for the component directly.

1. From Setup, find and select **App Manager**.
2. Search for the Omnistudio app to find it.

3. Click the dropdown arrow next to the app and select **Edit**.
4. In the app editor:
 - a. Scroll to User Profiles.
 - b. Move the profiles that shouldn't have access to the app from **Selected Profiles to Available Profiles** by using the arrows.
5. Save your changes.

Access Omnistudio Designer Components from the List View

View your Flexcards, Omniscripts, Integration Procedures, and Data Mappers on the list view pages, and open them in the standard designer. The list view page for the standard designer offers an enhanced user experience and better performance, when compared to the managed package designer.

With Summer '25, the standard designer is available by default. If you haven't upgraded to this version, you can enable the standard designer by toggling it on in Omnistudio Settings.

-  **Note** You can't import and export Omnistudio components from the list page. Use Salesforce CLI instead.

1. After enabling the standard designer, remove any duplicate components in the App Launcher by hiding the duplicate tabs for the System Administrator profile.
 - a. From Setup, in the Quick Find box, enter *Profiles* and select it.
 - b. Edit the System Administrator profile.
 - c. From Custom Tab Settings, select and hide the duplicate component tabs.
 - d. Save your changes.
2. To launch a list view page, from the App Launcher, find and select your component.
 - Flexcards
 - Omniscripts
 - Data Mappers
 - Integration Procedures

The list view shows a maximum of 2,000 records. To view records beyond the limit, search, sort, or filter records.

The list view page shows the last modified version of each record.

3. Add and save filters to view only the records that meet your criteria.
 - a. Click .
 - b. Click **Add Filter** and set the filter criteria.
 - c. To add logic to refine which records appear in your list view, click **Filter Logic**. To reference filters in your logic statement, use the number assigned to each filter.
 - d. Save the filter.

To load this view as the default list, pin the list view.

The view is added to the list view dropdown list.

4. To search for a record, enter the search text in the list view search box.
- The search goes through only the first 2,000 records in the list. If you can't find your record, try a more specific search or modify your filters or sorting.

5. To view all versions of a record, click **Versions**. Alternatively, you can click the dropdown at the end of a row and select **Manage Versions**.

The list of all versions of the record appears. You can click the component version that you want to view and open it in its designer.

6. To create and customize your list views, click .

7. To open a record in the standard designer in a new tab, click the record in the list view.

The list view shows all the components managed either by the standard designer or the managed package designer. Check the Managed Using Standard Designer column to determine which designer is used for managing a component.

 **Note** After you open a component in the standard designer, you can no longer edit it in the managed package designer. However, you can still reference it at runtime

Omnistudio Frequently Asked Questions (FAQs)

Get answers to common questions about using the Omnistudio standard designer.

What is the Omnistudio standard designer?

The Omnistudio standard designer operates directly within the Salesforce Platform infrastructure, rather than through a managed package. Standard runtime refers to the runtime components of Flexcards, Data Mappers, Integration Procedures, and Omniscripts operating on the Salesforce Platform.

What are the advantages of using the standard designer?

Using the standard designer offers several advantages:

- Improved performance and a more robust user experience.
- Instant activation for Omniscripts, Integration Procedures, and Flexcards.
- Enhanced list pages built based on Salesforce standards, with better performance.
- A dedicated Manage Versions page for easier version comparison and switching.
- Intuitive canvas experience with easier element searching and consistent left-to-right drag-and-drop functionality.
- Simplified process for creating and activating Flexcards.

What changes are in the Integration Procedure standard designer?

The Integration Procedure standard designer features:

- An intuitive layout.
- Easier insertion of actions following the industry builder framework.
- A designer experience that's consistent with Salesforce design and user experience principles.
- A requirement to save after each edit for functional benefit.

- Flexibility to rearrange the order of steps right within the canvas.

Is the Omnistudio app available with the standard designer?

Yes. However, it is recommended that you search for the required Omnistudio component directly rather than the app. This provides better performance and loads your list views faster. If you're an admin, it is recommended that you restrict the user profiles that can search for the Omnistudio app, thus making component search the default usage pattern. For more information, see [Hide the Omnistudio App While Using the Standard Designer](#).

What changes are in the Data Mapper designer?

The Data Mapper standard designer provides:

- Simplified mapping functionality.
- A simplified layout.
- Easy insertion of steps.
- Flexibility to rearrange the order of steps right within the canvas.

How can I tell if my org is using standard runtime or standard designer?

Navigate to Setup, use Quick Find to search for Omnistudio Settings. Check the settings for Managed Package Runtime and Managed Package Designer. If a setting is on, it means that the runtime or designer are on the managed package.

Is some functionality not supported in the standard designer?

Exporting and importing directly from the standard designer using data packs isn't supported; use Salesforce CLI instead.

Is OmniOut supported with standard designer?

No, OmniOut is not supported. To use OmniOut, install a managed package in your org, and use the managed package designer.

Who is considered a new customer?

New customers are those who haven't installed Vlocity packages (vlocity_cme, vlocity_ins, vlocity_ps). They also don't have the Omnistudio package or Omnistudio licenses enabled.

Who is considered an existing customer?

Existing customers have installed Vlocity packages (vlocity_cme, vlocity_ins, vlocity_ps), the Omnistudio

package, or have Omnistudio licenses enabled.

If I'm a new customer starting with Summer '25, is installing a package necessary to use Omnistudio standard designer?

No. For new customers, enabling the Omnistudio License provides standard designer and standard runtime by default. However, if you plan to use the OmniOut functionality, you still need to install a managed package.

If I'm an existing customer with a managed package installation, what is the process to use the standard designer?

First, enable the Omnistudio license. Then, install the latest package. If you're on the managed package runtime, migrate to the standard runtime. After this, the standard designer and standard runtime are available.

I was already using the standard runtime in Spring '25 (or earlier). What happens when I upgrade to the Summer '25 package?

The Omnistudio standard designer is the default designer in your org.

I've been using the managed package designer and runtime. What happens when I upgrade to the Summer '25 package?

There is no change in this scenario. If the runtime was pointing to the package before the upgrade, the designers will also continue to point to the package after upgrading to Summer '25.

Can I switch back to using the managed package designer after moving to the standard designer?

Yes, an administrator can navigate to Setup > Omnistudio Settings and toggle the Managed Package Designer setting back to On.

What happens if I switch from the standard designer back to the managed package designer?

Components created or edited using the standard designer may not open correctly in the managed package designer due to compatibility differences. It's recommended to create new versions when switching.

I'm an existing customer who upgraded to Summer '25 with the managed package runtime. Can I choose to use the standard designer?

Yes. You'll be on the managed package designer and runtime by default. However, you can switch to the standard designer by turning off the Managed Package Designer setting. It's crucial to test this thoroughly in a sandbox environment first before enabling standard designer in production orgs.

If I had the managed package runtime enabled in Spring '25, can I use the standard designer after upgrading to Summer '25?

To enable standard designer, the org must first be using standard runtime. Before switching the standard runtime, consult with your account executive or cloud team to ensure there are no dependencies. Once on standard runtime, you can then enable the standard designer.

If I have enabled an Omnistudio license in Spring '25 without a package installation, will I be upgraded to the standard designer in Summer '25?

No, unless you had requested and used the Omnistudio standard designer in Spring '25. In Spring '25, the standard designer was available on request from Salesforce Customer Support.

You can either install a Summer '25 package or request the standard designer from Salesforce Customer Support.

What is the recommended best practice for moving to the standard runtime after upgrading to Summer '25?

If your workflows rely heavily on existing configurations and you were previously using the standard runtime, it's recommended that you first turn on the managed package runtime after the Summer '25 upgrade. Then, follow these steps.

- Clone an org that has been upgraded to Summer '25. See [Test the Standard Designer in a Cloned Org](#).
- In the cloned org, enable the standard designer.
- Thoroughly test all critical flows and components in the cloned org to verify that they function correctly with the standard designer.
- If testing is successful, you can then enable standard designer in your production org.

Alternatively, you can create a new version of the component you want to test and verify that all workflows work as expected. For instance, if you want to test an Omniscript that has ten versions and version 9 is the current active version, follow these steps.

1. Turn on the Managed Package Designer setting.
2. Clone version 9 of the Omniscript, which gives you version 11.
3. Turn off the Managed Package Designer setting.

4. Verify that version 11 of your Omniscript works as expected with the standard designer.

If your Omniscript works correctly, you can then enable the newly created version where needed.

Omnistudio

Omnistudio provides a set of tools and components that help you design, automate, and manage digital experiences and guided business processes. These include Omniscripts, Flexcards, Data Mappers, Integration Procedures, and other runtime components that work together to deliver configurable, scalable solutions across industries.

When planning your Omnistudio implementation, review and account for the Omnistudio limits and recommended thresholds. These limits help you understand optimal performance, maintainability, and scalability of your Omnistudio applications across environments. See [Recommended Limits](#).

For more information about the feature availability with Omnistudio licenses, see [Supported features in Omnistudio and Omnistudio for Managed Package](#) and [Feature Support Changes with Standard Designers](#).

Recommended Limits

When you approach, reach, or exceed the recommended limits while using Flexcards and Omniscripts, an informational or warning message appears on the Omnistudio UI. For more information, click the informational or warning message icon.

Recommended Limits

When you approach, reach, or exceed the recommended limits while using Flexcards and Omniscripts, an informational or warning message appears on the Omnistudio UI. For more information, click the informational or warning message icon.

Recommended Limits for Flexcards

Component Name	Description	Recommended Limits
Data Table rows	Maximum number of rows in a Data Table.	750
Child Flexcards	Maximum number of child Flexcards within a parent Flexcard.	7
Child Omniscripts	Maximum number of child Omniscripts within a Flexcard.	3
Recursive Flexcards	Maximum number of recursive cards within a parent Flexcard.	1
Event Listeners	Maximum number of event listeners you can add.	20
Data Mappers	Maximum number of Data Mappers within a Flexcard.	20
Flexcard versions	Maximum number of Flexcard versions you can create.	10
Integration Procedures	Maximum number of Integration Procedures within a Flexcard.	20

Recommended Limits for Omniscripts

Component Name	Description	Recommended Limits
Action Blocks	Maximum number of Action Blocks.	10
Data Mappers	Maximum number of Data Mappers.	5

Component Name	Description	Recommended Limits
Omniscripts	Maximum number of Omniscript components you can include in an Omniscript.	750
Child Omniscripts	Maximum number of child Omniscripts.	20
PDF template file size	Maximum allowed file size of a PDF template.	250 KB
Custom LWCs	Maximum number of Custom LWCs.	30
Nested Blocks	Maximum number of nested Blocks.	10
Omniscript LWC template file size	Maximum allowed file size of an Omniscript LWC template.	4 MB
Edit Blocks	Maximum number of Edit Blocks.	10
Conditional Actions	Maximum number of Conditional Actions.	20

Omnistudio

The security checks for Omnistudio enforce stricter validation of access and permissions. These checks make sure that users, including guest, authenticated, and non-authenticated users, have the appropriate access to objects and fields used within Omnistudio components.

These feature flags control the Omnistudio security checks. The security checks only apply when the flag is enabled.

- `ApexClassCheck` : Requires users to have explicit Apex class access for any remote actions called from Omniscripts or Flexcards. See [Add an Apex Class Permissions Checker](#).
- `EnforceDMFLSAndDataEncryption` : Automatically enforces Object and field-level security (FLS) for all Data Mappers. Users without the `View Encrypted Data` permission can't view encrypted fields in plain text. See [Security for Omnistudio Data Mappers and Integration Procedures](#).
- `EnableQueryWithFLS` : Enforces FLS for all Salesforce Object Search Language (SOSL) and Salesforce Object Query Language (SOQL) queries within Flexcards, ensuring data visibility respects user permissions. See [Set Up a Data Source on a Flexcard](#).

There are no behavioral changes to these previously announced security flags. Enabling them continues to support consistent enforcement of data access and security controls across Omnistudio components.

To enable these Omnistudio security flags:

1. From Setup, in the Quick Find box, enter *Omni Interaction Configuration*, and then select **Omni Interaction Configuration**.
2. Click **New Omni Interaction Configuration**.
3. In the Label field, enter the flag name. For example, `ApexClassCheck`.
4. In the Value field, enter `true`.

! **Important** During the week of February 2, 2026, Salesforce enables the `ApexClassCheck`, `EnforceDMFLSAndDataEncryption`, and `EnableQueryWithFLS` settings by default to enhance org security. Review and prepare your configuration for a seamless transition and to prevent potential service interruptions.

Omnistudio

Omniscripts help you create dynamic customer interactions with low code and deploy them to multiple channels and devices. You can create an Omniscript and then deploy it on a Salesforce application, a mobile device, an Experience Cloud site, or a web page.

Create Omniscripts to guide users through sales and service processes with fast, personalized responses. Omniscripts also integrate seamlessly with enterprise applications and data.

Why use Omniscripts?

Create interactive experiences with Omniscripts to move quickly from development to production. With a fast drag-and-drop editor, add data inputs, outputs, and actions to an Omniscript without having to create custom coding or styling. If you need a branded or custom design, you can also easily customize Omniscripts. For a good customer experience, your Omniscript shows their task progress and guides them, like help text or helpful error messages.

Omniscripts are easy to develop, too. You can build reusable Omniscripts for common tasks in different workflows. You can also create one Omniscript to use across different channels, as shown in this diagram.



What are Omniscripts?

Think of an Omniscript as a guided workflow or process, such as a customer updating their contact or healthcare information. They first enter their patient or personal identifying information, and the Omniscript gets and displays their current information. A customer then updates their address or current medications. The customer's data is transformed to a specific format, such as the customer's date of birth converted to a specific date format. Their changes and new data are saved, including deletions. If a customer deletes a payment method or a medication they no longer take, their changes are saved. The Omniscript then sends them an email with the updates and confirmation.

An Omniscript has elements to complete those basic tasks: Get data, display data, update data, write data, and take an action.

An Omniscript can get or update data from a Salesforce object or from internal and external data sources. It can use an Omnistudio Data Mapper to retrieve data from sObjects or an Integration

Procedure. To get data from other sources, an Omniscript can work with REST APIs or Apex.



Working with Omniscripts

Use the Omniscript Builder to create and manage guided, interactive processes. Retrieve, update, and add data to a variety of sources with elements for Data Mappers, Integration Procedures, REST APIs, and Apex. Or allow users to add or modify data with input fields. Drag Omniscript elements to the canvas and edit properties without changing tabs. Omniscript activation is instant.

Create an Omniscript

Create an Omniscript to build a form in the Omniscript Designer.

Create Multi-Language Omniscripts

Enable a single Omniscript to run in multiple languages using custom labels.

Build an Omniscript with Elements

For guided workflows, Omniscripts use basic tasks: Get, save, modify, and display data from Salesforce objects and other data sources. They can also allow users to add, change, or remove data and can take actions, such as getting a DocuSign signature, sending an email, or opening a web page.

Configure Omniscript Settings

Configure optional settings for your Omniscript in the Setup panel of the Omniscript Designer, such as making your Omniscript reusable or enabling logging.

Integrate DocuSign with Omniscripts

Enable users to either sign DocuSign forms in an Omniscript or email a user a copy of the document to sign at their convenience. DocuSign integration requires an active DocuSign account.

Customize Omniscript Behaviors, Style, and Elements

To meet your business needs and improve user experience, you can customize Omniscripts for their style and appearance and for their actions. Apply custom styling via static resources, global stylesheets, or SLDS design tokens. Modify Omniscripts to conditionally hide or show elements and errors and to trigger platform events. For more complex customization, you can extend Omniscript elements to add custom behavior and styling. Finally, you can create custom Lightning web components (LWCs) to add custom HTML, JavaScript, and CSS.

Preview and Test an Omniscript

View the Omniscript's appearance and functionality by previewing the Omniscript in the Omniscript Designer.

Activate and Launch Omniscripts

Make Omniscripts available to Experience Sites, Lightning Pages, custom LWCs, and Lightning tabs by activating and launching the Omniscript. If an error occurs during deployment, the Omniscript is deactivated.

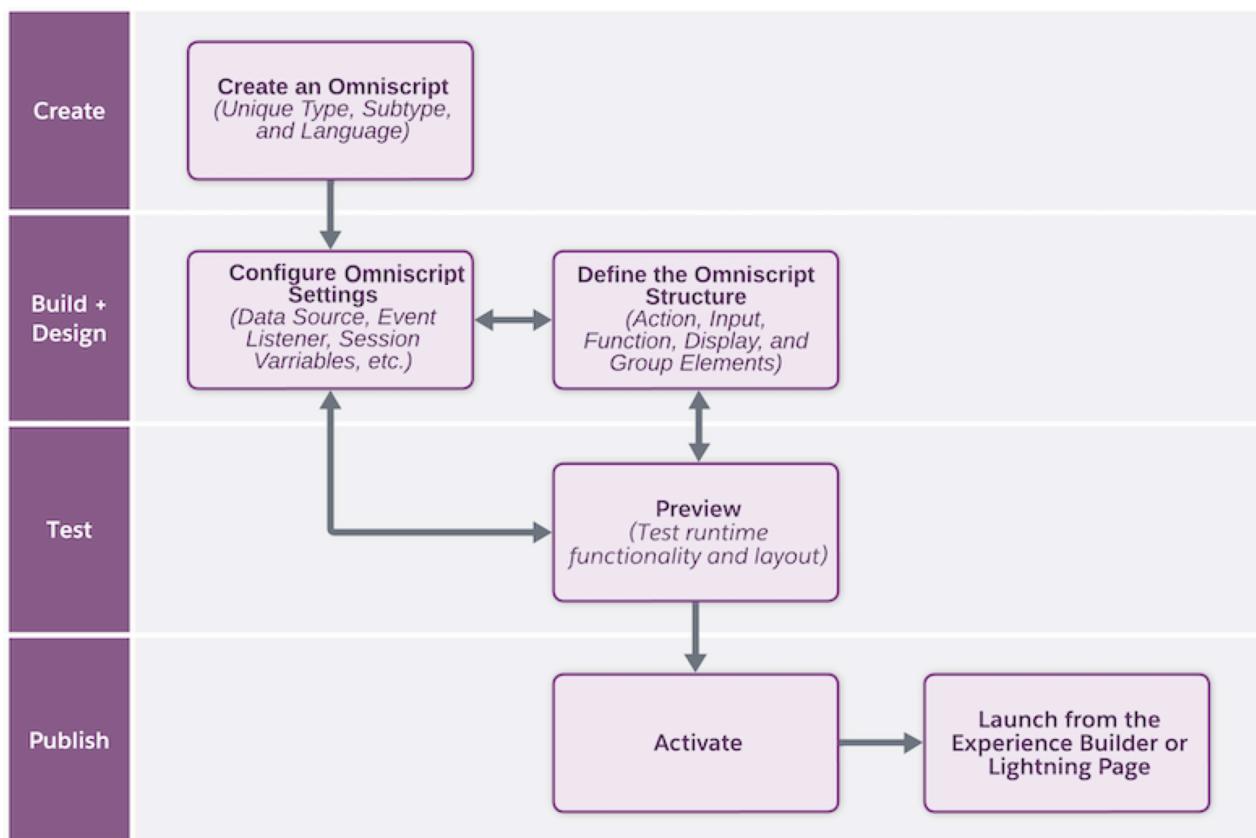
Omniscript Element Reference

The available elements are Omniscripts are organized as groups, Data Mapper actions, standard actions, display elements, functions, input, and Omniscripts (to embed reusable Omniscripts). Each element uses an extendable Lightning web component (LWC).

Working with Omniscripts

Use the Omniscript Builder to create and manage guided, interactive processes. Retrieve, update, and add data to a variety of sources with elements for Data Mappers, Integration Procedures, REST APIs, and Apex. Or allow users to add or modify data with input fields. Drag Omniscript elements to the canvas and edit properties without changing tabs. Omniscript activation is instant.

This diagram shows the high-level workflow for Omniscripts in the new designer and the designer on a managed package:



1. Create the Omniscript with a unique combination of Type, Subtype, and Language.
See [Create an Omniscript](#) or [Create Multi-Language Omniscripts](#).
2. Configure optional settings for your Omniscript, such as making your Omniscript reusable. Create the logic based on your requirements by building a hierarchical structure of action elements and steps in the Omniscript.
See [Configure Omniscript Settings](#) and [Omniscript Element Reference](#).
3. Verify the Omniscript's appearance and functionality by previewing the Omniscript. Test your Omniscript by adding test data, viewing the data JSON, and debugging action requests and response data.
See [Preview and Test an Omniscript](#)

4. After you confirm that the Omniscript is ready, activate it. Make an active Omniscript available for launch:
 - In the new designer, launch the Omniscript as a URL.
 - In the designer for a managed package, launch the Omniscript as a standalone Lightning web component on a Lightning web page, as an embedded link in another LWC or as a URL.
- See [Activate and Launch Omniscripts](#).

Export and Import Omniscripts

Export or import Omniscripts to use them in another org, such as when you create them in a sandbox or to use sample data packs that Salesforce provides. When exporting or importing data packs for Omniscripts, if you encounter Apex limit errors such as heap or CPU limits, reduce the size of the data pack by unselecting dependencies during the export process. Additionally, break the data pack into multiple smaller data packs. As a best practice, we recommend including no more than 10 elements in each data pack.

Export and Import Omniscripts

Export or import Omniscripts to use them in another org, such as when you create them in a sandbox or to use sample data packs that Salesforce provides. When exporting or importing data packs for Omniscripts, if you encounter Apex limit errors such as heap or CPU limits, reduce the size of the data pack by unselecting dependencies during the export process. Additionally, break the data pack into multiple smaller data packs. As a best practice, we recommend including no more than 10 elements in each data pack.

See Also

- [Create a DataPack or MultiPack](#)
- [Import a DataPack](#)
- [Data Deployment with Data Packs](#)

Export an Omniscript

To export an Omniscript in the new designer on the standard runtime, use the Salesforce CLI. Omnistudio exports the file in JSON format.

Before you begin:

- [Set up Salesforce CLI](#).
- [Authorize your org](#).
- Export any associated Data Mappers, Integration Procedures, and Flexcards, in that order.
See [Export or Import an Omnistudio Data Mapper](#), [Export or Import an Omnistudio Integration Procedure](#), and [Export and Import Flexcards](#).

1. From a terminal in VS Code, enter this command, and replace the variables with values for your org.

```
sfdx force:data:tree:export -q "SELECT IsMetadataCacheDisabled, IsTestProced
```

```
ure, Description, OverrideKey, Name, OmniProcessKey, Language, PropertySetCo
nfig, LastPreviewPage, OmniProcessType, ElementTypeComponentMapping, SubTyp
e, ResponseCacheType, IsOmniScriptEmbeddable, CustomJavaScript, IsIntegratio
nProcedure, VersionNumber, DesignerCustomizationType, Namespace, Type, Requi
redPermission, WebComponentKey, IsWebCompEnabled, (SELECT Description, Design
erCustomizationType, Name, EmbeddedOmniScriptKey, IsActive, Type, ParentElem
entId, PropertySetConfig, SequenceNumber, Level, Id from OmniProcessElement
s) from OmniProcess where OmniProcessType='Omniscript' and id='omniscript_i
d'" -u {org_alias} -p -d export_directory -p
```



Note Don't include the `isActive` field on OmniProcess when you export the Omniscript.

org_alias

The alias of the org from where you're exporting.

export_directory

The directory in the project where the exported JSON is stored.

omniscript_id

The ID of the Omniscript that you want to export.

2. Verify whether the JSON file is downloaded in the export directory.

Import an Omniscript

To import an Omniscript in the new designer on the standard runtime, use the Salesforce CLI.

Before you begin:

- Set up Salesforce CLI.
- Import any associated Data Mappers, Integration Procedures, and Flexcards, in that order.
See [Export or Import an Omnistudio Data Mapper](#), [Export or Import an Omnistudio Integration Procedure](#), and [Export and Import Flexcards](#).

From a terminal in VS Code, enter this command, and replace the variables with values for your org.

```
sfdx force:data:tree:import -p ./export_directory/OmniProcess-OmniProcessEleme
nt-plan.json -u {org_alias}
```

org_alias

The alias of the org from where you're exporting.

export_directory

The directory in the project where the exported JSON is stored.

-  **Note** If the exported JSON file has `isActive=true`, the import operation fails.

Export an Omniscript from the Designer on a Managed Package

Export Omniscripts in a data pack to share them with other Salesforce orgs.

1. From the App Launcher, find and select **Omniscripts**.
2. Expand an Omniscript and select the version to export.
3. Click , and then select **Export**.
4. Follow the instructions and enter the information as needed.
5. To store the data pack and access it from Omni DataPacks, select **Add To Library**.
6. To store the data pack on your local machine, select **Download**.
7. Click **Done**.

Import an Omniscript to the Designer on a Managed Package

If you're using the designer on a managed package, add existing Omniscripts to an org by importing them with a data pack.

1. From the App Launcher, find and select **Omniscripts**.
2. Click **Import** and then follow the instructions.

Create an Omniscript

Create an Omniscript to build a form in the Omniscript Designer.

1. From the App Launcher, find and select **Omniscripts**.
2. Click **New**.
3. Enter a **Name** for your Omniscript.
4. Enter a **Type**, **SubType**, and **Language**.

The Type must start with a letter and contain only alphanumeric characters without spaces or underscores. Subtype must be an alphanumeric text and contain no spaces or underscores. The Type, SubType, and Language combine to create a unique identifier that becomes the name of a compiled Omniscript's Lightning web component. For example, an Omniscript where `Type=account`, `SubType=Create`, and `Language=English` generates an LWC named **accountCreateEnglish**.

-  **Note** The combined Type, SubType, and Language mustn't exceed 60 characters.

5. If needed, enter a **Description** for your Omniscript.
6. Click **Save**.

Considerations for Single-Language Omniscripts

Omnistudio provides translations for system labels, such as standard error messages. In your single-

language Omniscripts, these translations automatically appear based on the user's locale. For example, if the user's locale is set to Japanese, all system-generated error messages also appear in Japanese.

[Upgrade an Omniscript's Property Set](#)

When using an existing Omniscript in the Omniscript Designer, create a new version of the Omniscript to update the Omniscript's property set. In an org with multiple developers, errors could appear in the designer when using the same Omniscript without a new version. Prevent or resolve this error by upgrading the Omniscript's property set.

Considerations for Single-Language Omniscripts

Omnistudio provides translations for system labels, such as standard error messages. In your single-language Omniscripts, these translations automatically appear based on the user's locale. For example, if the user's locale is set to Japanese, all system-generated error messages also appear in Japanese.

In the Omnistudio managed package runtime, you can add translations for the system labels using custom labels. In the standard runtime, Omniscripts use autotranslated system labels. After migrating to the Omnistudio standard runtime, custom label translations set up in the managed package runtime are overridden by the autotranslated labels.

The autotranslated system labels don't affect multi-language Omniscripts. These Omniscripts continue to use the translations defined using custom labels.

Upgrade an Omniscript's Property Set

When using an existing Omniscript in the Omniscript Designer, create a new version of the Omniscript to update the Omniscript's property set. In an org with multiple developers, errors could appear in the designer when using the same Omniscript without a new version. Prevent or resolve this error by upgrading the Omniscript's property set.

Without a new version, errors could appear in the designer. Prevent or resolve this error by upgrading the Omniscript's property set.

1. From your Omniscript, click **New Version** to trigger a class that adds any missing JSON properties.
2. Repeat the process for any older or imported Omniscripts.

Create Multi-Language Omniscripts

Enable a single Omniscript to run in multiple languages using custom labels.

Before you begin, enable Translation Workbench. See [Enable or Disable Translation Workbench](#).

Note these considerations for multi-language Omniscripts:

- The language users see in Omniscript picklists depends on the language setting in Language and Timezone in Setup.
- Multi-language Omniscripts provide beta support for right-to-left languages. When using RTL languages, some styling may result in functional limitations.
- Due to a known Salesforce issue, multi-language Omniscripts don't display correctly in Community Builder.

For single-language Omniscripts, Omnistudio provides autotranslated system labels, such as standard error messages. These translations automatically appear based on the user's locale.

[Enable Multi-Language Omniscript Support](#)

Enable multi-language Omniscript support by modifying the Omni Process object in Salesforce Setup.

[Define Custom Label Translations in Multi-Language Omniscripts](#)

Define translations in various languages for custom labels in multi-language Omniscripts. Create localized content so that users see the Omniscripts in the language based on their locale. Create custom labels in English first, followed by translations for other languages.

[Test a Multi-Language Omniscript](#)

View custom translations using the Omniscript Designer preview.

[Launch a Multi-Language Omniscript](#)

Set the language a multi-language Omniscript renders in by using a language code parameter in a URL.

Enable Multi-Language Omniscript Support

Enable multi-language Omniscript support by modifying the Omni Process object in Salesforce Setup.

1. From Salesforce Setup, select **Object Manager**.
2. Locate and click the **Omni Process** object.
The custom object definition is displayed.
3. In the **Fields and Relationships** section, click to access **Language**.
4. If **Multi-Language** isn't already in the **Language Picklist Values** related list, click **New** in the related list header.
5. Add *Multi-Language* to the text box.
6. Click **Save**.
7. In your Omniscript, click **Edit**, and set **Language** to *Multi-Language*.

Define Custom Label Translations in Multi-Language Omniscripts

Define translations in various languages for custom labels in multi-language Omniscripts. Create localized content so that users see the Omniscripts in the language based on their locale. Create custom labels in English first, followed by translations for other languages.

To avoid duplications and improve querying, use naming conventions when you create custom labels in a Salesforce org with multiple developers. For example, *jSmithCustomTextLabel*.

-  **Note** In addition to Salesforce custom labels, Omnistudio provides default custom labels with translations. Omnistudio doesn't support Salesforce standard labels. The Set Values and Set Errors actions don't support custom labels.

1. Create a multi-language Omniscript.
 - a. From the App Launcher, find and select **Omniscripts**.
 - b. On the Omniscripts tab, click **New**.
 - c. Enter a unique name, type, and subtype for your Omniscript.
 - d. From the language dropdown list, remove English and select **Multi-Language**.
 - e. Save your changes.
 2. Create the Omniscript's basic structure by adding elements and their properties.
 3. Add or edit translations for custom labels for elements in the Omniscript.
 - Click .
 - If you're using the designer for a managed package, click **Edit Translations**.
The Edit Your Omniscript Translations window opens.
 4. Add English translations for the custom labels.
 - a. Click  and select **English (US)** from the dropdown list.

 **Note** To ensure that the custom label is created properly, always create the English translation first.

b. For each element that you want to translate, enter a custom label name.

c. In **Custom Label Value in English (US)**, enter the value that you want for the element in the English Omniscript.
Merge field syntax is supported in custom label values.

 **Note** Custom labels must have values. If a custom label value doesn't exist, an error message appears when you preview the Omniscript.

d. Click **Save as English (US)**.
 5. Create another translation.
 - a. Click  and select another language.
 - b. Add a translation by entering custom label values for the element properties for which you entered English values.
 - c. Save the translations.
 6. Add other languages as necessary.
 7. Preview the Omniscript and test each language.
Sometimes it takes a few minutes for the custom labels to be created. If the labels don't appear in the preview at first, try again in a few minutes.
-  **Note** Custom labels don't render in the design mode. You must preview the Omniscript to render the custom labels correctly.

If the **Save** operation is not completed error appears, or if labels don't appear in preview after a few minutes, add the custom labels in Setup. See [Create and Edit Custom Labels](#).

[Access Custom Labels in an Omniscript Custom Lightning Web Component](#)

Provide custom Lightning web components access to Salesforce custom labels in your Omniscript.

Translate Tooltip Help Text in Omniscripts

Display tooltip help text with multiple translations in multi-language Omniscripts. You must use custom labels to add help text in multi-language Omniscripts. If a custom label is not defined for a help text tooltip, an error renders.

Translate Custom Labels for Date and Time Components

Define custom label translations for date and time pickers using JSON strings and Omniscript custom labels. For example, if a Spanish translation is defined, the date picker displays Spanish translations for the months and days of the week.

Create Multi-Language Select Elements

Populate Select, Multi-select, and Radio elements for a multi-language Omniscript using custom configuration options.

Omniscript Custom Label Reference

Review the Omniscript element properties for which you must define custom labels and translations for multi-language Omniscripts. To use a default translation, clear the property.

Example: Translate an Omniscript by Using Custom Labels

This example illustrates how you can translate Text and Text Block elements by using custom labels. You can follow similar steps to translate other Omniscript elements.

1. Create a multi-language Omniscript.
 - a. From the App Launcher, find and select **Omniscripts**.
 - b. On the Omniscripts tab, click **New**.
 - c. Enter a unique name, type, and subtype for your Omniscript.
 - d. From the language dropdown list, remove English and select **Multi-Language**.
 - e. Save your changes.
2. Configure the multi-language Omniscript.
 - a. From the elements panel, drag a second step in the Omniscript.
 - b. In the properties of the second step, remove the **Step2** value from the field label.
 - c. Expand both steps.
 - d. From the elements panel, drag a **Text** element into the first step.
 - e. From the elements panel, drag a **Text Block** element into the second step.
3. In the Edit Your Omniscript Translations window, edit the custom label values in English.
 - a. In the Text1 element's field label row, in Custom Label Name, enter *Name* and in Custom Label Value in English (US), enter *Name*.
 - b. In the TextBlock1 element's text key row, in Custom Label Name, enter *Greeting* and in Custom Label Value in English (US), enter *<ph>Hello, </ph><ph>%Step1:Text1%</ph>*.
A custom label value for a Text Block can contain HTML markup. To ensure that a merge field value renders correctly, put the merge field between **<ph>** HTML tags. To begin on a new line, use **
**.
- c. Save the English translations.
Two custom labels are created: Name and Greeting.

4. Create a Spanish translation for the Omniscript.

- a. In the Text1 element's field label row, enter *NOMBRE* for the custom label value in Spanish.

- b. In the TextBlock1 element's text key row, enter <*ph>Hola, </ph><ph>%Step1:Text1%</ph>* for the custom label value in Spanish.
- c. Save the Spanish translations.

Here's how the first page looks.

The screenshot shows a form with a single step. On the left, there is a text input field labeled "Nombre" containing "Juan". On the right, there is a vertical "Steps" column with one step highlighted in blue. At the bottom, there are "Save for later" and "Next" buttons.

Here's how the second page looks.

The screenshot shows a form with two steps completed. On the left, the text input field now displays "Hola, Juan". On the right, the "Steps" column shows two steps: the first is checked with a green checkmark, and the second is also checked. At the bottom, there is a "Previous" button.

Access Custom Labels in an Omniscript Custom Lightning Web Component

Provide custom Lightning web components access to Salesforce custom labels in your Omniscript.

Custom Lightning web components that extend the Omniscript Base Mixin component or an Omniscript element's Lightning web component can access both Omniscript custom labels and Salesforce custom labels. For information on custom LWCs in Omniscript, see [Create a Custom Lightning Web Component for Omniscript](#).

1. In the Omniscript, click **Edit as JSON**.
2. In the JSON, add the node `"moreCustomLabels":` and set the node's value equal to an array of the custom labels that do not exist in the OS.

```
"moreCustomLabels" : ["myCustomLabel1", "orgCustomLabel2"]
```

3. In your custom LWC's code, add a line of code to access a map of all the custom labels using the **allCustomLabels** attribute. Each component type requires a different attribute name to access the **allCustomLabels** attribute.

Component Type	Syntax
Component extending the Omniscript Base Mixin component	<code>this.omniScriptHeaderDef.allCustomLabels</code>
Component extending an Omniscript element's LWC	<code>this.scriptHeaderDef.allCustomLabels</code>

4. Using dot notation, append the name of the attribute the custom LWC requires to access the label.

Component Type	Syntax
Component extending the Omniscript Base Mixin component	<code>this.omniScriptHeaderDef.allCustomLabels.myCustomLabel</code>
Component extending an Omniscript element's LWC	<code>this.scriptHeaderDef.allCustomLabels.myCustomLabel</code>

5. Deploy the custom Lightning components to a Salesforce org.
 6. Activate and preview the Omniscript to ensure the label displays correctly.

See Also

- [Deploy Lightning Web Components](#)
- [Define Custom Label Translations in Multi-Language Omniscripts](#)
- [Custom LWC Element](#)

Translate Tooltip Help Text in Omniscripts

Display tooltip help text with multiple translations in multi-language Omniscripts. You must use custom labels to add help text in multi-language Omniscripts. If a custom label is not defined for a help text tooltip, an error renders.

1. In your Omniscript, select an element that requires help text translation.
2. In the element's properties, click **Help Options**.
3. Select **Show help text**. In the designer for a managed package, select **Activate Help Text**.
4. In the **Help Text** field, enter the custom label.
5. Preview the Omniscript to render the custom labels.

Translate Custom Labels for Date and Time Components

Define custom label translations for date and time pickers using JSON strings and Omniscript custom labels. For example, if a Spanish translation is defined, the date picker displays Spanish translations for the months and days of the week.

Learn how to translate custom labels in Salesforce. See [Translate Custom Labels](#).

1. Go to [Day.js](#), click **List of supported locales**, and choose a language file.
2. In the language's JavaScript file, within the `locale` object, copy the outer braces and everything in between.

For example, in the `es-us.js` file, copy the highlighted text:

```
// Spanish (United States) [es-us]
import dayjs from 'dayjs'

const locale = {
  name: 'es-us',
  weekdays: 'domingo_lunes_martes_miércoles_jueves_viernes_sábado'.split('_'),
  weekdaysShort: 'dom._lun._mar._mié._jue._vie._sáb.'.split('_'),
  weekdaysMin: 'do_lu_ma_mi_ju_vi_sá'.split('_'),
  months: 'enero_febrero_marzo_abril_mayo_junio_julio_agosto_septiembre_octubre_noviembre_diciembre'.split('_'),
  monthsShort: 'ene_feb_mar_abr_may_jun_jul_ago_sep_oct_nov_dic'.split('_'),
  relativeTime: {
    future: 'en %s',
    past: 'hace %s',
    s: 'unos segundos',
    m: 'un minuto',
    mm: '%d minutos',
    h: 'una hora',
    hh: '%d horas',
    d: 'un día',
    dd: '%d días',
    M: 'un mes',
    MM: '%d meses',
    Y: 'un año',
    yy: '%d años'
  },
  ordinal: n => `${n}º`,
  formats: {
    LT: 'h:mm A',
    LTS: 'h:mm:ss A',
  }
}
```

```
L: 'MM/DD/YYYY',
LL: 'D [de] MMMM [de] YYYY',
LLL: 'D [de] MMMM [de] YYYY h:mm A',
LLLL: 'dddd, D [de] MMMM [de] YYYY h:mm A'

}

}

dayjs.locale(locale, null, true)

export default locale
```

3. Paste the JavaScript code you copied into a JavaScript-to-JSON converter such as <https://www.convertsimple.com/convert-javascript-to-json/>, and convert it.

The converted JSON code looks like this:

```
{
  "name": "es-us",
  "weekdays": [
    "domingo",
    "lunes",
    "martes",
    "miércoles",
    "jueves",
    "viernes",
    "sábado"
  ],
  "weekdaysShort": [
    "dom.",
    "lun.",
    "mar.",
    "mié.",
    "jue.",
    "vie.",
    "sáb."
  ],
  "weekdaysMin": [
    "do",
    "lu",
    "ma",
    "mi",
    "ju",
    "vi",
    "sá"
  ]
}
```

```
],  
  "months": [  
    "enero",  
    "febrero",  
    "marzo",  
    "abril",  
    "mayo",  
    "junio",  
    "julio",  
    "agosto",  
    "septiembre",  
    "octubre",  
    "noviembre",  
    "diciembre"  
,  
  "monthsShort": [  
    "ene",  
    "feb",  
    "mar",  
    "abr",  
    "may",  
    "jun",  
    "jul",  
    "ago",  
    "sep",  
    "oct",  
    "nov",  
    "dic"  
,  
  "relativeTime": {  
    "future": "en %s",  
    "past": "hace %s",  
    "s": "unos segundos",  
    "m": "un minuto",  
    "mm": "%d minutos",  
    "h": "una hora",  
    "hh": "%d horas",  
    "d": "un día",  
    "dd": "%d días",  
    "M": "un mes",  
    "MM": "%d meses",  
    "y": "un año",  
    "yy": "%d años"  
,  
  },
```

```

"formats": {
    "LT": "h:mm A",
    "LTS": "h:mm:ss A",
    "L": "MM/DD/YYYY",
    "LL": "D [de] MMMM [de] YYYY",
    "LLL": "D [de] MMMM [de] YYYY h:mm A",
    "LLLL": "dddd, D [de] MMMM [de] YYYY h:mm A"
}
}

```



Note Some JavaScript-to-JSON converters don't support functions such as

`relativeTimeFormatter`. If a locale uses `relativeTimeFormatter`, either remove the `relativeTime` key and everything within its braces, or add values for `mm`, `hh`, and so on. Or if you're familiar with JavaScript, you can run the `.js` file, run the `JSON.stringify(locale)` command, and skip the next step.

- Remove line breaks and extra white space using a JSON formatter with a Compact JSON or Minify JSON feature, such as <https://jsoneditoronline.org/>.

The compacted JSON code looks like this:

```
{"name": "es-us", "weekdays": ["domingo", "lunes", "martes", "miércoles", "jueves", "viernes", "sábado"], "weekdaysShort": ["dom.", "lun.", "mar.", "mié.", "jue.", "vie.", "sáb."], "weekdaysMin": ["do", "lu", "ma", "mi", "ju", "vi", "sá"], "months": ["enero", "febrero", "marzo", "abril", "mayo", "junio", "julio", "agosto", "septiembre", "octubre", "noviembre", "diciembre"], "monthsShort": ["ene", "feb", "mar", "abr", "may", "jun", "jul", "ago", "sep", "oct", "nov", "dic"], "relativeTime": {"future": "en %s", "past": "hace %s", "s": "unos segundos", "m": "un minuto", "mm": "%d minutos", "h": "una hora", "hh": "%d horas", "d": "un día", "dd": "%d días", "M": "un mes", "MM": "%d meses", "y": "un año", "yy": "%d años"}, "formats": {"LT": "h:mm A", "LTS": "h:mm:ss A", "L": "MM/DD/YYYY", "LL": "D [de] MMMM [de] YYYY", "LLL": "D [de] MMMM [de] YYYY h:mm A", "LLLL": "dddd, D [de] MMMM [de] YYYY h:mm A"}}
```

(A few line breaks have been added to wrap the text in this Guide.)



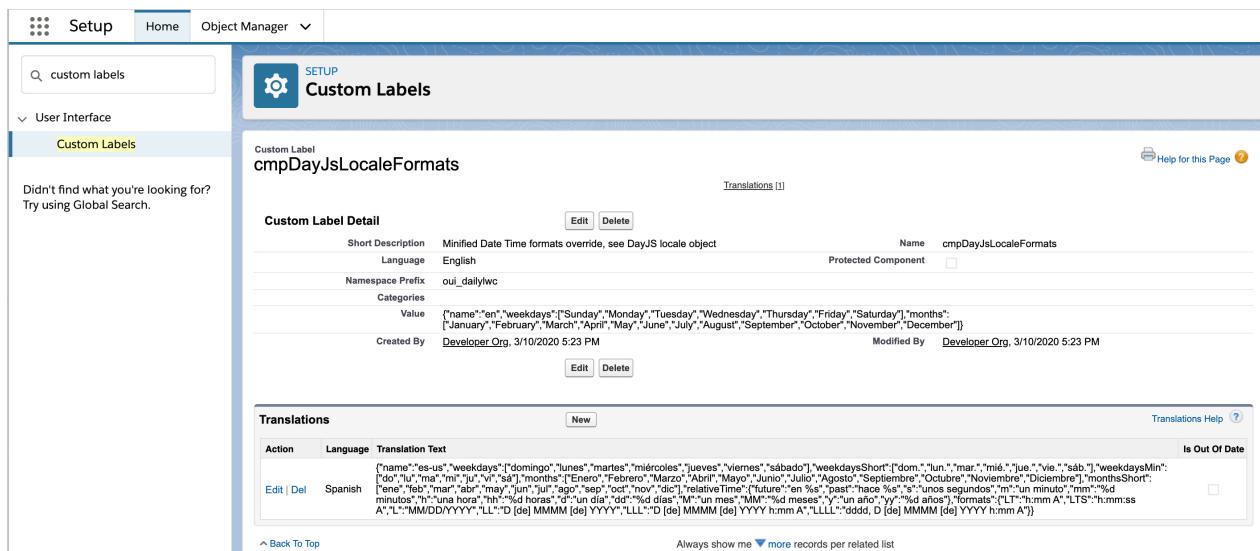
Note Custom labels accept valid JSON strings only.

- Copy the compacted JSON code.
- In Setup, go to **Custom Labels**, and open a custom label that meets your requirements.

The following custom labels are recommended:

Custom Label	Description
cmpDayJsLocaleFormats	Applies translations to LWC date components and Omniscript date elements. The cmpDayJsLocaleFormats label doesn't apply translations to Omniscripts if the OmniDayJSLocaleFormats label has translations defined.
OmniDayJsLocaleFormats	Applies translations to Omniscript date and time elements only. Overrides the cmpDayJsLocaleFormats label in Omniscripts if both custom labels have translations defined.

7. In the custom label, click **New** to add a translation.
8. Select the **Language**.
9. In the **Translation** field, paste in the compacted JSON code.
10. Click **Save**.



11. Open an Omniscript that includes a Date or Date/Time element, go to Preview, select the language, and open the date picker.
If you've followed the example and selected Spanish, the date picker displays Spanish translations for the months and days of the week.

Create Multi-Language Select Elements

Populate Select, Multi-select, and Radio elements for a multi-language Omniscript using custom configuration options.

Configure Options Manually

In Salesforce, define a set of custom labels and configure translations for them. In Omniscript designer, use the custom labels to specify the options for the Select elements.

1. In Salesforce, define a set of custom labels and specify the translations for each one.
2. In the Omniscript designer, add a Select element configured as follows:
 - Option Source: Manual
 - Options:
 - Value: The value to be written to the data JSON if this option is selected.
 - Label: The name of the Salesforce custom label from which the text for the option is read.

 **Example** For example, for a list of animals, three custom labels have translations in Salesforce.

Custom Label	Default Value	Spanish Translation
Animal_Bear	Bear	Oso
Animal_Dog	Dog	Perro
Animal_Cat	Cat	Gato

In Omniscript Designer, specify the Options for the Select element.

Value	Label
1	Animal_Bear
2	Animal_Dog
3	Animal_Cat

Populate the Options Programmatically

Create an Apex method that returns a set of options that are translated according to the language code in effect when the method is called.

For dependent Select elements that are populated programmatically (custom), configure a controlling field by specifying the name of an Omniscript element. The element has the value that determines the options in the dependent Select element. For example, a list of cities depends on a state specified in another element. For the dependent element, specify its CONTROLLING FIELD settings as follows:

- Option Source: Custom
- Source: Omit
- Element: The Omniscript element containing the value that determines how this dependent element is populated.

The Apex method must have logic that uses the value of the controlling element and the language code

to determine the values to return and populate the dependent element.

1. Create a class and method that returns a map of options translated according to the language code in effect when the method is called. See the sample code.
2. Add a Select element to your Omniscript, configured as follows:
 - Option Source: Custom
 - Source: The class and method that return the options



Example

```
global class PicklistPopulation implements VlocityOpenInterface { public Boolean invokeMethod(String methodName, Map<String, Object> input, Map<String, Object> outMap, Map<String, Object> options) { try { if (methodName.equals('PopulatePicklist')) { PopulatePicklist(input, outMap, options); } else if (methodName.equals('PopulateDependentPicklist')) { PopulateDependent Picklist(input, outMap, options); } } catch (Exception e) { System.debug(LoggingLevel.ERROR, 'Exception: ' + e.getMessage() + ' ' + e.getStackTraceString()); } return true; } // Get All Contacts Associated with the Context Id Account where the Omniscript is Launched public void PopulatePicklist(Map<String, Object> input, Map<String, Object> outMap, Map<String, Object> options) { List<Map<String, String>> plOptions = new List<Map<String, String>>(); String lang = (String)input.get('LanguageCode'); for(Integer i=0; i<2; i++) { Map<String, String> tempMap = new Map<String, String>(); tempMap.put('name', 'test'+ String.valueOf(i)); // Language Independent tempMap.put('value', 'testV'+ String.valueOf(i)); // Displayed in Picklist UI if(lang == 'zh_CN') tempMap.put('value', 'testV' + String.valueOf(i) + ' 中文'); plOptions.add(tempMap); } outMap.put('options',plOptions); } // Populate a Dependent Picklist based on the Contacts ReportsToId Field Selected in the Other Field this field depends on public void PopulateDependentPicklist(Map<String, Object> input, Map<String, Object> outMap, Map<String, Object> options) { // Map of List where the Key is the Potential Values in the Other Picklist Map<String, List<Map<String, String>>> dependency = new Map<String, List<Map<String, String>>>(); String lang = (String)input.get('LanguageCode'); String elementName = (String)input.get('controllingElement'); // itself Logger.debug('hello1 ' + lang); Logger.debug('hello2 ' + elementName); List<String> controlValList = new List<String>(); controlValList.add('Licensing & Permitting'); controlValList.add('Contract'); for(Integer i=0; i<2; i++) { List<Map<String, String>> optionList = new List<Map<String, String>>(); for(Integer j=0; j<2; j++) { Map<String, String> tempMap = new Map<String, String>(); tempMap.put('name', controlValList[i] + 'Child' + String.valueOf(j)); // Language Independent tempMap.put('value', controlValList[i] + 'ChildV' + String.valueOf(j)); // Displayed in Picklist UI if(lang == 'zh_CN') tempMap.put('value', controlValList[i] + 'ChildV' + String.valueOf(j) + ' 中文' ); optionList.add(tempMap); } dependency.put(elementName,optionList); } }
```

```
t(controlValList[i], optionList); } outMap.put('dependency', dependency); }
```

Use a Translated Salesforce Picklist

In Salesforce, create a picklist and use Translation Workbench to define translations for it. In Omniscript designer, define a Select element that is bound to the Salesforce picklist.

1. In the Omniscript Designer, add a Select element and set these properties:

- **Options Source:** sObject
- **Source:** <sObjectName>.<fieldName>

For example, if you define a custom object named `Multi_Language_Picklist` and define a picklist field named `Animal_Type`, the syntax is `Multi_Language_Picklist__c.Animal_Type__c`.

2. In Setup, under Administration Setup, choose **Translation Workbench**.
3. On the **Translate** screen, choose the target language.
4. From the **Setup Component** list, choose **Picklist Value**.
5. From the **Object** list, choose the object where the picklist field is defined. The picklist and its values are displayed as a tree.
6. For each option, specify the translated value, then save your changes.

Omniscript Custom Label Reference

Review the Omniscript element properties for which you must define custom labels and translations for multi-language Omniscripts. To use a default translation, clear the property.

 **Note** If you are converting an Omniscript from single- to multi-language, specify valid custom label names or clear all translatable properties. If there is literal text in the property and no custom label exists, a **Missing label** error is displayed when you attempt to preview the script.

Required Translations for Omniscripts

This table lists the properties for which Omnistudio provides custom labels with default translations.

Element	Properties
All elements	label

Element	Properties
Calculation Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Checkbox	checkLabel
Omnistudio Data Mapper Extract Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Data Mapper Post Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Data Mapper Transform Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Disclosure	checkLabel
DocuSign Envelope Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
DocuSign Signature Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Edit Block	newLabel, editLabel, deleteLabel
Email Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Integration Procedure Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage

Element	Properties
Matrix Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Multi-Select	options[i].value
PDF Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Radio	options[i].value
Remote Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Rest Action	errorMessage.custom[n].message, errorMessage.default, failureNextLabel, failureGoBackLabel, inProgressMessage
Script Header	consoleTabLabel
Select	options[i].value
Step	cancelLabel, nextLabel, previousLabel, saveLabel, cancelMessage, saveMessage
Type Ahead Block	newItemLabel
Validation	messages[i].text

Omnistudio provides some standard labels that you can use as the default in your translations. You can find these labels by searching for *Omni* in the custom labels of your org.

Property	Default Custom Label
cancelLabel	OmnicancelLabel

Property	Default Custom Label
cancelMessage	OmnicancelMessage
consoleTabLabel	OmniconsoleTabLabel
editLabel	OmnieditLabel
failureGoBackLabel	OmnifailureGoBackLabel
failureNextLabel	OmnifailureNextLabel
inProgressMessage	OmniinProgressMessage
newItemLabel	OmninewItemLabel
newLabel	OmninewLabel
nextLabel	OmninextLabel
omniRequired	OmniRequired
postMessage	OmnipostMessage
previousLabel	OmnipreviousLabel
remoteConfirmMsg	OmniremoteConfirmMsg
saveLabel	OmnisaveLabel
saveMessage	OmnisaveMessage

Test a Multi-Language Omniscript

View custom translations using the Omniscript Designer preview.

1. Click **Preview**. If the Omniscript contains custom Lightning web components, activate the Omniscript before previewing.
2. From preview, select a language.

3. View the custom label translations for every Omniscript element to ensure they display correctly.
4. Repeat the testing process for every translated language.

Launch a Multi-Language Omniscript

Set the language a multi-language Omniscript renders in by using a language code parameter in a URL.

1. From the Omniscript, click **How to launch?**.
2. Copy the URL you want to use to launch your Omniscript.
For information on Omniscript URLs, see [How to Launch Omniscripts](#).
3. Determine which language code to use in Salesforce.
See [Supported Languages](#).
4. Edit the URL to add the language parameter and set the parameter equal to the Salesforce language code.

Example URL for a lightning URL in Omnistudio standard runtime:

```
https://exampleURL.force.com/AccountCommunity/s/AccountPage?c__target=c:docL  
WCTestEnglish&c__layout=lightning&c__LanguageCode=es
```

Build an Omniscript with Elements

For guided workflows, Omniscripts use basic tasks: Get, save, modify, and display data from Salesforce objects and other data sources. They can also allow users to add, change, or remove data and can take actions, such as getting a DocuSign signature, sending an email, or opening a web page.

You can add some elements to the Omniscript canvas, as shown in this diagram. Action, Step, Omniscript, and Action Block elements can go directly into an Omniscript. To add any other element, you must first add a Step or Action Block element. For an Action Block, you can add only four action elements. Some actions can't be added to action blocks: DocuSign Signature, Navigate, PDF, Set Errors, and Set Values. With a Step, you can add any element except an Omniscript.



This Omniscript example guides users to update the primary contact for an account. It uses elements for all the basic tasks and shows where the elements are added. First, an Integration Procedure element pulls the data from an Integration Procedure that uses a Data Mapper Extract to retrieve data from the Account object.



In the first step, read-only text elements show the name, email, and phone number of the current main contact. This information comes from the Integration Procedure element. Then, users select from a radio list to add or update the primary contact. In the second step, users enter or change the contact information. After a user clicks next, an action element sets the values from the user input to JSON.

Then, an Integration Procedure element with a Data Mapper Post saves the updated contact data to the Account object. Finally, a Navigate element opens the Account record page so the user can see the applied changes.

Load Data into Omniscript Elements

You'll likely need to pull data into an Omniscript to prefill elements, create selectable options, dynamically display elements, or test your Omniscript. You can use data from Salesforce objects or external and internal sources. Omniscripts use several ways to get this data: Omnistudio Data Mappers, Integration Procedures, REST APIs, or Apex. Users can also add or change data in an Omniscript.

Use and Modify Data in Omniscripts

After an Omniscript retrieves data, it can then work with and transform, rename, or convert the data. You can use a Data Mapper Transform element, an Integration Procedure element, or a formula element. For example, if you have user input and data from an external source, you might need to combine data, remove duplicates, and put the data into a consistent format.

Create and Update Data from Omniscripts

After you've gotten new or changed data from other sources or users, you'll likely need to update records in Salesforce or other data sources. As with getting data, Omniscripts can create or update records with various methods: Omnistudio Data Mappers, REST APIs, or Apex. You can also write data to a fillable PDF.

Take Actions with Omniscripts

Besides getting, modifying, and writing data, Omniscripts can take other types of actions. Omniscripts can send messages to users or systems. Users can also provide input to the Omniscript or go to another page or site from an Omniscript. Users can also use DocuSign to sign documents electronically. You can automate decisions in Omniscripts with the Business Rules Engine. You can also extend Omniscript elements for custom behaviors.

Omniscript Best Practices

Enhance the performance and usability of Omniscripts by following the recommended Omniscript best practices.

Load Data into Omniscript Elements

You'll likely need to pull data into an Omniscript to prefill elements, create selectable options, dynamically display elements, or test your Omniscript. You can use data from Salesforce objects or external and internal sources. Omniscripts use several ways to get this data: Omnistudio Data Mappers, Integration Procedures, REST APIs, or Apex. Users can also add or change data in an Omniscript.

For each of these methods, you'll use Omniscript elements.

- To get data from Salesforce objects, you can use a Data Mapper Extract or Turbo Action element or an Integration Procedure Action element.
- If you need data from an internal or external source or to collect data from multiple sources, you can also use an Integration Procedure Action element.
- Or you can directly connect with the HTTP (to use REST APIs) and Remote (to use Apex classes and

methods) action elements.

You can think of user input as a method to get data. Omniscripts have many types of input elements, including images, choices, URLs, phone numbers, signatures, time, and currency.

To prefill an Omniscript with data, see [Seed Data Into an Omniscript](#)



[Load Salesforce Data into an Omniscript Using an Omnistudio Data Mapper](#)

Add Salesforce data to your Omniscript using a Data Mapper Turbo Action or Data Mapper Extract Action.

[Prefill Repeatable Blocks](#)

Prefill repeatable blocks with data by passing data to the block in an array format.

[Set a Default Value for an Omniscript Element](#)

The Default Value property sets the JSON value of an element to a static value by default. The default value is overwritten if the element is prefilled from an Omnistudio Data Mapper or JSON response.

The Default Value property is available for the Input elements (except Custom LWC, Disclosure, File, Image, and Password).

Load Salesforce Data into an Omniscript Using an Omnistudio Data Mapper

Add Salesforce data to your Omniscript using a Data Mapper Turbo Action or Data Mapper Extract Action.

1. To get data from Salesforce into Omniscript elements, create a Data Mapper Turbo Extract or Data Mapper Extract. See [Omnistudio Data Mappers](#).
2. Make sure the inputs from the Omniscript match the filter values on the Data Mapper's **Extract** tab. For example, if the Data Mapper has the filter `Name LIKE Key`, make sure the Omniscript's Data Mapper Turbo Action or Data Mapper Extract Action has an Input Parameter with a **Filter Value** of `Key`.
3. For a Data Mapper Turbo Extract, make sure the **Response JSON Path** in the Omniscript's Data Mapper Turbo Action matches the Data Mapper Turbo Extract's **Extract Output Path**.
4. Make sure the Data Mapper's output field names match the Name values of Omniscript elements that you want to populate with the data.
For a Data Mapper Turbo Extract, this direct matching only works for the primary object fields, and you can't change the field names. For a Data Mapper Extract, you can name the **Output JSON Path** fields so they match the Name values of Omniscript elements.
For example, if a Data Mapper's output field is named `LastName`, make sure the Text input element in the Omniscript that it populates is also named `LastName`.
5. To retrieve related object fields from a Data Mapper Turbo Extract, use the Omniscript element's Default Value property.
For example, if the Data Mapper Turbo Action is in a Type Ahead block, change the Default Value to something like `%Step1:TypeAhead1-Block:Account:Name%`.

-  **Note** To avoid overwriting JSON nodes for related object fields, make sure that the output JSON nodes of Data Mapper Extract and Data Mapper Turbo Extract and the Name values of Omniscript elements do not match.

To make Data Mapper Extract mapping easier, run the Omniscript in Preview, and in the Step with the elements to be populated, copy the **Data JSON** that corresponds to those elements. You can paste this Data JSON into the **Expected JSON Output** pane on the Data Mapper's **Output** tab, which populates the **Output JSON Path** fields.

Prefill Repeatable Blocks

Prefill repeatable blocks with data by passing data to the block in an array format.

The data's root node name must match the name of the block element. The fields within each array instance of the data JSON node must match the name of an element within the block. Prefill data into repeatable blocks with Actions by returning data that matches the block's element name and the element's within the block using an array format.

Data JSON array format example:

```
"EditBlock1": [
    {
        "FirstName": "John",
        "LastName": "Smith"
    },
    {
        "FirstName": "Jane",
        "LastName": "Smith"
    }
]
```

To map Data JSON to a Repeatable Block with a Set Values action:

1. Add a Block or Edit Block to the Omniscript.
 2. (Optional) When using a Block, check the Block's **Repeat** checkbox property.
-  **Note** Edit Blocks are inherently repeatable because they store entry values in an array format.
3. Add Input elements to the Block.
 4. Add a **Set Values Action** to the Omniscript.
 5. In the Set Values Action, click **Add New Value**.
 6. In the **Element Name** field, enter the name of your block element.
 7. Click **Edit as JSON** in the Set Values Action element to edit the JSON directly.
 8. In the **elementValueMap** JSON node, locate the block element's JSON node, and enter an array of data.

For example, to prefill two Text input elements in an Edit Block element, add an array that maps to the elements in the block.

The screenshot shows the Omniscript Structure panel. A section named "Step3" is expanded, revealing an "EditBlock1" element. Inside "EditBlock1", there are two fields: "FirstName" and "LastName", both of which have a checked checkbox next to them, indicating they are selected for configuration.

The edit block example has a JSON array.

```
"EditBlock1": [
  {
    "FirstName": "John",
    "LastName": "Smith"
  },
  {
    "FirstName": "Jane",
    "LastName": "Smith"
  }
]
```

- To convert back to the default view of the Set Values Action, click **Edit in Property Editor**.

The screenshot shows the Element Value Map in the Property Editor. An entry for "EditBlock1" is listed under "Element Name". The "Type" is "Edit Block". The "Value" field contains the JSON array from the previous code block: "[{"FirstName": "John", "LastName": "Smith"}, {"FirstName": "Jane", "LastName": "Smith"}]". There is also a "fx" icon and a "New" button next to the value field.

- Preview the Omniscript to view the nested data in the Omniscript's data JSON.

The screenshot shows the Omniscript preview interface. It displays a table with two rows. The first row has "FIRST NAME" as "John" and "LAST NAME" as "Smith". The second row has "FIRST NAME" as "Jane" and "LAST NAME" as "Smith". Each cell in the table has a dropdown arrow icon to its right. At the bottom of the table, there is a "+ New" button. Below the table, there are "Cancel" and "Save for later" buttons.

Set a Default Value for an Omniscript Element

The Default Value property sets the JSON value of an element to a static value by default. The default value is overwritten if the element is prefilled from an Omnistudio Data Mapper or JSON response. The

Default Value property is available for the Input elements (except Custom LWC, Disclosure, File, Image, and Password).

For more information on how to dynamically prefill values, see [Set Values in an Omniscript](#).

Default Value fields in LWC Omniscripts accept merge field syntax. For more information, see [Access Omniscript Data JSON with Merge Fields](#).

1. From the element's Properties panel, click the **Default Value** property.
2. Enter a literal value or a merge field.

For a merge field, the value in the referenced field must already be present when the element is instantiated. The default value is set only when the element is instantiated for the first time.

For elements such as Range and Number, the Default Value property only lets you enter a number. To enter a merge field, switch to the JSON editor.

For Select, Multi-Select, and Range elements, select a default value while adding an option for users to select.

The date picker lets the user choose a date that is +/- 100 years from the **Default Value** of a Date element.

For example, to set the default value of a Date element to the value of a JSON node called `EffectiveDate`, enter the merge field `%EffectiveDate%` in the **Default Value** property.

Use and Modify Data in Omniscripts

After an Omniscript retrieves data, it can then work with and transform, rename, or convert the data. You can use a Data Mapper Transform element, an Integration Procedure element, or a formula element. For example, if you have user input and data from an external source, you might need to combine data, remove duplicates, and put the data into a consistent format.

Use Omniscript formulas and functions to manipulate data directly in an Omniscript. They're executed in real-time as a user interacts with an Omniscript, providing immediate feedback and results to users. Use them for simpler tasks, such as logical operations to make decisions based on data or arithmetic operations.

For more complex tasks, such as converting, restructuring, or renaming data, use a Data Mapper Transform. In cases where you need to modify or transform data from various sources or where the transformations are more complex, such as conditions or filtering, you might use an Integration Procedure with an Omniscript.

[Access Omniscript Data JSON with Merge Fields](#)

Enable Omniscript elements and element properties to access data JSON using merge fields. A merge field is a variable that references the value of a JSON node. For example, if a JSON node is `"FirstName": "John"`, then the merge field `%FirstName%` returns `John`.

Manipulate JSON with the Send/Response Transformations Properties

Specify and transform the input and output JSON of Omniscripts or Integration Procedures with the Send and Response Transformation properties. The Send properties transform the input; the Response properties transform the output.

Access to Data Within and Outside a Repeatable Block

Suppose you place an element in a repeatable block (a Block or an Edit Block) and you want to access data that is calculated in a different element. The method that you use to access the data depends on whether the element containing the data is within or outside the same repeatable block in the Omniscript. Review guidelines and examples for the Formula element and for the Select element, which is used to add a dependent picklist to an Omniscript.

Access Omniscript Data JSON with Merge Fields

Enable Omniscript elements and element properties to access data JSON using merge fields. A merge field is a variable that references the value of a JSON node. For example, if a JSON node is `"FirstName": "John"`, then the merge field `%FirstName%` returns `John`.

Merge fields access data JSON using syntax to indicate to an Omniscript Element that a merge field is present. The syntax requires you to wrap a full JSON path with a percent sign on both ends.

-  **Note** Only certain element properties support merge fields. Text between two percent (%) signs in a Set Values formula or text field is treated as a merge field. To represent literal percent (%) signs, use the `$Vlocity.Percent` environment variable.

Common Use Cases for Accessing JSON with Merge Fields

Learn about common use cases for accessing Omniscript data JSON with merge fields.

- Setting Values to rename elements, access JSON nodes, run formulas, and populate elements. For more information, see [Set Values in an Omniscript](#)
- Access data stored in Omniscript elements in a formula or in a future step.
- Access data returned from an Action. For example, an HTTP Action or Omnistudio Data Mapper Action returns data from Salesforce or an external source. For more information on Actions, see [Omniscript Action Elements](#).
- Access data passed in as a parameter from a page.

Access the Data JSON

Use existing data JSON in an element property by indicating the use of a merge field.

1. Locate the name of the JSON node in the Omniscript's data JSON.

The screenshot shows the 'Data' tab of the Lightning Universal Page configuration. It includes fields for 'ContextId' and 'Fetch' (which is checked). Below these are sections for 'DATA' and 'Clear Data'. The 'DATA' section displays a JSON object with several fields, including 'firstName': "John", which is highlighted with a red box.

- Enter the name of the JSON node and wrap the name in percentage signs to indicate it is a merge field.

For example, a merge field accessing a JSON node named `firstName` must use the syntax `%firstName%`.

The screenshot shows the Omniscript editor interface. On the left, there's a 'Script Configuration' sidebar with sections for 'SetValues1', 'Step1' (containing 'Headline1'), 'Step2' (containing 'Headline2'), 'TextBlock2', and 'TextBlock1'. On the right, the main area shows a 'Headline1' component with an 'Element Name' of 'Headline1'. In the rich text editor, the text 'Welcome %firstName%' is displayed, with the placeholder '%firstName%' highlighted by a red box.

- Preview the Omniscript to ensure the Merge field works correctly.

The screenshot shows the Omniscript preview. The configuration on the left is identical to the editor. The preview on the right shows the rendered output: 'Welcome' followed by a horizontal line and 'Welcome, John!', where the merge field has been replaced by its value.

Additional Syntax

Check out additional syntax examples for nested JSON.

JSON Node	Merge Field Syntax Example
<pre>"ContactInfo": { "FirstName": "John"</pre>	<p>Use a colon symbol <code>:</code> to access a nested JSON node.</p>

JSON Node	Merge Field Syntax Example
}	%ContactInfo:FirstName%
<pre>"ParentObject": { "NumberMap": [1, 2, 3] }</pre>	<p>Use a colon symbol <code>:</code> to access nested JSON nodes and a pipe symbol <code> </code> to access the index of the array that value.</p> <pre>%ParentObject:NumberMap 3%</pre>
<pre>"ContactInfoStep": { "ContactInfoBlock": [{ "FirstName": "John" }, { "FirstName": "Adam" }, { "FirstName": "Steve" }] }</pre>	<p>When a formula exists within a repeatable block, use <code> n</code> to access the node in which the formula exists. Depending upon which node the formula exists in, it will return the correlating value. For more information, see Access to Data Within and Outside a Repeatable Block.</p> <pre>%ContactInfoStep:ContactInfoBlock n:FirstName%</pre>

Manipulate JSON with the Send/Response Transformations Properties

Specify and transform the input and output JSON of Omniscripts or Integration Procedures with the Send and Response Transformation properties. The Send properties transform the input; the Response properties transform the output.

The following table outlines the different transformation options:

Purpose	Send/Response JSON Path	Send/Response JSON Node	Example Path	Example Result
Reparenting the node	path	node	{path:{a:b}}	{node: {a:b}}

Purpose	Send/Response JSON Path	Send/Response JSON Node	Example Path	Example Result
Reparenting the node and adding a nested node	path	node1:node2	{path:{a:b}}	{node1:{node2:{a:b}}}
Reparenting the node dynamically using merge field syntax	path	%nodename% where the value of nodename is node	{path:{a:b}}	{node: {a:b}}
Removing the parent node of an object	path	VlocityNoRootNode (Not supported in Integration Procedures)	{path:{a:b}}	{a:b}
Specifying the root node to drill down on instead of sending the entire JSON	path		{path:{a:b}}	{path:{a:b}}

1. To reparent the node of a JSON object:
 - a. Set the **Path** to the node you are drilling down on.
 - b. Enter the new node name into the **JSON Node** field.
2. To reparent the node and add a nested node:
 - a. Set the **Path** to the node you are drilling down on.
 - b. Enter the new node name into the **JSON Node** field, and enter a colon (:).
 - c. After the colon, without leaving a space, enter the second JSON node name.
3. To reparent the node using merge field syntax:
 - a. Set the **Path** to the node you are drilling down on.
 - b. In the **JSON Node** field, enter a merge field that represents the new name, for example %NewNode%.
 - c. Use a SetValues action or an input parameter on the Preview tab to test the merge field.

For example, on the Preview tab, enter a **Key** of *NewNode* and a **Value** of **MightyNode**, click **Execute** and then look for **MightyNode** in the output JSON.
4. To write to a list item using merge field syntax:
 - a. In the **Response JSON Node** field of a step, enter the path to the list node followed by a pipe character (|) and a merge field that represents the list item number, for example,

ListNode | %item%.

- b. Use a SetValues action or an input parameter on the Preview tab to test the merge field.

For example, on the Preview tab, enter a **Key** of *item* and a **Value** of *2*, click **Execute**, and watch the step output appear as the second item in the list.

If the step output node and the existing list item node match, the list item value is replaced. If they don't match, the step output node is added as a peer of the existing list item node.

5. To remove the parent node of an object:
 - a. Set your **Path** to the node you are drilling down on.
 - b. Set the **Node** to *VlocityNoRootNode*.
6. To specify the root node:
 - a. Set the **Path** to the node you are drilling down on.
 - b. Leave the **JSON Node** field blank.
7. To test the result of your transformation, open the Debug console:
 - a. Preview the Omniscript in the Omniscript Designer and enter input that you need to transform.
 - b. After the action containing the JSON transformation has run, click **Debug** to open the Debug console.
 - c. In the Debug console, expand the action that is sending or receiving the transformed JSON.
 - d. Expand the **Request** to see JSON that has been transformed by the **Send JSON Path** and **Send JSON Node**, or expand the **Response** to see JSON that has been transformed by the **Response JSON Path** and **Response JSON Node**.

Access to Data Within and Outside a Repeatable Block

Suppose you place an element in a repeatable block (a Block or an Edit Block) and you want to access data that is calculated in a different element. The method that you use to access the data depends on whether the element containing the data is within or outside the same repeatable block in the Omniscript. Review guidelines and examples for the Formula element and for the Select element, which is used to add a dependent picklist to an Omniscript.

Guidelines for Accessing Formula Element Data in Repeatable Blocks

When you use a Formula element within a repeatable block, review guidelines and examples for accessing data within or outside the same repeatable block.

Guidelines for Accessing Select Element Data in Repeatable Blocks

When you use a Select element within a repeatable block, review guidelines and examples for accessing data within or outside the same repeatable block. The Select element is used with Salesforce dependent picklists.

See Also

- [Combine Elements Logically in a Block](#)
- [Manage Records with an Edit Block](#)
- [Using Salesforce Picklists with Omniscript Inputs](#)

Guidelines for Accessing Formula Element Data in Repeatable Blocks

When you use a Formula element within a repeatable block, review guidelines and examples for accessing data within or outside the same repeatable block.

Guidelines

When using Formula elements inside multi-level repeat blocks, if the Formula elements are not placed in the top-level block, it appears as null or false. This can cause issues when trying to use them on the page. In the Omniscript preview, you can verify this behavior by inspecting the formula elements in the JSON. These elements are either absent, or return null or false, regardless of the intended formula logic.

Within a repeatable block containing a Formula element, access the data in another element by referencing the element in the Formula element's Expression property. Use these guidelines:

Location of Data	How to Access the Data
Different element inside the same repeatable block	In the Expression property of the Formula element, enter <code>BlockName n:ElementName</code> . When the Omniscript runs, it accesses only the elements in the current block.
Different repeatable block	In the Expression property of the Formula element, enter <code>BlockName 1:ElementName</code> (where the number after <code>BlockName </code> is the block's count in the Omniscript).

 **Example 1: Formula Element That Accesses Data Within the Same Block** In this example, a Formula element references data from another element within the same repeatable Block element. The unlabeled block is repeatable (1).



In the properties for the unlabeled block, Enable Repeat is selected. (Edit Block elements are automatically repeatable.)

Build Properties Setup

BLOCK PROPERTIES ⓘ

Active

* Name ⓘ
blockUnlabeled

Field Label ⓘ

Collapse

Heading Level

REPEAT SETTINGS ⓘ

Enable Repeat ⓘ

Clone When Repeating ⓘ

Limit Repeat ⓘ

This screenshot shows the 'Properties' tab of the Omnistudio interface for a block element. The 'BLOCK PROPERTIES' section includes fields for 'Name' (set to 'blockUnlabeled'), 'Field Label' (empty), and 'Collapse' (unchecked). The 'REPEAT SETTINGS' section has 'Enable Repeat' checked. A small orange box highlights the 'Enable Repeat' checkbox.

Properties for the Name element.

Build Properties Setup

TEXT PROPERTIES ⓘ

Active

* Name ⓘ
childName

Field Label ⓘ
Name

Placeholder ⓘ

Required ⓘ Read-only ⓘ

Default Value ⓘ

Mask ⓘ

Minimum Length ⓘ
0

Maximum Length ⓘ
255

Autocomplete ⓘ

REPEAT SETTINGS ⓘ

Enable Repeat ⓘ

This screenshot shows the 'Properties' tab of the Omnistudio interface for a text element. The 'TEXT PROPERTIES' section includes fields for 'Name' (set to 'childName'), 'Field Label' (set to 'Name'), and 'Placeholder' (empty). It also includes checkboxes for 'Required' and 'Read-only'. The 'REPEAT SETTINGS' section has 'Enable Repeat' unchecked.

Properties for the Birthday element.

DATE PROPERTIES

Name: childBirthday

Field Label: Birthday

Placeholder:

Required:

Read-only:

Default Value:

Minimum Date:

Maximum Date:

Date Format: MM-dd-yyyy

REPEAT SETTINGS

Enable Repeat:

Properties for the Age (Formula) element include the formula

AGE (%blockUnlabeled|n:childBirthday%) .

FORMULA PROPERTIES

Name: childAge

Field Label: Age

Data Type: Default

Expression: AGE(%blockUnlabeled|n:childBirthday%)

Hide:

Preview shows the repeating Formula element calculation as each child's age.

Summer Camp Enrollment

Child Information

Add Delete

Name	Birthday
Charlotte	07-14-2021
Age	
2	

Add Delete

Name	Birthday
Raghu	09-01-2020
Age	
3	

 **Example 2: Formula Element That Accesses Data in a Different Block** In this example, the Formula element Total references the data from the Age element, which is in a different repeatable block.

Step1
Step

Summer Camp Enrollment

Child Information

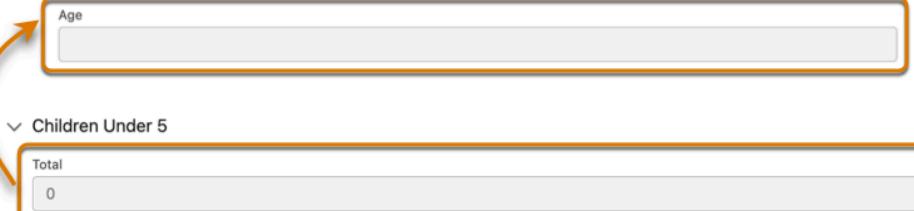
Add

Name	Birthday
Age	

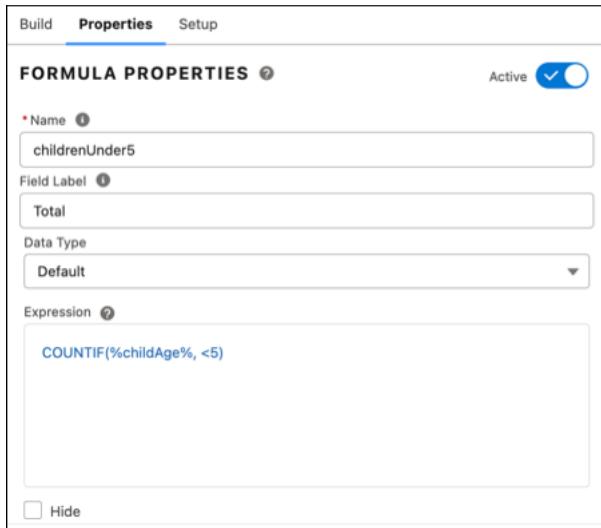
Children Under 5

Total

0



The properties for the block and elements within Child Information are the same as in Example 1. In the properties for the Formula element Total, the formula `COUNTIF(%childAge%, <5)` references the Age element in the previous block.



Preview shows the total entries that meet the criterion in the formula, by using data from the Age element.

The interface shows a list of children with their names, birthdays, and ages. Two orange arrows point to the age fields of the first two children (Lisa and Charlotte), which are 7 and 3 respectively. Below the list, a summary section titled 'Children Under 5' shows a single entry with a value of 1.

Name	Birthday
Lisa	07-01-2017
Charlotte	10-02-2020

Total
1

See Also

- [Combine Elements Logically in a Block](#)
- [Manage Records with an Edit Block](#)

Guidelines for Accessing Select Element Data in Repeatable Blocks

When you use a Select element within a repeatable block, review guidelines and examples for accessing data within or outside the same repeatable block. The Select element is used with Salesforce dependent picklists.

Guidelines

In a repeatable block containing a Select element tied to a Salesforce dependent picklist, you use the dependent picklist element to access controlling picklist selections.

Guidelines for the controlling picklist element properties:

Property	Configuration
Name	We recommend indicating in the name that this property is tied to the controlling picklist.
Field Label	The label that your users see.
Option Source	Select SObject.
Source	Indicate the picklist controlling field, separating the object and field names with a period.
Controlling Field Type	Leave as None.

Guidelines for the dependent picklist element properties:

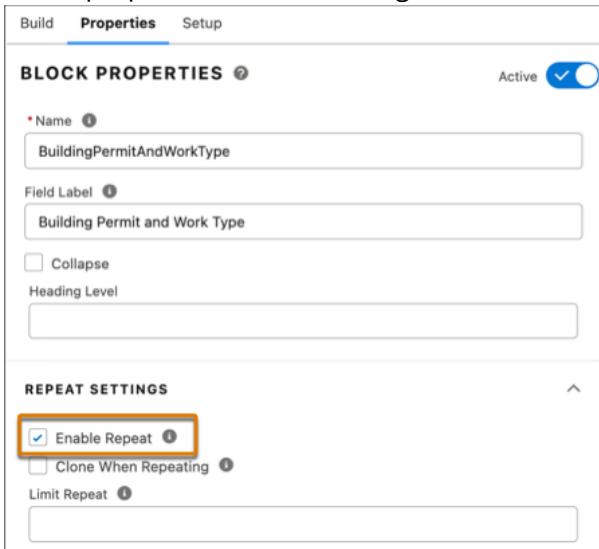
Property	Configuration
Name	We recommend indicating in the name that this property is tied to the controlling picklist.
Field Label	The label that your users see.
Option Source	Select SObject.
Source	Indicate the picklist dependent field, separating the object and field names with a period.
Controlling Field Type	Select SObject.
Controlling Field Source	Indicate the picklist controlling field, separating the object and field names with a period.
Controlling Field Element In the same repeatable block	If the controlling field element is within the same repeatable block, enter <code>BlockName n:ElementName</code> . Then, when the Omniscript runs, it accesses the named element in the current block.
Controlling Field Element In a different repeatable block	If the controlling field element (which references the controlling picklist) is in a different repeatable block, enter <code>BlockName 1:ControllingFieldElementName</code> .

Property	Configuration
	e (where the number after <code>BlockName </code> is the block's count in the Omniscript).

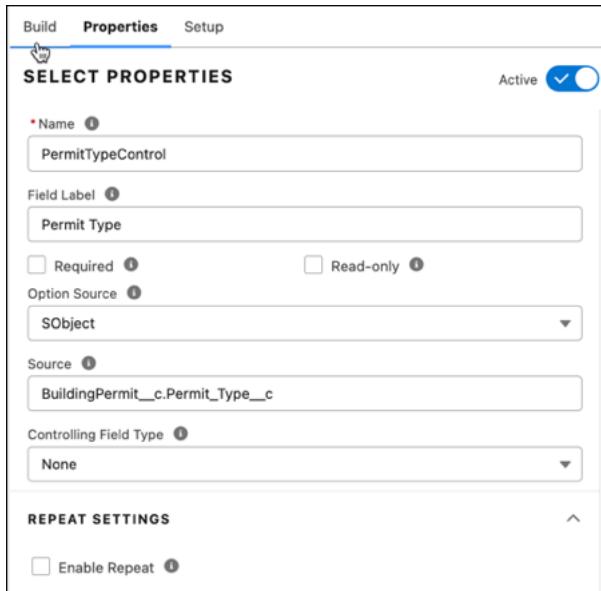
 **Example 3: Controlling and Dependent Picklists in the Same Block** In this example, the Building Permit and Work Type block contains Select elements for both the controlling picklist and the dependent picklist. The Building Permit and Work Type block is repeatable (1).



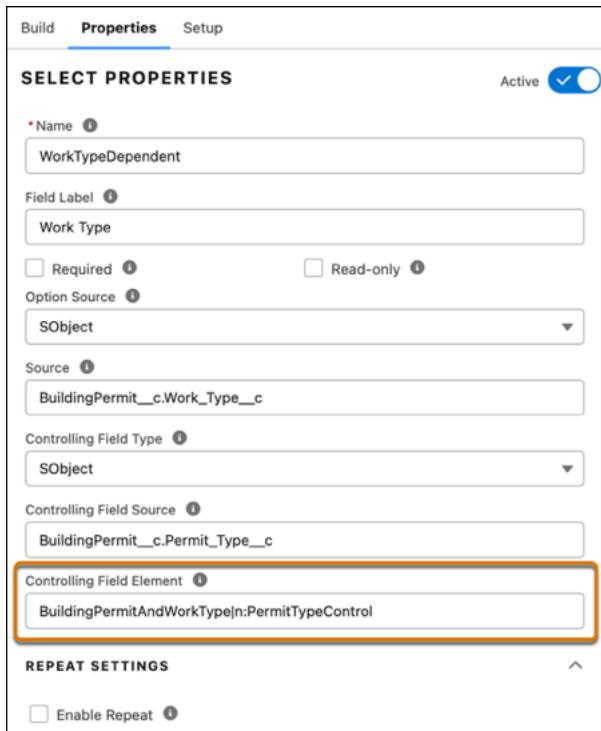
In the properties for the Building Permit and Work Type block, Enable Repeat is selected.



In the properties for the Select element for the controlling picklist, the Option Source is SObject, the Source is the object name and field, and the Controlling Field Type is None.



In the properties for the Select element for the dependent picklist, the Controlling Field Type is SObject instead of None. The Controlling Field Source is the object name and field. The Controlling Field Element property uses `|n` to reference the controlling picklist because the picklists are in the same block.



In Preview, the dependent picklist content depends on the user's selection in the controlling picklist.

Building Permit Application

Building Permit and Work Type

Permit Type: Commercial: Restaurant

Work Type:

- Clear --
- Plumbing
- Electrical
- Framing
- Waterproofing
- Fire
- Accessibility
- Final



Example 4: Controlling and Dependent Picklists in Different Select Elements In this example, the Select the Type of Work element contains the dependent picklist, which references the controlling picklist in the Select the Permit Type element. The Work Type block is repeatable (1).

Building Permit Application

Building Permit Type

Select the Permit Type

1

Work Type

Select the Type of Work

In the properties for the Work Type block, Enable Repeat is selected.

Properties

BLOCK PROPERTIES

Name: WorkType

Field Label: Work Type

Collapse

Heading Level

REPEAT SETTINGS

Enable Repeat

Clone When Repeating

Limit Repeat

Properties for the Select the Permit Type element. It's named PermitTypeControl to indicate that it contains the controlling picklist.

SELECT PROPERTIES

- Name: PermitTypeControl
- Field Label: Select the Permit Type
- Required:
- Read-only:
- Option Source: SObject
- Source: BuildingPermit__c.Permit_Type__c
- Controlling Field Type: None

REPEAT SETTINGS

- Enable Repeat:

Properties for the Select the Type of Work element. It's named WorkTypeDependent to indicate that it contains the dependent picklist. The Controlling Field Type is SObject instead of None. The Controlling Field Source is the object name and field. The Controlling Field Element property references the controlling picklist element by specifying its location in the first block, Building Permit Type.

SELECT PROPERTIES

- Name: WorkTypeDependent
- Field Label: Select the Type of Work
- Required:
- Read-only:
- Option Source: SObject
- Source: BuildingPermit__c.Work_Type__c
- Controlling Field Type: SObject
- Controlling Field Source: BuildingPermit__c.Permit_Type__c
- Controlling Field Element: BuildingPermitType|1:PermitTypeControl

REPEAT SETTINGS

- Enable Repeat:

Preview shows the controlling picklist selections.

Building Permit Application

Building Permit Type

Select the Permit Type

-- Clear --
Commercial: Retail
Commercial: Restaurant
Commercial: Light Industrial
Residential: Single-Family Home
Residential: ADU
Residential: Multifamily
Residential: Garage or Shed

The dependent picklist content depends on the user's selection in the controlling picklist.

Building Permit Application

Building Permit Type

Select the Permit Type

Commercial: Restaurant

Work Type

Select the Type of Work

-- Clear --
Plumbing
Electrical
Framing
Waterproofing
Fire
Accessibility
Final

See Also

- [Combine Elements Logically in a Block](#)
- [Manage Records with an Edit Block](#)
- [Dependent Picklists](#)
- [Using Salesforce Picklists with Omniscript Inputs](#)

Create and Update Data from Omniscripts

After you've gotten new or changed data from other sources or users, you'll likely need to update records in Salesforce or other data sources. As with getting data, Omniscripts can create or update records with various methods: Omnistudio Data Mappers, REST APIs, or Apex. You can also write data to a fillable PDF.



For each of these methods, you'll use Omniscript elements.

- To write data to Salesforce objects, use a Data Mapper Post Action.
- To write to an internal or external source, you can directly connect with the HTTP (for REST APIs) and Remote (for Apex classes and methods) actions.
- To update an uploaded PDF form, use a PDF action element with a Data Mapper Transform to map the data to the PDF fields.

Take Actions with Omniscripts

Besides getting, modifying, and writing data, Omniscripts can take other types of actions. Omniscripts can send messages to users or systems. Users can also provide input to the Omniscript or go to another page or site from an Omniscript. Users can also use DocuSign to sign documents electronically. You can automate decisions in Omniscripts with the Business Rules Engine. You can also extend Omniscript elements for custom behaviors.



To give users help messages, you can add tooltips to elements or set an error message for an element, such as a step. Or use the [Set Error](#) element when required information isn't provided or a condition in a step isn't provided. To communicate between an Omniscript and windows, Lightning web components, or other Omniscript elements, you can set up the [messaging framework](#).

A common type of Omniscript action is to go to another Omniscript, site, Knowledge article, Experience Cloud page, or Salesforce record or object with the [Navigate](#) element. Users can select to go to these and other pages, or you can have them happen automatically. You can use these for different purposes, such as confirming a record change or addition.

When configured with DocuSign, Omniscripts can open a PDF for a user's signature and then send a signed document with the DocuSign Signature and DocuSign Envelope elements.

To check customer eligibility or automate decision making, you can use the Expression Set and Decision Matrix elements to use those existing components in the Business Rules Engine.

If the standard Omniscript actions don't cover your needs, you can [extend an element](#) for custom behaviors.

Omniscript Best Practices

Enhance the performance and usability of Omniscripts by following the recommended Omniscript best practices.

Business Process and Logic

Business process and logic best practices include:

- Use unique names for Omniscript elements and Omnistudio Data Mapper response nodes.
- Identify reusable elements by building an outline of the entire Omniscript.
- Document the purpose of an element in the element's Internal Notes property.
- Avoid element name changes after deploying to production. When unavoidable, apply the name changes wherever the element is referenced.
- Don't assign a ContextId within the Omniscript. An Omniscript's ContextId is a reserved key that assigns a Record ID from the URL.
- When processes are repeatable across multiple Omniscripts, create a reusable Omniscript, and add it to the appropriate parent Omniscripts. See [Embed an Omniscript in Another Omniscript](#).
- Try to enforce required data and validations at the current step instead of at the end of a form. This helps users correct their inputs immediately instead of looking at validations or errors at the end and retracing their steps to fix each one.
- Where possible, break down complex tasks in an application by using multiple Omniscripts. Use the Omniscripts with Flexcards, if needed. Creating a large Omniscript with several steps can hinder performance and user experience.

User Interface

Omniscripts use Lightning Web Components to define the styling for both individual elements and the Omniscript itself.

Best practices include:

- Apply global styling. For more information, see [Custom Styles for Omniscripts](#).

Accessibility

Accessibility best practices include:

- Currently, edit blocks configured in table mode in Omniscripts aren't compatible with screen readers. If you use a screen reader, we recommend that you use Flexcards instead of Omniscripts. In Flexcards, you can use data tables or Lightning datatables that are compatible with screen readers. See [Show Data in a Table on a Flexcard](#) and [Lightning Web Component – Datatable](#).

Security

Data security best practices include:

- To ensure data security and maintain compliance with Salesforce encryption access controls, always check that a user has the **View Encrypted Data** permission before displaying or processing decrypted values of encrypted fields.

User Experience Design Principles

UX design principles include:

- Reduce the number of fields the user must input information into by prefilling the fields using contextual data. For more information, see [Populate Input Elements with Set Values Action](#).
- To avoid cognitive load, carefully consider how you break up processes. Too many short steps that contain only a minimal number of elements can overwhelm the user as easily as large steps with several elements.
- Guide the user by creating contextual help text and logically ordering input fields.
- Ensure that the recursive nested blocks within an Omniscript do not exceed 10.

Performance Factors

A set of best practices exists for both Client-side performance and Server-side performance.

Client-side best practices include:

- Reduce Conditional Views, Merge Fields, Formulas where possible.
- Speed up the application of responses by trimming the Response JSON. For more information, see [Manipulate JSON with the Send/Response Transformations Properties](#).
- Remove spaces from element names to improve the Omniscript's load time.
- Reduce the number of elements in the script. Try to limit a single Omniscript to a maximum of 750 elements.
- Run logic on the server where possible, for example, conditional logic in Integration Procedures and formulas in Data Mappers.
- PDFs larger than 250 KB generate slowly. PDFs larger than 1 MB can take several minutes to generate and sometimes time out.

Server-side best practices include:

- Cut down the payload size of a request by trimming the JSON request. For more information, see [Manipulate JSON with the Send/Response Transformations Properties](#).
- Don't create Omniscripts with Lightning web components larger than 4 MB. In preview, Omniscripts that are larger than 4 MB show an error to indicate that the Omniscript is either inactive or isn't deployed. Download the Omniscript LWC to check its size.
- Reduce server roundtrips by using Integration Procedures whenever there are multiple actions between steps. Run Integration Procedures asynchronously by enabling the fire and forget property.
- Remove unnecessary data by trimming the Data Mapper extract output.
- When building features with Omnistudio, keep the Apex governor limits in mind. These limits ensure that a single process doesn't control shared resources. During development, use debug logs to track and manage governor limits, ensuring your development is efficient and doesn't exceed Salesforce limits during execution. See [Apex Governor Limits](#).

To monitor how governor limits are used by Omnistudio components during execution, collect Apex debug logs for a transaction, for example, activating an Omniscript, launching a Flexcard, or executing a step in an Omniscript. To collect Apex debug logs, create a trace flag and set it for the user executing Omnistudio processes. See [Set Up Debug Logging](#).

When you run an Omniscript, Integration Procedure, or Data Mapper, the debug logs capture all transactions and provide insights into the governor limits usage.

Implementation Use Cases

- Save selected records from Flexcard Datatable to an Omniscript.

1. Create a Flexcard and add a Datatable element.

 **Note** Ensure you have created your Flexcard and added the necessary data table component within the Flexcard.

2. Configure Event Listener in the Flexcard setup:

a. Go to the Flexcard's setup tab.

b. Add an Event Listener with these settings:

- Event Type: Custom Event
- Event Name: selectrow
- Action Label: Choose a label that you prefer.
- Action Type: Update Omniscript
- Parent Node: OsDataNode

This configuration ensures that you are updating a specific node in an array in the Omniscript.

This helps you track and update the selected records. You can replace records with any array node name as required.

 **Note** Ensure that you update the correct node in the Omniscript for accurate data handling.

3. Add Input parameters with these settings:

- Key: Choose the appropriate JSON tag for the selected record.
- Value: {action.result}

This configuration ensures that the entire data row is included, along with the selectrow variable, which indicates if the record is selected or not (true or false).

 **Note** The input parameters should match your Omniscript variable names to ensure data is transferred correctly.

4. If required, add conditions. For example, use action.result.selectrow to determine if the record should be added or removed.

This condition is optional. If you need different actions for selected and unselected records, you can create two event listeners with different conditions for each case. You can customize the logic further by adding more conditions based on the requirements of your workflow.

Configuration

Designing Omniscripts with specific configurations may sometimes lead to unintended behaviors on the sites they're embedding in. This can occur due to the way Omniscripts are designed or due to a combination of factors outside of the scope of Omnistudio. Some known issues and related best practices are described here.

Configuration best practices include:

- When using an Action, Set Values, or Data Mapper element to populate data in an Edit Block within an Omniscript, data duplication may occur if the **Merge saved data JSON to the updated version** checkbox is selected in the Save Options section, and if a user resumes a saved session. To avoid this,

add a flag to track the execution of the Action, Data Mapper, or Set Values element and include a condition in the Conditional View section to proceed only if the flag is false. For instance, you can use a flag with a Set Values element to prevent data duplication as shown.

The screenshot shows the Omniscript Designer interface. On the left, there's a list of steps: 'Step 1' (a Step element), 'SetValues1' (a Set Values element), and 'Step2' (another Step element). The 'SetValues1' step is currently selected. On the right, the 'Set Values Properties' panel is open, showing details like 'Field Label: SetValues1'. Below it, the 'Element Value Map' section contains a single entry: 'setvaluehasrun' with 'Value: true'. Further down, the 'Conditional View' section is expanded, showing a condition type 'Select an Option' and a specific condition: 'Show Element if True (setvaluehasrun <> true)'. Both the 'Element Value Map' and 'Conditional View' sections are highlighted with orange boxes.

- Tables aren't supported inside text blocks.

Configure Omniscript Settings

Configure optional settings for your Omniscript in the Setup panel of the Omniscript Designer, such as making your Omniscript reusable or enabling logging.

Default Setup Properties

Default setup properties apply to your entire Omniscript and aren't included in each element's properties.

Note After updating or configuring Omniscripts, allow 15–20 minutes for updates to appear, even after clearing your cache. If you see any errors or if old versions load, contact Salesforce support to enable the **Template API** setting. This helps updates show up faster and more reliably.

Default Setup Property	Description	Task Documentation
Enable reusable	If checked, enables the Omniscript to be embedded in another Omniscript.	Embed an Omniscript in Another Omniscript

Default Setup Property	Description	Task Documentation
Enable SEO	Makes your Omniscript's URL available to search engines.	Enable SEO for Omniscripts
Set the currency code	Overrides the default currency by configuring the currency code field. By default, Omniscript uses the org default currency in single-currency orgs and the user's currency in multi-currency orgs.	Configure an Omniscript's Currency Code
Fetch picklist values at script load	If selected, enables picklist values to be retrieved at script load. If deselected, picklists are fetched at design time. Sometimes, the Omniscript can't retrieve the picklist values because it reached the Apex CPU limits. In that case, deselect the checkbox and reactivate the Omniscript. If you later add more values, deactivate and reactivate the Omniscript to refresh the data.	Populating Picklist Values in Omniscript Inputs from Apex
Seed data JSON	Enables the Omniscript to be seeded with JSON data on launch. The property doesn't allow for referencing other data with the <code>%element%</code> syntax or use of expressions. For more robust functionality, see Set Values in an Omniscript .	Seed Data Into an Omniscript
Enable unload warning	Triggers a Leave Site? warning when reloading, navigating away, or closing the window.	None available
Enable logging metrics	Sends data to Salesforce about which Omniscript elements are used at runtime. The data doesn't include hidden actions and steps, and it doesn't collect user data.	None available
Enable logging metrics for hidden actions and steps	Sends data to Salesforce about which Omniscript hidden actions and steps are used at runtime. The data doesn't collect user data. By default, the property isn't selected due to possible decreased performance.	None available
Set the console tab title and icon	Applies text and an icon to the console tab where the Omniscript is embedded. These settings apply	None available

Default Setup Property	Description	Task Documentation
	only if the Omniscript owns the entire tab or subtab. They aren't used if the Omniscript is embedded on a page with other components.	

Setup Properties

Setup property sections include sets of properties that apply to a specific configuration, such as displaying Knowledge Base articles in your Omniscript.

Setup Property	Description	Task Documentation
Step Chart Options	Hide the step chart or choose where to display it relative to your Omniscript.	Configure Step Chart for an Omniscript
Cancel Options	Enable cancel functionality and configure how the cancel behavior executes.	Configure Omniscript Cancel Options
Save Options	Enable users to save an Omniscript for later use.	Save and Resume an Omniscript
Knowledge Options	Enable Knowledge Articles to be displayed in or alongside your Omniscript.	Integrate Salesforce Knowledge with Omniscript
Error Messages	Set a default error message or simplify error messages for users.	Customize Omniscript Error Messages
Messaging Framework	Pass information using Pub/Sub, windowPostMessage, and Session Storage.	Messaging Framework for Omniscripts
Styling Options	Customize the design of your Omniscript using custom style sheets, design tokens, and a scroll animation.	Custom Styles for Omniscripts

Configure an Omniscript's Currency Code

Override an org's default currency in an Omniscript by setting the Omniscript's Currency Code.

By default, Omniscript uses an org's default currency in single currency orgs and a user's currency in multi-currency orgs. Use one of the options in this table to override the default Currency Code.

Org Currency	Option	Example
Single Currency	From Omniscript Setup, click Currency Code and select a Currency from the dropdown list.	Currency Code = JPY
Multi-Currency	Dynamically pass in the Currency Code from a URL using the OmniScriptCurrencyCode parameter.	& <code>c__OmniScriptCurrencyCode=JPY</code>

Embed an Omniscript in Another Omniscript

An Omniscript can be reused in one or more existing Omniscripts. Reusing Omniscripts enables you to build a variety of smaller scripts and then piece them together into one or more parent scripts. Embedded Omniscripts behave just like other Omniscript elements.

A reusable Omniscript cannot contain another reusable Omniscript. Additionally, no element in a reusable Omniscript can have the same name as an element in the parent script. Reusable Omniscripts adopt the script configuration of the parent script.

Make sure that each embedded Omniscript defines unique **Element Name** JSON nodes in its Set Values elements. If two child Omniscripts of the same parent set values for the same JSON nodes, only values from the first child Omniscript are set, even if both children run conditionally.

1. In the Omniscript you want to reuse, click Setup, and select **Reusable**.
2. Activate the reusable Omniscript.
3. Navigate to the Omniscript where you want to use the reusable Omniscript.
4. From the elements panel, expand the **Omniscripts** section.
5. Locate the reusable Omniscript and drag it into the canvas.
6. Preview your Omniscript to test the behavior.

Activate and Deactivate Embedded Omniscripts

When you activate or deactivate an embedded script, Salesforce updates all activated parent Omniscripts to reflect the change.

1. From the Setup section of the embedded Omniscript, activate or deactivate the Omniscript.
2. Verify the Affected Omniscripts (any active Omniscript that this form is embedded in), and click **Proceed**.

This OmniScript "Addr script" is embedded in the following list of active OmniScripts. Updating it will result in updating all the embedding OmniScripts.

OmniScript Name	Version	Status
FoodStampApplication	3	Not Updated
Marriage License	12	Not Updated

- After the parent scripts update, click **Done**.

Conditionally Run Embedded Omniscripts

Using the Conditional View property in the parent Omniscript, you can run an embedded Omniscript only under certain conditions.

- From the Properties section of the reusable Omniscript element of the parent Omniscript, expand **Conditional View**.
- Click the **Show Element If True** link. The Edit Show Hide Rules window opens.
- Create conditions as described in [Conditionally Display Omniscript Elements](#).
- Click **Save**.

Elements in the embedded Omniscript run only if the conditions are true for those elements. If the conditions change while the embedded Omniscript is running, subsequent embedded Omniscript elements don't run. If the conditions are false for any embedded Omniscript elements, those elements are skipped.

Enable SEO for Omniscripts

Make Omniscripts in Communities appear in online searches by enabling SEO.

SEO Omniscripts store the state of an Omniscript in the URL by setting the `c__step` parameter to the name of an active Step automatically. Direct users to a specific step in the Omniscript by setting the `c__step` parameter in SEO URLs to a Step name in the Omniscript. Refreshing the page does not cause the Omniscript to begin at the first Step because the Step is stored in the `c__step` parameter. Include additional parameters to store the state of elements, insert data into the Omniscript's data JSON, prefill elements, or conditionally fire Omniscript Actions. Storing the state enables a user to navigate to previous or next steps using their browser's back and forward buttons.

- See [Set Up SEO for Your Community](#) (Salesforce Documentation).
- Review [SEO Best Practices and Considerations for Guest Users](#) (Salesforce Documentation).

Note SEO is automatically disabled when running inline Omniscripts.

- In the Omniscript's Setup section, select **Enable SEO**. The Omniscript automatically adds the `c__step` parameter in the URL to store the state of a Step.

2. Store the state of elements in your Omniscript, prefill elements, or pass data into the Omniscript's data JSON using additional URL parameters.
 - a. In the **Additional SEO URL Parameters** field, enter a parameter with the prefix `c__`. When storing an element's state or prefilling an element, use the name of the element as your parameter. For example, the parameter for an element named **Text1** is `c__Text1`.
 - b. Set the parameter equal to an element in the Omniscript using merge field syntax or a static value. If there is more than one parameter present, separate the parameters with an `&` symbol.
- See [Access Omniscript Data JSON with Merge Fields](#).
For example, to store the state of the **Text1** element, set the `c__Text1` parameter equal to `%Text1%`.
3. Activate the Omniscript, and add it to a Community.
See [Add Your Omniscript to a Lightning Page](#) and [Add Your Omniscript to an Experience Cloud Page](#).
4. Grant guest user access in your Community.
See [Create Sharing Rules for Guest Users](#).
5. Add the Omniscript URL to the Community's `sitemap.xml` file.
See [Set Up SEO for Your Community](#).
6. Use a search engine, such as Google, to test the SEO functionality.
7. Test the Omniscript's forward and backward navigation using a browser's forward and back buttons.

Configure Step Chart for an Omniscript

Hide or move an Omniscript's Step Chart using Step Chart Options.

1. In the Omniscript's Setup section, expand **Step Chart Options**.
2. Display the Step Chart on either side of the Omniscript or above the Omniscript by selecting **Right**, **Left**, or **Top** in the Step Chart Placement dropdown.

 **Note** In the mobile view, the step chart appears on the top of the Omniscript irrespective of its placement setting. Also, step chart alignments are not supported if you're using the Newport Design System.
3. Select **Hide** to hide the step chart from users.

Configure Omniscript Cancel Options

Navigate users to different Salesforce experiences after canceling an Omniscript by configuring cancel options. Omniscript's cancel functionality enables users to cancel an Omniscript from a Step by clicking a cancel link. The cancel link directs the user to a Salesforce experience set in the cancel options.

Cancel options are built on top of the Navigate Action and use the same Page Reference Types to navigate to different experiences.

 **Note** Restart Omniscript isn't supported as a cancel option.

1. From the Omniscript's Setup panel, expand **Cancel Options** and enable users to cancel the

Omniscript.

Enabling users to cancel an embedded Omniscript also enables them to cancel the parent Omniscript and any other Omniscripts embedded in the parent. Similarly, enabling users to cancel the parent Omniscript enables them to cancel all embedded Omniscripts. In both cases, a cancel link is included in all parent and child Omniscripts.

2. In **Field Label**, enter the cancel option's display text. The text displays as a clickable link in every Step element.
3. If needed, select the option to show a prompt before canceling the Omniscript, and enter a confirmation message for the prompt.
Users can return to the Omniscript or confirm the cancellation.
4. If needed, select the option to replace the existing entry in the browser history with a **Page Reference Type**.
Enabling the property prevents users from navigating back to the Omniscript.
5. Determine the Salesforce experience to which the cancel link directs by clicking and selecting a **Page Reference Type**.
For available page reference types, see [Open Other Pages from Omniscripts with the Navigate Action](#).
6. For the selected Page Reference Type, enter additional parameters to pass to the experience. For information about Page Reference Types, see [PageReference Types](#).

Save and Resume an Omniscript

Enable users to save and resume Omniscripts by using Omniscript's Save Options property. Users can bookmark links, launch a saved instance from an object page, or email the link. Configure custom URLs to resume saved Omniscript instances in different domains. For example, a customer can initiate an Omniscript inside of a Community, and then a customer service representative can resume the Omniscript inside of the Service Console.

[Restrictions and Limitations for Save and Resume](#)

Restrictions and limitations exist for saving and resuming Omniscripts. Some configurations do not support save and resume. Some users are not allowed to perform save and resume, and some users require certain permissions to use the feature.

[Configure Save Options](#)

Enable your users to save and resume Omniscripts by configuring the save for later functionality. To save only a step in the Omniscript, select or deselect **Allow save for later** in the Step properties.

[Map Saved Instance URLs to Custom Fields](#)

Configure where a saved Omniscript instance launches in Salesforce by mapping URLs to custom fields on the Saved Omniscript object.

[Customize the Save for Later Error Message](#)

Create custom error messages with Omniscript's error handling framework.

[Conditionally Display Elements in Saved Omniscripts for Different User Profiles](#)

Control which Omniscript elements display to different profile users in a saved Omniscript instance by using conditions.

[View In-Progress and Completed Omniscripts](#)

Enable users to view in-progress Omniscripts that have been saved for later by using Visualforce pages

and the Vlocity Omniscript Workbench.

Specify a Resume Link in a Visualforce Page

Users can create a VisualForce page that specifies the custom URL used for Save and Resume.

Restrictions and Limitations for Save and Resume

Restrictions and limitations exist for saving and resuming Omniscripts. Some configurations do not support save and resume. Some users are not allowed to perform save and resume, and some users require certain permissions to use the feature.

The save for later feature isn't supported in these cases:

- *Embedded OmniScripts*. When an Omniscript is included in a custom Lightning web component, the outer Omniscript does not save the inner Omniscript.
- *Child (reusable) Omniscripts*. When a child Omniscript is included in a parent Omniscript, only the steps and actions from the child Omniscript are copied to the parent Omniscript. All configurations, such as save for later and custom styles, are based on the parent Omniscript. They are not copied from the child Omniscript to the parent Omniscript.
- *Set error*. Because set error is not saved in the JSON data, it is not saved in the Saved Session (**OmniScriptSavedSession**) object.
- *Guest users and Experience Cloud guest users*. Such users can't use Omniscript Saved Sessions because [security policies](#) prevent them from retaining ownership of records they create.

The save for later feature requires specific permissions in these cases:

- If a user updates an Omniscript and saves it for later, another user can't access the saved Omniscript unless that user is granted the necessary permission.
-  **Note** Note: Due to a limitation, Save for Later doesn't work for edit scenarios between users accessing a form via Experience Cloud sites, even when the requisite permissions are assigned.
- Users must have access to the **File Based Omniscript Name** field of an Omniscript Saved Session object to use save for later. Without this access, they can't save an Omniscript for later or resume an Omniscript that was previously saved for later.

Configure Save Options

Enable your users to save and resume Omniscripts by configuring the save for later functionality. To save only a step in the Omniscript, select or deselect **Allow save for later** in the Step properties.

1. From the Omniscript's Setup panel, expand the Save Options section.
2. To enable the save for later feature, select **Allow save for later**.
3. If needed, save the Omniscript automatically when users click Next.
Users see an auto save message when they click Next.
4. In the **Save Name Template** field, enter a template name format with which to save the Omniscript. This field supports merge fields. For example, `%LastName%, %FirstName% - Application`. If the field is blank, the default is `Saved-Omniscript Name-RowId`.

5. Enter the number of days for which the Omniscript is to remain usable before it expires.
If a user attempts to resume the Omniscript after it expires, the Omniscript restarts.
6. Set the object ID to attach the saved Omniscript to a record.
The default value is `%ContextId%`. This field supports merge fields. For more information, see [Access Omniscript Data JSON with Merge Fields](#).
7. If needed, enable users to merge data into an updated Omniscript by selecting **Merge Saved Data JSON into updated Omniscript**.

If **Merge Saved Data JSON into updated Omniscript** is selected, the Omniscript merges the data from the saved instance into the updated Omniscript when a saved instance launches. However, if you used the Set Value action in a step of your Omniscript, that step takes precedence over the saved value, and Omnistudio doesn't retain any previously saved value.



Note If an Omniscript is deactivated or activated, a new instance of the Omniscript is always created. Multiple instances of forms are created, and saved instances are no longer available. Selecting **Merge Saved Data JSON into updated Omniscript** prefills a form of the new instance of the Omniscript with saved data. To use saved fields in the new version of the Omniscript, first select either **Merge Saved Data JSON into updated Omniscript** or set the **ContextId** field. Then ensure that the value of the **Save Object Id** field is the same in both Omniscripts. These steps map JSON data from the saved version of the Omniscript into the new version of the Omniscript that is created.

8. In the Save URL Patterns section, map the Omniscript URL to a custom field in an object. See "Map Saved Instance URLs to Custom Fields."
9. To enable **Allow save for later** on a per-step basis:
 - a. From a Step element, expand the Save Options section.
 - b. To enable or disable save for later on the step, select or deselect **Allow save for later**.

Map Saved Instance URLs to Custom Fields

Configure where a saved Omniscript instance launches in Salesforce by mapping URLs to custom fields on the Saved Omniscript object.

Configure **Save URL Patterns** to control the URL format that maps to any custom fields. You must create a custom field in the Saved Omniscript object for every additional URL that you save to an Omniscript instance.

1. From Salesforce Setup, open the **Object Manager**, and select the **Saved Omniscript** object.
2. Add a new custom field that accepts URLs.
3. Copy the field's **API name**.
4. In the **Save URL Patterns** section, in the **Field API Name** field, paste the field API name.
5. In the **URL Pattern** field, enter a URL using these syntax patterns:



Note Omniscripts must use the LWC Omniscript Wrapper URL patterns to launch saved Omniscript instance URLs.

- Community Pages:

Using the Community LWC Omniscript Wrapper URL pattern, add the parameter `c__instanceId`, and set it to `{0}`. Set the `c__target` parameter to `{1}`.

```
https://myDomain.force.com/CommunityName/s/CommunityPage/c__layout=lightning&c__instanceId={0}&c__target={1}
```

- Lightning Pages:

Using the Lightning LWC Omniscript Wrapper URL pattern, add the parameter `c__instanceId` and set it to `{0}`. Set the `c__target` parameter to `{1}`.

```
https://myDomain.force.com/lightning/cmp/namespace__vvelocityLWCOmniWrapper?c__layout=lightning&c__instanceId={0}&c__target={1}
```

6. Test the Omniscript by taking these steps:

- Save and activate the Omniscript.
- Preview the Omniscript and save the instance.
- Copy the saved Omniscript link.

Paste the link into your browser to view the saved instance.



Note If the link does not work, ensure your URL format and any Community page names are correct.

Customize the Save for Later Error Message

Create custom error messages with Omniscript's error handling framework.

When multiple users are updating the same saved Omniscript instance, the first user that saves the Omniscript becomes the owner of that instance. The user that is unable to save the Omniscript receives an error that informs them that they need to refresh the Omniscript. Customize the error message users receive by using the **Error Messages** property. For more information, see [Customize Omniscript Error Messages](#).

- In the Omniscript's Setup panel, expand the **Error Messages** section.
- Leave the **Path** value blank. The error message is in the Omniscript's root path.
- In the **Custom Error Messages** section, leave the **Path** value blank. The error message is in the Omniscript's root path.
- In the **Value** field, enter this default error message text: *This saved Omniscript has been updated since resuming. To see the latest updates, please exit and resume the saved Omniscript again.*
- In the **Message** field, enter a custom error message.
- Save and activate the Omniscript.
- Test the Omniscript by taking these steps:
 - Open the Omniscript and save an instance.
 - Resume the saved Omniscript instance in two separate tabs.

- c. In one of the instances, add additional information and save the Omniscript.
- d. In the other instance, try to add information and save the instance to view the custom error.

Conditionally Display Elements in Saved Omniscripts for Different User Profiles

Control which Omniscript elements display to different profile users in a saved Omniscript instance by using conditions.

When the Omniscript resumes, it uses the profile of the current user. Conditions on elements can determine whether the user has access to the element. For example, an agent may see data that does not display to customers.

1. In an element's Conditional View property, click **Add Condition**.
2. In the field, enter *userProfile*.
3. In the value field, enter the name of a User Profile. For example, System Administrator.

Specify a Resume Link in a Visualforce Page

Users can create a VisualForce page that specifies the custom URL used for Save and Resume.

Create a new Visual ForcePage and enter the code below. Replace the FieldAPIName variable with the Field API Name specified under the **Save URL Patterns**.

```
<apex:page standardStylesheets="false" showHeader="true" standardController="Account" extensions="VFPageControllerBase" sidebar="true" docType="html-5.0" >
<c:OmniScriptInstanceComponent standardController="{!!stdController}" resumeFieldIdName="FieldAPIName"/>
</apex:page>
```

View In-Progress and Completed Omniscripts

Enable users to view in-progress Omniscripts that have been saved for later by using Visualforce pages and the Vlocity Omniscript Workbench.

Use prebuilt Visualforce pages for Accounts and Contacts, create custom Visualforce pages for additional records, or view all records in the Vlocity Omniscript Workbench.

-  **Note** If an Omniscript launches from the context of a Contact or Account record, you can view all in progress or completed Omniscripts for that specific Account or Contact under the **Customer Story** section.

Display Saved Omniscript Instances for Accounts and Contacts

Add the provided **OmniScriptInstanceAccountPage** Visualforce page to the **Account layout** or the **OmniScriptInstanceContactPage** to the **Contact layout**.

[Display Saved Omniscript Instances on ObjectRecord Pages](#)

Display saved Omniscript instances on an object record page by creating a Visualforce page and adding the component to the object's layout.

[View All Saved Omniscript Instances](#)

View all Saved Omniscript instances by opening the Vlocity Omniscript Workbench.

Display Saved Omniscript Instances on ObjectRecord Pages

Display saved Omniscript instances on an object record page by creating a Visualforce page and adding the component to the object's layout.

1. Create a new Visualforce page.
2. Copy and paste the following code into a new Visualforce page.

```
<apex:page standardStylesheets="false" showHeader="true" standardController="ObjectName" extensions="NS.PlatformVFPageControllerBase" sidebar="true" docType="html-5.0" >

<NS:OmniScriptInstanceComponent standardController="{!!stdController}" />

</apex:page>
```

3. Replace these variables in the code with the appropriate values:
 - **ObjectName** Enter the API name of the standard or custom object.
 - **NS** : Enter the Namespace of the package. For more information, see [View the Namespace and Version of Managed Packages](#).

View All Saved Omniscript Instances

View all Saved Omniscript instances by opening the Vlocity Omniscript Workbench.

To open the Omniscript Workbench:

1. Open the Salesforce App Launcher.
2. Click **Omniscript Workbench**.

Integrate Salesforce Knowledge with Omniscript

Enable users to search and view Salesforce Knowledge Articles when using an Omniscript. Configure Omniscript Knowledge Options to search for articles based on static values and dynamic inputs from

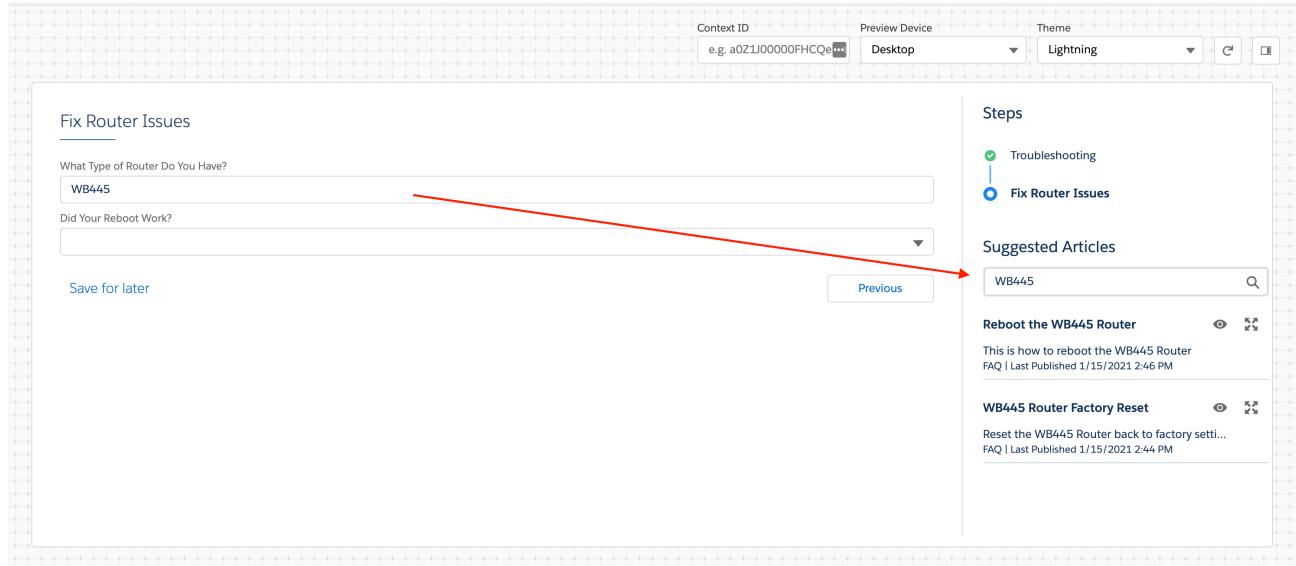
Omniscript fields, or a query from a simple search bar.

For each set of search criteria for knowledge articles, the maximum number of results displayed is 2000 records. Salesforce set this limit for SOSL queries.

! **Important** Your organization must have [Salesforce Knowledge](#) enabled to use this feature.

From the Omniscript Designer Preview, you can open the article in a new browser tab.

Simple Search Bar



Set Up Omniscript to Work with Salesforce Knowledge

To let users search and view Salesforce Knowledge articles in an Omniscript, integrate Salesforce Knowledge into the Omniscript. Omniscript also supports querying Lightning Knowledge.

Configure Knowledge for Individual Steps in the Omniscript

After the initial configuration to set up your Omniscript to work with Salesforce Knowledge, integrate Knowledge in your Omniscript Steps. Configure options on each step to enable Knowledge and filter results.

Open Knowledge Base Articles Outside an Omniscript

Launch Knowledge base articles outside of an Omniscript using the Omniscript Knowledge Base component in a Community or Lightning page.

Set Up Omniscript to Work with Salesforce Knowledge

To let users search and view Salesforce Knowledge articles in an Omniscript, integrate Salesforce Knowledge into the Omniscript. Omniscript also supports querying Lightning Knowledge.

Before you begin:

- Enable Salesforce Knowledge.

- (Optional) Enable Lightning Knowledge.

 **Tip** Because search results display a maximum of 2000 results, set up search filters so users can refine their criteria for more accurate results.

To set up Omniscript for Salesforce Knowledge:

1. From the Setup panel, expand the **Knowledge Options** section and select **Enable Knowledge**.
2. If needed, to integrate Lightning Knowledge with Omniscript, select **Use Lightning Knowledge**. Lightning Knowledge must be enabled in your org.
3. If needed, to open Knowledge articles outside of an Omniscript using the **Vlocity Omniscript Knowledge Base** component in a Community or Lightning page, select **Display Outside Omniscript**.
4. If needed, to filter Knowledge articles by record type, select **Filter Knowledge Search by Record Type**. Knowledge searches use only the types listed in the Record Type Filter property for the step.
5. To search only specific fields in articles, use the Knowledge Article Type Query Fields Map section.
 - a. In **Article/Record Type API Name**, enter the Article Type API name for Classic Knowledge or enter the Record Type API name when **Use Lightning Knowledge** is enabled.
 - b. In **Field API Name**, enter the API name of the field to include in searches, such as *Title*.

For multiple fields, enter a comma-separated list with no spaces, such as *Title*,
namespace__richText__c,*Summary*.

The Omniscript preview displays these fields.

6. To query additional fields and record types, click **+ Add New Knowledge Type Field Map** and repeat step 5.
7. If **Use Lightning Knowledge** is enabled, enter the Lightning Knowledge object's API name into the **Lightning Knowledge Object API Name** field, such as *namespace__Knowledge__kav*.
8. (Optional) In **Label**, enter the text displayed above the Knowledge component, such as *Suggested Articles*.

Configure Knowledge for Individual Steps in the Omniscript

After the initial configuration to set up your Omniscript to work with Salesforce Knowledge, integrate Knowledge in your Omniscript Steps. Configure options on each step to enable Knowledge and filter results.

Before you begin, set up your Omniscript to work with Salesforce Knowledge.

1. From the Omniscripts Designer, click the Step that you want to display the Knowledge component.
2. In the Properties panel, expand the **Knowledge Options** section, and click **Enable Knowledge**.
3. Update **Language** from the default value **English** to your preferred language.
4. Update **Public Status** from the default value *Online* to **Archive** or **Draft**.
5. In the **Keyword** field, enter the text to search for.

This text can be literal text, such as *Service Disconnect*.

Or you can use merge fields for variable data. For example, enter `%RouterType%` to enable your users

to enter keywords to search for in a Text element whose **Name** is RouterType. For information on merge fields, see [Access Omniscript Data JSON with Merge Fields](#).

6. To filter articles by their [Data Category](#), enter a SOQL query in the **Data Category Criteria** field. For example, enter `Troubleshooting_c AT (Router_c)` to show articles from the subcategory `Router` under the parent category `Troubleshooting`. For information on building queries for data categories, see [With Data Category filtering Expression](#).
 **Note** You don't need to enter **WITH DATA CATEGORY** in your query.
7. If needed, when Filter Knowledge Search by Record Type or Lightning Knowledge is enabled in the Setup panel, enter the Record Type's API name in **Record Type Filter** to search those record types. For multiple Record Types, enter a comma-separated list with no spaces, such as `FAQ, INFO`.

Open Knowledge Base Articles Outside an Omniscript

Launch Knowledge base articles outside of an Omniscript using the Omniscript Knowledge Base component in a Community or Lightning page.

The Omniscript Knowledge Base component renders Knowledge base articles side by side with an Omniscript or as a standalone component. When opening Knowledge Base articles from an Omniscript, Omniscript passes specific information into the component to search the Knowledge base and render articles.

1. In your Omniscript's Setup panel, expand **Knowledge options**, and enable knowledge articles.
2. Enable Salesforce Lightning Knowledge, and configure the remaining fields using the instructions in [Set Up Omniscript to Work with Salesforce Knowledge](#).
3. If needed, enable users to open knowledge base acrticles outside the Omniscript.
4. Activate the Omniscript.
5. Open the Lightning App Builder and create a page with a layout containing at least two regions. See [Lightning App Builder](#).
6. From the Lightning page, drag the deployed LWC Omniscript component into the page. See [Add Your Omniscript to a Lightning Page](#) and [Add Your Omniscript to an Experience Cloud Page](#).
7. From the Components section, drag the **Vlocity Omniscript Knowledge Base** component to where you want it displayed.
8. In the **omniscriptKey** field, enter the name of the LWC Omniscript component using the syntax `type_SubType_Language`.
9. If needed, in the **layout** field, enter `newport` to render the Knowledge base article with the Newport Design System styling.
10. Save the Lightning page and test the Omniscript on the Lightning page to preview the behavior.

Seed Data Into an Omniscript

Prefill Omniscript fields with data or add hidden nodes to the Omniscript Data JSON using the Seed Data JSON property in the Omniscript's Setup section.

Use the Seed Data JSON property with static values that seed during the initial load of the Omniscript. To set values using formulas or with information entered into the Omniscript after the initial load, see [Set Values in an Omniscript](#).

-  **Note** When embedding Omniscripts in the utility bar, flyout, or modal, ensure they scroll to the top when users navigate with Previous or Next buttons by using the "omniscriptEmbeddedSource" key in the prefill. Set the key `omniscriptEmbeddedSource` with a value of `utilitybar`, `flyout`, or `modal` accordingly.

1. Open the Omniscript Setup panel.
2. In the Omniscript's **Setup**, find **Seed Data JSON** section, and provide information to prefill the Omniscript during the initial load.
 - a. Click **Add New Key/Value Pair**.
 - b. In the **Key** field, enter the name of the element to prefill.
 - c. In the **Value** field, enter the desired value of the element.
If the element has multiple responses, such as a multi-select, you can prefill more than one value by separating the entries with a semicolon.

When the user launches the Omniscript, the elements specified in the Seed Data JSON section contain the static values.
3. If needed, use the Seed Data JSON property to add hidden nodes to the Data JSON of the Omniscript. For example, an Omniscript used for troubleshooting may create a case. You could use the default values from hidden nodes to prefill values. You may want the Omniscript to create a Case with Type, Status, and Origin defined in the Data JSON of the Omniscript. Using this method, you can have multiple Omniscripts associated with one Omnistudio Data Mapper interface that creates the Case.

When the user launches the Omniscript, the elements specified in the Seed Data JSON section contain the static values.

Customize Omniscript Error Messages

The error message handler enables you to replace default error messages from remote apex calls with more user-friendly error messages when errors occur in Omniscript. The Error Messages handler property is available in File elements, Image elements, all Action elements, and Setup.

-  **Note** Error Messages set in the Setup properties apply to any matching error returned by an element, while Error Messages set in an Action's properties only render for that Action.

Error Message Type	Description
Default Error Message	A default message that returns when no Custom Error Messages are matched or defined.

Error Message Type	Description
Custom Error Messages	<p>Messages that are returned based on the defined path and value. Custom Error Messages are configured using the following fields:</p> <ul style="list-style-type: none"> • Path: A merge field path that locates the original default value inside an error object. • Value: The original error message, including spaces, that will be replaced by the text defined in the Message field. This value must match the original error exactly. • Message: The custom message that will replace the error text defined in the Value field.

Set the Default Error Message on an Omniscript Element

Replace a Default Error Message with a user-friendly error message when a remote apex call error occurs in your Omniscript. A default message returns when no custom error messages are matched or defined. Update the default error messages for Image, Action, and File elements.

Before You Begin

Before configuring **Error Messages**, preview the errors to determine their format.

1. Open the **Error Messages** property in the element that renders the error.
2. In the **Default Error Message** field, enter your custom message.
3. Preview the error to ensure the message renders.

Set a Custom Error Message That Returns as an Object

Set a Custom Error Message for an error that returns as an object. A Custom Error Message returns based on a defined path and value. Set custom error messages for File elements, Image elements, all Action elements, and Setup.

Before You Begin

Before configuring **Error Messages**, preview the errors to determine their format.

1. Run the Omniscript to encounter the error.
2. Determine the path to the error. For example, the path for errorCode in the following error

```
[{"errorCode": "NOT_FOUND", "message": "The requested resource does not exist"}]
```

is 0 | : | errorCode .
3. Copy the error message text that follows the path. For example, the error message text for errorCode

in the following error: `[{"errorCode": "NOT_FOUND", "message": "The requested resource does not exist"}]` is NOT_FOUND .

4. Open the **Error Messages** property in the **Action Element** that is rendering the error.
5. In the **Path** field, enter the path determined by Step 2.
6. In the **Value** field, paste the original error.
7. In the **Message** field, enter your custom message.
8. Test to ensure the message is being replaced.

Set a Custom Error Message That Returns as a String

Set a Custom Error Message for an error that returns as a string. A Custom Error Message returns based on a defined path and value. Set custom error messages for File elements, Image elements, all Action elements, and Setup.

Before You Begin

Before configuring **Error Messages**, preview the errors to determine their format.

1. Run the Omniscript to encounter the error.
2. Copy the text following the colon after the word Error.
3. Open the **Error Messages** property in the Action Element that's rendering the error.
4. In the **Path** field, enter the path to the original error using a merge field syntax.

For example, to reference the error message in this JSON string:

```
{  
    "errorMessage" : "An Error has occurred in an apex class",  
    "error" : "OK"  
}
```

Enter `errorMessage` in the Path field.

If the JSON string is more complex such as:

```
{  
    "errors" : [ { "errorMessage" : "An error has occurred. " } ]  
}
```

Enter `errors|0:errorMessage` in the Path field.

5. In the **Value** field, paste the original error.
6. In the **Message** field, enter your custom message.
7. Test to ensure the message is being replaced.

See Also

[Access Omniscript Data JSON with Merge Fields](#)

Messaging Framework for Omniscripts

Communicate between an Omniscript, windows, Lightning web components, and Omniscript elements with window post messages, session storage messages, and PubSub messaging.

Determine which object or component needs to handle and use data, and use the corresponding property.

Property	Description	Object or Component	Example
Window postMessage	Send information from an element to a window object in key-value pairs.	Window object	Message with Window Post Messages and Session Storage Messages
Pub/Sub	Communicate with Omniscript from a custom LWC by sending key-value pairs. Use the events passed in the key-value pairs to trigger custom behavior in a component.	Lightning Web Components	Communicate from Omniscript to a Lightning Web Component This topic is part of creating custom Lightning web components for Omniscripts.
Session Storage	Send information to a Session Storage object using key-value pairs. A session storage object clears when a page session ends.	Session Storage object	Message with Window Post Messages and Session Storage Messages

Message with Window Post Messages and Session Storage Messages

Communicate between Omniscript, windows, and components with window post messages and session storage messages.

Pass element information and data JSON from Omniscript elements containing the window post message, session storage message, and LWC Pub/Sub message. For information on using the pub/sub message in Omniscript, see [Communicate from Omniscript to a Lightning Web Component](#).

1. To add window post messages or session storage messages:

- a. In the element, check the **Window Post Message** checkbox or the **Session Storage Message** checkbox.

Default Messaging:

```
{  
    Type: <element_type>,  
    OmniEleName: <element_name>  
}
```

When enabled, the default messaging sends the name of the element and the type of the element.

- a. (Optional) In the **Messages** property, add additional messaging to the node by entering a Key/Value pair.
The **Value** field accepts merge syntax. For information on merge fields, see [Access Omniscript Data JSON with Merge Fields](#).
- a. Pass a value in the `c__messagingKey` URL parameter to change the node that stores the messaging payload for `window.postMessage` and session storage message.

2. To add messages to a Step:

- a. In the Step's properties, click **Edit Properties as JSON**.
- a. Paste in the following JSON before the closing curly bracket:

```
"wpm": false,  
"ssm": false,  
"message": {}
```

- b. Enable WPM or SSM by setting the value equal to `true`.

```
"wpm": true
```

- c. Add additional key/value pairs inside the message object.

```
"message": {"Account": "%AccountId%", "Status": "%AccountStatus%"}
```

What's next: Preview the Omniscript and test the messaging behavior by inspecting the corresponding element in your browser's console or developer tools.

Integrate DocuSign with Omniscripts

Enable users to either sign DocuSign forms in an Omniscript or email a user a copy of the document to sign at their convenience. DocuSign integration requires an active DocuSign account.

[Prepare a DocuSign Account for Omnistudio](#)

Create a developer account in DocuSign and configure it for integration with Omnistudio.

[Configure a Salesforce Org for DocuSign Integration](#)

Enable Omnistudio to use DocuSign templates by configuring authentication providers, named

credentials, and Omni Interaction Configurations. You can also enable sending or signing DocuSign documents on behalf of other users.

Prepare a DocuSign Template and Omnistudio Data Mapper Transform for Omniscript Use

Prepare DocuSign templates to receive data from Omniscripts or Integration Procedures. Retrieve those templates in Omnistudio. Then map Omniscript data to a template using a Data Mapper Transform.

Prepare a DocuSign Account for Omnistudio

Create a developer account in DocuSign and configure it for integration with Omnistudio.

1. Create a standard [DocuSign](#) account.
2. Go to **Apps and Keys** in the DocuSign **Settings** and click the link to create a developer account. See [Apps and Keys](#) in the DocuSign documentation.
If you use the same login for both DocuSign accounts and you log out, log back into the developer account at developer.docusign.com. To go to the Home, Templates, and Settings pages, click your profile icon and select **My Apps & Keys**.
3. In Apps and Keys for the developer account, create a DocuSign application that uses **Authorization Code Grant** as the authentication type.
4. Add a **Secret Key** to your DocuSign application. See [Add Integration Keys](#) in the DocuSign documentation.
5. Copy the Account ID, Base URI, Integration Key, and Secret Key from the developer account. These items are required for Omnistudio integration.

What's Next: [Create an Authentication Provider to Connect to DocuSign](#)

After you've tested your Omniscripts and DocuSign templates, move your templates and app into a DocuSign production account. See [Download Templates](#), [Upload Templates](#), and [Go-Live](#) in the DocuSign documentation.

Configure a Salesforce Org for DocuSign Integration

Enable Omnistudio to use DocuSign templates by configuring authentication providers, named credentials, and Omni Interaction Configurations. You can also enable sending or signing DocuSign documents on behalf of other users.

1. [Create an Authentication Provider to Connect to DocuSign](#)
Add an authentication provider for Salesforce to create a secure connection to your DocuSign account. Omnistudio uses these credentials to fetch document templates from DocuSign.
2. [Create an OAuth 2.0 Named Credential to Connect to DocuSign](#)
Add a Named Credential using authentication provider information to create a secure connection. Omniscripts and Integration Procedures use these credentials to connect with DocuSign. Create a named credential for the admin who manages templates and your DocuSign account. You can also create a Send on Behalf named credential to send or sign DocuSign forms on behalf of other users.
3. [Add the DocuSign Login Page as a Trusted URL](#)

DocuSign runs on an iFrame. To allow the content to show on Experience Cloud sites, add the DocuSign login page as a trusted URL and allow iFrame content.

4. Add Omni Interaction Configuration Settings for DocuSign

Provide DocuSign authorization by adding two Omni Interaction Configuration entries for DocuSign.

5. Set Up Send on Behalf of Another User for Omniscripts

To send or sign DocuSign on behalf of another user, create a Send On Behalf field in the User object and assign the Send on Behalf named credential from that field in the user's page. If you want multiple users to send or sign on behalf of another user, assign the named credential to each user. When you log in, Salesforce checks whether a Send On Behalf setup exists. If it doesn't, Salesforce sends or signs DocuSign using the default admin user.

Create an Authentication Provider to Connect to DocuSign

Add an authentication provider for Salesforce to create a secure connection to your DocuSign account. Omnistudio uses these credentials to fetch document templates from DocuSign.

Before you begin: [Prepare a DocuSign Account for Omnistudio](#)

1. In your Salesforce org, from Setup, enter *Auth. Providers* in the Quick Find box, then select **Auth. Providers**.
2. Click **New**, and select **Open ID Connect** as the **Provider Type**.
3. Name the Auth Provider *DocuSign*, and enter *DocuSign* in the URL Suffix field.
4. In the **Consumer Key** field, enter your DocuSign App's **Integration Key**.
5. For **Consumer Secret**, enter your DocuSign App's **Secret Key**.
6. For **Authorize Endpoint URL**, enter <https://account-d.docusign.com/oauth/auth>.
7. For **Token Endpoint URL**, enter <https://account-d.docusign.com/oauth/token>.
8. For **Default Scopes**, enter *refresh_token full*.
9. Ensure that **Send access token in header** is selected.
10. Ensure that **Include Consumer Secret in API Responses** is selected.

11. Click **Save**.

This saves your auth provider and generates Salesforce Configuration URLs.

12. Copy the **Callback URL**.
13. Log into your DocuSign developer account.
14. From your DocuSign account, in the App and Keys section, click **Edit** for your app.
15. Paste the *Callback URL* in the **Redirect URIs** field.
16. Click **Save**.

What's next: [Create an OAuth 2.0 Named Credential to Connect to DocuSign](#).

Create an OAuth 2.0 Named Credential to Connect to DocuSign

Add a Named Credential using authentication provider information to create a secure connection. Omniscripts and Integration Procedures use these credentials to connect with DocuSign. Create a named credential for the admin who manages templates and your DocuSign account. You can also create a Send on Behalf named credential to send or sign DocuSign forms on behalf of other users.

Before you begin: [Create an Authentication Provider to Connect to DocuSign](#)

1. In your Salesforce org, from Setup, enter *Named* in the Quick Find box, then select **Named Credentials**.
2. In the **New** dropdown list, select **New Legacy**.
3. For **Label** and **Name**, enter *DocuSign*.
4. For **URL**, enter <https://demo.docusign.net>.
5. For **Identity Type**, select **Named Principal**.
6. For **Authentication Protocol**, select **OAuth 2.0**.
7. For **Authentication Provider**, select **DocuSign**.
8. Enter a **Scope** of *signature impersonation*.
9. Ensure that **Start Authentication Flow on Save** is selected.
10. Ensure that **Generate Authorization Header** is selected.
11. Click **Save**.
The DocuSign login window appears.
12. From the DocuSign login window, log in to DocuSign.
This changes the Named Credential's **Authentication Status** to **Authenticated**.
13. If needed, create a Send on Behalf user.
You can repeat the previous steps. In Step 3, enter a different **Label** and **Name**, for example, *SendOnBehalf*. You authenticate the Send On Behalf user with the the Send On Behalf user's DocuSign account.

What's next: [Add Omni Interaction Configuration Settings for DocuSign](#)

Add the DocuSign Login Page as a Trusted URL

DocuSign runs on an iFrame. To allow the content to show on Experience Cloud sites, add the DocuSign login page as a trusted URL and allow iFrame content.

To add the DocuSign login page as a trusted URL, perform these tasks.

1. From Setup, find and select **Trusted URLs**.
2. Click **New Trusted URL**.
3. Provide an API Name such as *DocuSignAccount*.
4. In the URL field, enter <https://account-d.docusign.com>.
5. Select the **Active** checkbox.
6. In the Content Security Policy (CSP) Settings section, select the **frame-src (iframe content)** checkbox.
You may also select other checkboxes as per your business needs.
7. Save the trusted URL.

Add Omni Interaction Configuration Settings for DocuSign

Provide DocuSign authorization by adding two Omni Interaction Configuration entries for DocuSign.

Before you begin: [Create an OAuth 2.0 Named Credential to Connect to DocuSign](#)

1. From Setup, enter *Omni Interaction Configuration* in the Quick Find box, then select **Omni Interaction Configuration**.
2. Click **New**.
3. For Name and Label, enter *DocuSignAccountId*.
4. For Value, enter the *API Account ID* of your DocuSign Account. The API Account ID is present on DocuSign's Apps and Keys page.
5. Click **Save**.
6. Click **New**.
7. For Name and Label, enter *DocuSignNamedCredential*.
8. For Value, enter *DocuSign*.

What's next: [Set Up Send on Behalf of Another User for Omniscripts](#).

Set Up Send on Behalf of Another User for Omniscripts

To send or sign DocuSign on behalf of another user, create a Send On Behalf field in the User object and assign the Send on Behalf named credential from that field in the user's page. If you want multiple users to send or sign on behalf of another user, assign the named credential to each user. When you log in, Salesforce checks whether a Send On Behalf setup exists. If it doesn't, Salesforce sends or signs DocuSign using the default admin user.

Before you begin: [Add Omni Interaction Configuration Settings for DocuSign](#)

1. From the object management settings for users, go to **Fields & Relationships**.
2. Click **New**.
3. Enter a **Field Name** of *DocuSignNamedCredential*.
4. Enter a **Field Label**, for example *DocuSign Named Credential*.
5. Enter 255 as the **Length**, and click **Next**.
6. Save your changes.
7. From Setup, in the Quick Find box, enter *Users*, and then select **Users**.
8. Click **Edit** for your DocuSign user.
9. Under Additional Information, click **DocuSign Named Credential**, enter the name of the Send On Behalf user named credential that you created, and then save your changes.
10. Repeat steps 7–9 for each user that you want to set up to send emails or signatures as the Send On Behalf user.

Prepare a DocuSign Template and Omnistudio Data Mapper Transform for Omniscript Use

Prepare DocuSign templates to receive data from Omniscripts or Integration Procedures. Retrieve those templates in Omnistudio. Then map Omniscript data to a template using a Data Mapper Transform.

 **Note** For merge fields to work, make sure that the connected user, like other users, is using the

same email address and name for the Salesforce user as is being used by the DocuSign user. For instance, if the Salesforce username is DocusignIntegrationUser@yourcompany.com, make sure that the DocuSign username is also DocusignIntegrationUser@yourcompany.com.

1. [Prepare DocuSign Documents to Use with Omniscripts](#)

Configure DocuSign documents for use with Omniscripts.

2. [Get DocuSign Templates from Omnistudio DocuSign Setup](#)

Retrieve the templates from your DocuSign account.

3. [Map Fields from the Omniscript to the DocuSign Template](#)

To populate a DocuSign template, you define Omniscript data as input to an Omnistudio Data Mapper Transform, which uses that data to populate the template. You can construct the Omniscript data manually or copy it from the Omniscript.

4. [Populate a Docusign Template for Omniscripts with a Data Mapper Transform](#)

Map Omniscript data to a Docusign template using a Data Mapper Transform.

Prepare DocuSign Documents to Use with Omniscripts

Configure DocuSign documents for use with Omniscripts.

Before you begin: [Prepare a DocuSign Account for Omnistudio](#)

1. Log in to your DocuSign developer account.

2. Click **Templates**, then click **Create a Template** or **New | Create Template**.

3. Enter a **Template name**.

4. Add a document.

5. Add a **Role** for the recipient, such as *Approver*.

6. For the role, select **Needs to Sign**, **Receives a Copy**, or **Needs to View**.

7. Add more roles and set a signing order, if necessary.

8. Click **Next**.

9. Add to the document the Text, Radio Button, and Checkbox **Custom Fields** that receive Omniscript data.

Omnistudio doesn't support mapping to DocuSign standard or prefill fields.

10. Add a **Data Label** to each custom field.

These data labels appear in the Omnistudio Data Mapper Transform that maps Omniscript fields to DocuSign template fields.

11. Click **Back** and enter a **Template description**.

Adding a description that lists the custom field data labels that receive Omniscript data is recommended.

12. Click **Next**, then click **Save and Close**.

What's Next: [Get DocuSign Templates from Omnistudio DocuSign Setup](#)

Get DocuSign Templates from Omnistudio DocuSign Setup

Retrieve the templates from your DocuSign account.

Before you begin:

- [Prepare DocuSign Documents to Use with Omniscripts](#) and [Configure a Salesforce Org for DocuSign Integration](#).
- If you're using the new designers, install a managed package in your org.

1. From the App Launcher, in the Search apps and items box, enter *DocuSign*, then click **Omnistudio DocuSign Setup**.
2. Click **Fetch DocuSign Templates**.

Names, last updated dates, and descriptions of templates are listed.

What's Next: [Map Fields from the Omniscript to the DocuSign Template](#)

Map Fields from the Omniscript to the DocuSign Template

To populate a DocuSign template, you define Omniscript data as input to an Omnistudio Data Mapper Transform, which uses that data to populate the template. You can construct the Omniscript data manually or copy it from the Omniscript.

Before you begin: [Get DocuSign Templates from Omnistudio DocuSign Setup](#)

You can construct the data manually. For example, if the Omniscript data is from text input fields in Step1, you can create JSON like this with null values:

```
{  
  "Step1": {  
    "Account": null,  
    "Email": null,  
    "Name": null,  
    "Phone": null  
  }  
}
```

If the Omniscript data is from a Type Ahead block in Step1, you can create JSON like this with null values:

```
{  
  "Step1": {  
    "TypeAhead1": {  
      "DRExtract1": {  
        "Account": null,  
        "Email": null,  
        "Name": null,  
        "Phone": null  
      }  
    }  
  }  
}
```

```
        "Email": null,  
        "Name": null,  
        "Phone": null  
    }  
}  
}  
}  
}
```

Or you can get the data from the Omniscript. Here's how to get it from Preview:

1. Click **Preview**.
2. Go to the step with the data.
3. Click the copy icon in the **Data JSON** tab.
4. Paste this JSON in a text editor, make the values null, and save it so you can copy it into the Data Mapper Transform.

For example:

```
{  
    "timeStamp": null,  
    "userProfile": null,  
    "userTimeZoneName": null,  
    "userTimeZone": null,  
    "userCurrencyCode": null,  
    "userName": null,  
    "userId": null,  
    "omniProcessId": null,  
    "localTimeZoneName": null,  
    "Step1": {  
        "Account": null,  
        "Email": null,  
        "Name": null,  
        "Phone": null  
    }  
}
```

What's next: [Populate a Docusign Template for Omniscripts with a Data Mapper Transform](#)

Populate a Docusign Template for Omniscripts with a Data Mapper Transform

Map Omniscript data to a Docusign template using a Data Mapper Transform.

Before you begin: [Map Fields from the Omniscript to the DocuSign Template](#)

1. From the Omnistudio Data Mappers tab, click **New**.
2. Name the Data Mapper.
3. For Interface Type, select **Transform**.
4. Configure these settings:
 - **Input Type:** JSON
 - **Output Type:** Docusign
 - **Target Output Docusign Template Id:** Choose the template that your Omniscript is designed to populate.
5. Click **Save**.
6. Copy the data JSON from your Omniscript and paste it into the **Input JSON** pane in the Data Mapper Designer.
7. On the **Outputs** tab, click **Quick Match**.

The mappings look something like this:

Input Mappings (from Omniscript)	Output Mappings (from DocuSign Template)
Step1:Account	tabs:textTab:Account
Step1:Email	tabs:textTab:Email
Step1:Name	tabs:textTab:Name
Step1:Phone	tabs:textTab:Phone

8. Click **Auto Match**, or use **Pair** to create individual mappings.



Note If you have updated your Docusign template, you must fetch the Docusign templates to update the input fields. For more information on fetching templates, see [Get DocuSign Templates from Omnistudio DocuSign Setup](#).

Customize Omniscript Behaviors, Style, and Elements

To meet your business needs and improve user experience, you can customize Omniscripts for their style and appearance and for their actions. Apply custom styling via static resources, global stylesheets, or SLDS design tokens. Modify Omniscripts to conditionally hide or show elements and errors and to trigger platform events. For more complex customization, you can extend Omniscript elements to add custom behavior and styling. Finally, you can create custom Lightning web components (LWCs) to add custom HTML, JavaScript, and CSS.

Custom Styles for Omniscripts

Add custom styling to Omniscripts by using static resources, overriding global stylesheets, or using SLDS design tokens. View the behavior of each customization option in this page's table, and choose the solution that meets your requirements.

Modify Omniscript UI Behavior

You modify UI behaviors in an Omniscript to hide or show elements and errors conditionally. You can also trigger Salesforce platform events. When previewing large Omniscripts, you can hide the

userProfile node.

[Customize Omniscript Elements](#)

Add custom behavior and styling to an Omniscript element by extending the functionality of an Omniscript element. By extending the Omniscript component, you modify its properties and add new properties. Then, replace or override the Omniscript element's Lightning web component with the customized component in the designer to use the customized component at run time.

[Create a Custom Lightning Web Component for Omniscript](#)

Add custom HTML, JavaScript, and CSS to an Omniscript by creating a custom Lightning web component. The Custom LWC element enables custom components to either interact with the Omniscript or to act as a standalone component. Standalone components do not interact with the Omniscript or the Omniscript's data JSON.

[Omniscript ReadMe Reference](#)

This page lists the Lightning web components ReadMes available for Omniscripts. Extend Lightning web components to add custom behavior and styling.

Custom Styles for Omniscripts

Add custom styling to Omniscripts by using static resources, overriding global stylesheets, or using SLDS design tokens. View the behavior of each customization option in this page's table, and choose the solution that meets your requirements.

[Apply Custom Styling to Omniscripts with Static Resources](#)

To apply CSS changes to an Omniscript using SLDS styling, refer to a CSS file static resource in the Styling Options section of the Omniscript designer. For example, a CSS sheet can change the text color for every Number element in an Omniscript without using a custom Lightning web component to override the Number element.

[Customize SLDS Design Tokens in Omniscript](#)

Modify the appearance of an Omniscript by overriding Salesforce's SLDS Design tokens. Omniscript elements use SLDS Design Tokens for styling. Override tokens that have global access to modify the appearance of your Omniscript.

Apply Custom Styling to Omniscripts with Static Resources

To apply CSS changes to an Omniscript using SLDS styling, refer to a CSS file static resource in the Styling Options section of the Omniscript designer. For example, a CSS sheet can change the text color for every Number element in an Omniscript without using a custom Lightning web component to override the Number element.



Warning Use custom CSS cautiously and avoid targeting DOM elements in components you don't own. Changes to a component's internal structure might break your CSS. Salesforce may update component implementations at any time, and Salesforce Support can't assist with custom CSS issues.

1. After you create a custom style sheet, add the CSS file as a static resource.

See [Using Static Resources](#).

For information about styling systems, see [Light DOM, CSS](#), and [Styling Hooks](#).

For example, `customstyle.css` is uploaded as a static resource named `customstyle`.

2. In the Omniscript designer, from Setup, click **Styling Options**.
3. In the **Custom Lightning Stylesheet File Name** or **Custom Newport Stylesheet File Name** field, enter the name of the static resource that stores your custom styling.

For example, the static source `customstyle` refers to the uploaded `customstyle.css` file.

4. If needed, display custom styling for right-to-left languages.
 - a. In Setup, click **Edit as JSON**.
 - b. To display the styling for right-to-left languages, add the name of the static resource as a value to `lightningRtl`.

If you're updating an existing Omniscript, add `lightningRtl` manually.

```
"stylesheet": {  
    "lightning": "",  
    "lightningRtl": "myRtlLightningStyles"  
}
```

5. To view your custom styling, activate and preview the Omniscript.

When you preview an Omniscript with a child Omniscript with a custom CSS, the content for the child Omniscript doesn't show the custom styling.

Customize SLDS Design Tokens in Omniscript

Modify the appearance of an Omniscript by overriding Salesforce's SLDS Design tokens. Omniscript elements use SLDS Design Tokens for styling. Override tokens that have global access to modify the appearance of your Omniscript.

 **Note** Design Tokens are not supported in Safari browsers.

Before You Begin

- View the Design Tokens that have global access. See [Design Tokens](#).
- Preview and inspect an Omniscript in a developer console to view the current tokens present in the Omniscript.
- You can now add an Omniscript on a Lightning Web Runtime (LWR) Experience Cloud site. However, review [Considerations for Using Omniscripts with Lightning Web Runtime Sites](#) before you do this.

1. In your Omniscript's Setup panel, click **Styling Options** section.
2. In the **Lightning Design System Design Tokens** field, enter tokens by applying these changes:
 - a. Remove the dashes and apply camelcase to the token. For example, the token `$spacing-xx-small` must be `$spacingXxSmall`.

- b. Replace the token's \$ symbol with --lwc-.

```
--lwc-spacingXxSmall
```

- c. Set the token's value, and end the line for each token using a ;.

```
--lwc-spacingXxSmall: 10rem;  
--lwc-spacingSmall: 5rem;  
--lwc-spacingMedium: 2rem;
```

- d. If you need to use your Omniscript on Experience Cloud pages, you must add design tokens specific to those pages. Note that design tokens for Experience Cloud sites differ based on whether you're using an Aura-based site or an LWR site. You can, however, add all tokens together. The page referencing your Omniscript picks up the tokens it needs to style your Omniscript.

```
// Note: Style for Aura-based Experience Cloud site  
--lwc-spacingXxSmall: 10rem;  
--lwc-spacingSmall: 5rem;  
--lwc-spacingMedium: 2rem;  
// Note: Style for a Lightning Web Runtime (LWR) site  
--dpx-g-spacing-xxsmall: 10rem;  
--dpx-g-spacing-small: 5rem;  
--dpx-g-spacing-medium: 2rem;  
// Note: Style for a Lightning page  
--slds-g-spacing-1: 10rem;  
--slds-g-spacing-2: 5rem;  
--slds-g-spacing-3: 2rem;
```

Modify Omniscript UI Behavior

You modify UI behaviors in an Omniscript to hide or show elements and errors conditionally. You can also trigger Salesforce platform events. When previewing large Omniscripts, you can hide the userProfile node.

Conditionally Display Omniscript Elements

With the Conditional View property, you can hide an element or a group of elements in a block or step based on certain conditions. Every step, block, or element supports at least one conditional view.

Elements contain up to three conditional view options depending on the type of element. For example, an action outside of a step doesn't have a Read-Only option since the action isn't visible.

Hide Omniscript Elements

Hide elements and child elements using the Hide property available on Omniscript Input Components, Blocks, Steps, Groups, Formulas, and Aggregate elements.

Fire Platform Events from Omniscripts

Fire Salesforce platform events from the context of an Omniscript.

Hide the userProfile Node in Omniscript Preview

To improve the performance of Omniscript previews, you can hide the userProfile node. This helps previews load faster, especially for larger Omniscripts.

Conditionally Display Omniscript Elements

With the Conditional View property, you can hide an element or a group of elements in a block or step based on certain conditions. Every step, block, or element supports at least one conditional view. Elements contain up to three conditional view options depending on the type of element. For example, an action outside of a step doesn't have a Read-Only option since the action isn't visible.

When you hide an element in an Omniscript, Omnistudio sets the contents of the element to null in the data JSON. Consider this behavior if your Omniscript references the hidden field. For example, suppose a Set Error element assesses whether a Text element is filled before proceeding to the next page. If the Text element is conditionally hidden, the Text element value is null, and the error is set.

1. From an element's properties, expand **Conditional View** section.
2. Select a **Condition Type** from the dropdown menu.

Condition	Description
Show Element if True	The element renders only if the conditional is true. If the conditional is false, the element is hidden from the UI. For blocks, this condition type is the only one available.
Disable Read Only if True	Disables Read Only on an element if the conditional is true. If the value is false, the element becomes Read Only.
Set Element to Required if True	The element becomes required if the condition is true.



Note If a condition type isn't listed for an element, it isn't supported. Adding it using the JSON editor doesn't work.

3. Choose the logical operator for the group (AND or OR). The AND operator requires that all conditions are met. The OR operator requires that at least one condition is met.
4. Click **Add Condition**.
5. In the left operand, enter the name of the element in plain text.



Note If the element is repeatable, enter the name of the element followed by `| n`, where n is the nth repeated element. For example, if Age is repeated, Age|2 refers to the second instance of the

repeated element.

6. In the right operand, enter one of these syntax types:
 - If the element type is Text, enter the string in plain text.
 - If the element type is Checkbox or Disclosure, enter **true** or **false**.
 - If the element type is Number, enter a number.
 - If the element type is Select or Radio, type in the name of the option in plain text—for example, **Yes** or **No**.
 - If the element type is Multi-Select, enter a semicolon-separated string. For example, **Ind | ; | Small**.
 - If the element type is Date, Time, or Date/Time, enter a value in **ISO format**:
 - Date–2014-10-01T07:00:00.000Z (Wed Oct 01 2014 00:00:00 GMT-0700 (PDT))
 - Date/Time–1970-01-02T06:19:00.000Z (Tue Feb 17 2015 22:20:03 GMT-0800 (PST))
 - Time–2015-02-21T00:09:40.270Z (Thu Jan 01 1970 22:19:00 GMT-0800 (PST))
 - Date–2014-10-01T07:00:00.000Z (Wed Oct 01 2014 00:00:00 GMT-0700 (PDT))
 - Date/Time–1970-01-02T06:19:00.000Z (Tue Feb 17 2015 22:20:03 GMT-0800 (PST))
 - Time–2015-02-21T00:09:40.270Z (Thu Jan 01 1970 22:19:00 GMT-0800 (PST))
 - The right operand supports merge fields, for example `Name = %contactLastName%`.
7. If you're using the designer on a managed package, click the **Hide Conditional Elements** checkbox to hide all of the conditional elements in the Omniscript. The conditional elements are marked with a yellow eye icon.

Running Validation with onChange

Omniscripts run validation when a user clicks out of a field by using the `onBlur` function.

1. In the Setup properties, click **Edit as JSON**.
2. Add the property `"commitOnChange": true`.
3. Preview the behavior.



Note In LWC Omniscripts, the `onChange` behavior runs after a half-second delay.

Hide Omniscript Elements

Hide elements and child elements using the `Hide` property available on Omniscript Input Components, Blocks, Steps, Groups, Formulas, and Aggregate elements.

When you hide an element in an Omniscript, Omnistudio sets the contents of the element to null in the data JSON. Consider this behavior if your Omniscript references the hidden field.

1. From the element properties panel, click **Edit as JSON**.
2. Set the `"hide"` property's value to `"true"`.

Fire Platform Events from Omniscripts

Fire Salesforce platform events from the context of an Omniscript.

1. Go to **Setup** and, under **Develop**, click **Platform Events**.
2. On the Platform Events page, click **New Platform Event**.
3. On the New Platform Event page, define the platform event, specifying the fields for the data that you want to associate with the event.
4. To fire the event, create an Omnistudio Data Mapper Load that populates the platform event fields with event-specific data.
5. In the Omniscript, compose the data JSON containing the data required by the Data Mapper Load.
6. Add a Data Mapper Load action to the Omniscript, specifying the Data Mapper that fires the platform event.
7. Verify that your platform event fired correctly, using one of the programmatic methods described in the [Salesforce developer documentation](#).

Hide the userProfile Node in Omniscript Preview

To improve the performance of Omniscript previews, you can hide the userProfile node. This helps previews load faster, especially for larger Omniscripts.

1. In Salesforce Setup, type *Omni Interaction Configuration* in the Quick Find box and then click **Omni Interaction Configuration**.
2. Click **New Omni Interaction Configuration**.
3. In the Label field, type *SkipUserProfileOnOSLoad*.
4. In the Value field, type *true*.
5. Save.

Customize Omniscript Elements

Add custom behavior and styling to an Omniscript element by extending the functionality of an Omniscript element. By extending the Omniscript component, you modify its properties and add new properties. Then, replace or override the Omniscript element's Lightning web component with the customized component in the designer to use the customized component at run time.

Download the extendable Omniscript components from Salesforce, customize them, and deploy them to your org. After deployment, you can override an element with the custom component in the Omniscript.

Extend and Override an Omniscript Element's Lightning Web Component

A Lightning web component provides the HTML, JavaScript, XML, and CSS files for an Omniscript element. Modify these files to add custom behavior and styling to a Lightning web component. For example, a Text element has a custom component named `omniscriptText`. Customize the Omniscript element's functionality and styling by extending the custom component. Then, override the Omniscript's element with the custom Lightning web component in the Omniscript designer.

The custom Lightning web component must extend the element's property because Omniscript passes

in properties specific to the original element that the custom element is overriding.

To override individual elements in an Omniscript, add the custom component by using LWC Component Override in the element property.

To override all Omniscript elements of a type, map the element to the custom component in Setup. Use mapping for the Step Chart, Save for Later Acknowledge, and Modal elements.

Customizable and Uncustomizable Omniscript Elements

You can extend most, but not all, Omniscript components.

Set Up Your Environment to Customize Omniscript Elements

Set up your developer environment by downloading the Omniscript elements that you want to customize and by deploying these elements in a Salesforce org.

Extend an Omniscript Element's Lightning Web Component

Add custom behavior and styling to an Omniscript element while maintaining its core functionality. Replace an Omniscript element's Lightning web component with a component that re-creates its properties and adds new ones.

Customizable and Uncustomizable Omniscript Elements

You can extend most, but not all, Omniscript components.

Customizable Omniscript Elements

You can extend these Omniscript components:

- Action Elements
 - omniscriptCalculationAction
 - omniscriptCancelAction
 - omniscriptDeleteAction
 - omniscriptDocuSignEnvelopeAction
 - omniscriptDocuSignSignatureAction
 - omniscriptDrExtractAction
 - omniscriptDrPostAction
 - omniscriptDrTransformAction
 - omniscriptDrTurboAction
 - omniscriptEmailAction
 - omniscriptHttpAction
 - omniscriptIpAction
 - omniscriptMatrixAction
 - omniscriptNavigateAction
 - omniscriptPdfAction
 - omniscriptRemoteAction
 - omniscriptSetErrors
 - omniscriptSetValue

- Miscellaneous Elements
 - omniscriptModal
 - omniscriptSaveForLaterAcknowledge
 - omniscriptStepChart
- Display Elements
 - omniscriptLineBreak
 - omniscriptTextBlock
- Functions
 - omniscriptAggregate
 - omniscriptFormula
 - omniscriptMessaging
- Group Elements
 - omniscriptActionBlock
 - omniscriptBlock
 - omniscriptEditBlock
 - omniscriptEditBlockLabel
 - omniscriptEditBlockNew
 - omniscriptRadioGroup
 - omniscriptStep
 - omniscriptTypeaheadBlock
- Input Elements
 - omniscriptCheckbox
 - omniscriptCurrency
 - omniscriptDate
 - omniscriptDateTime
 - omniscriptDisclosure
 - omniscriptEmail
 - omniscriptFile
 - omniscriptImage
 - omniscriptLookup
 - omniscriptMultiselect
 - omniscriptNumber
 - omniscriptPassword
 - omniscriptRadio
 - omniscriptRange
 - omniscriptSelect
 - omniscriptTelephone
 - omniscriptText
 - omniscriptTextarea
 - omniscriptTime
 - omniscriptUrl

Omniscript Elements that can't be Customized

You can't customize these Omniscript elements.:

- omniscriptAtomicElement
- omniscriptBaseAction
- omniscriptBaseElement
- omniscriptCustomLabels
- omniscriptCustomLwc
- omniscriptEditBlockUtils
- omniscriptEditBlockWrapper
- omniscriptFormattedRichText
- omniscriptGroupElement
- omniscriptInternalUtils
- omniscriptOptionsMixin
- omniscriptPlacesTypeahead
- omniscriptRestApi
- omniscriptStepChartItems
- omniscriptTrackingServiceUtils
- omniscriptUtils
- omniscriptValidation
- omniscriptActionUtilsForCore
- omniscriptActionUtilsForCoreExample

Set Up Your Environment to Customize Omniscript Elements

Set up your developer environment by downloading the Omniscript elements that you want to customize and by deploying these elements in a Salesforce org.

To customize Omniscript elements, use Salesforce CLI or Salesforce Extensions for Visual Studio Code.

- [Install Salesforce CLI](#).
- [Set Up Visual Studio Code](#).

1. Request an NPM repository access key from Salesforce Customer Support.
2. Create an `.npmrc` file in the directory where you want to install the Omniscript customization npm package, and set `_auth` equal to your NPM repository access key.

```
always-auth=true
registry=https://repo.vlocity.com/repository/npm-public/
//repo.vlocity.com/repository/npm-public/:_auth="auth key"
```

3. From the command-line interface, install the Omniscript customization npm package with the custom components.

```
npm install @omnistudio/omniscript_customization@250.0.0
```

See [npm documentation](#).

After successful installation of the Omniscript customization npm package, verify the following:

- These files exist in the directory you created in Step 2: `node_modules`, `package-lock.json`, `package.json`.
- The `node_modules/@omnistudio/omniscript_customization` directory contains these files: `CHANGELOG.txt`, `LICENSE.txt`, `labels`, `lwc`, `messageChannels`, `package.json`, and `package.xml`.

4. Create a Salesforce DX Project.

See [Set Up Visual Studio Code](#) and [Create a Salesforce DX Project](#).

5. Copy the `labels`, `lwc`, and `messageChannels` directories and the `package.xml` file from the Omniscript customization npm package to the DX project, matching the directories in the package.

Name	Date Modified	Size	Kind
<code>force-app</code>	Today at 9:43 AM	--	Folder
<code>main</code>	Today at 9:43 AM	--	Folder
<code>default</code>	Today at 9:44 AM	--	Folder
<code>applications</code>	Today at 9:41 AM	--	Folder
<code>aura</code>	Today at 9:41 AM	--	Folder
<code>classes</code>	Today at 9:41 AM	--	Folder
<code>contentassets</code>	Today at 9:41 AM	--	Folder
<code>flexipages</code>	Today at 9:41 AM	--	Folder
<code>layouts</code>	Today at 9:41 AM	--	Folder
<code>lwc</code>	Today at 9:45 AM	--	Folder
<code>action</code>	May 28, 2024 at 4:40 PM	--	Folder
<code>actionUtility</code>	May 28, 2024 at 4:20 PM	--	Folder
<code>alert</code>	May 28, 2024 at 4:40 PM	--	Folder
<code>asyncUtils</code>	May 28, 2024 at 4:20 PM	--	Folder
<code>baseFlexElementMixin</code>	May 28, 2024 at 4:20 PM	--	Folder
<code>buffer</code>	May 28, 2024 at 4:20 PM	--	Folder
<code>button</code>	May 28, 2024 at 4:40 PM	--	Folder
<code>changeCase</code>	May 28, 2024 at 4:40 PM	--	Folder
<code>checkboxGroup</code>	May 28, 2024 at 4:40 PM	--	Folder
<code>checkboxImageGroup</code>	May 28, 2024 at 4:40 PM	--	Folder
<code>combobox</code>	May 28, 2024 at 4:40 PM	--	Folder
<code>currencyjs</code>	May 28, 2024 at 4:20 PM	--	Folder
<code>customEvents</code>	May 28, 2024 at 4:40 PM	--	Folder

6. Connect to your Salesforce org by using the CLI or Visual Studio Code.

- To log in using CLI, run the `org login web` command.
See [Authorize an Org Using a Browser](#).
- In Visual Studio Code, type Command-Shift-P (Mac) or Ctrl-shift-P (Windows) to open the command palette, and then enter `SFDX: Authorize an Org` and select **Custom**.

7. Deploy your DX project to your org:

- To deploy the project by using CLI, run the `project deploy start` command.
- To deploy the project by using Visual Studio Code, right-click each folder and select **Deploy This Source to Org**.

Deploy the folders in this order: `/labels`, `/messageChannels`, and `/lwc`. The deploy order is important because only one folder is deployed at a time and each folder has dependencies on the other folders.

See [Deploy Source to a Non-Source-Tracked Org](#).

Extend an Omniscript Element's Lightning Web Component

Add custom behavior and styling to an Omniscript element while maintaining its core functionality. Replace an Omniscript element's Lightning web component with a component that re-creates its properties and adds new ones.

Before you begin:

See [Set Up Your Environment for Customizing Omniscript Elements](#).

A Lightning web component provides the HTML, JavaScript, XML, and CSS files for an Omniscript element. For example, a Text element has an extendable component named omniscriptText. For information about creating custom Lightning web components for Omniscripts that aren't element extensions, see [Create a Custom Lightning Web Component for Omniscript](#).

-  **Note** Custom Lightning web components built outside of the managed package can't use any Salesforce Lightning web component that uses Salesforce resources or affects the component at run time. See [Salesforce Modules](#). If you turn off the Managed Package Runtime setting in Setup, we recommend that you don't extend Lightning web components such as Block or TypeAhead. Extending these components can introduce incompatibilities between Omnistudio and Omnistudio for Managed Packages.

1. In the `lwc` folder of your project, create a subfolder named `lwcName`.
For example, create a subfolder named `customText`.
2. In the new subfolder, create a file named `lwcName.js`.
For example, create a file named `customText.js`.
3. Paste this code into the JavaScript file. Substitute the name of your new component and the namespace of the Omniscript element's LWC.
This example customizes `omniscriptText` with a custom component called `customText`. For Omnistudio, the namespace is `c`.

```
import OmniscriptText from 'c/omniscriptText';
import tmpl from './customText.html';
export default class customText extends OmniscriptText {
    render() {
        return tmpl;
    }
}
```

4. In the subfolder, create a file named `lwcName.js-meta.xml`.
For example, create a file named `customText.js-meta.xml`.
5. Paste this code into the XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata" fq  
n="omniscript">  
    <apiVersion>60.0</apiVersion>  
    <isExposed>false</isExposed>  
    <masterLabel>Custom Text</masterLabel>  
</LightningComponentBundle>
```

6. In the unzipped file structure, locate the folder of the Lightning web component for the Omniscript element that you want to extend.

For example, locate the folder named `omniscriptText`.

7. Edit the HTML file.

- a. Right-click the HTML file and select **Open With Visual Studio Code**.
- b. Copy the content of the HTML file.
- c. Create a file in the custom component subfolder named `lwcName.html`.
For example, create a file named `customText.html`.
- d. Paste the copied content of the HTML file for the Lightning web component into the new HTML file.

8. To deploy the custom component to your Salesforce org, right-click the subfolder and select **Deploy Source to Org**.

For more information, see [Deploy Lightning Web Components](#).

9. Add the custom component to an Omniscript in one of these ways:

- Override an individual element by using **LWC Component Override**.
- Map all Omniscript elements of the type you're extending to the custom component.

Override an Omniscript Element with a Custom Lightning Web Component

Override an Omniscript element with custom Lightning web components (LWCs) to add custom behavior and styling to the Omniscript element.

Override all Elements of an Omniscript Element Type with a Custom Lightning Web Component

To override all elements of a type in an Omniscript, map the element to the custom Lightning web component. You can't directly drag these components to the Omniscript canvas and map them to their custom Lightning web component. These components include Step Chart, Save for Later, Acknowledge, and Modal.

Override an Omniscript Element with a Custom Lightning Web Component

Override an Omniscript element with custom Lightning web components (LWCs) to add custom behavior and styling to the Omniscript element.

Before You Begin

- Ensure your custom LWC is deployed with `<isExposed>true</isExposed>` in its `.xml` file.
- Review the base Omniscript LWC in the `omni-base-templates` folder to identify the correct component to extend.

To override an Omniscript element:

1. From the elements panel, drag an element that you want to override to the Omniscript canvas.
For example, drag a Text element.
2. In the Properties panel, enter the name of your custom component in **LWC Component Override**.
For example, enter the name `customText`.
3. In the Setup panel, click **Edit Properties As JSON** and add this property: `"useCustomLMS": true`.

This allows your custom components to communicate with the Omniscript standard runtime components.

4. To close the property editor, click **Close the JSON Editor**.
5. Save and activate the Omniscript.
6. To preview the Omniscript, click **Preview**.

Points to consider:

- Your custom LWC should extend the correct base Omniscript component. For example, `OmniscriptDate`.
- Override specific methods in the parent component as needed. For example, `setElementFormattedValue()`.
- After overriding, test the Omniscript in **Preview** mode to verify styling and behavior changes.

Override all Elements of an Omniscript Element Type with a Custom Lightning Web Component

To override all elements of a type in an Omniscript, map the element to the custom Lightning web component. You can't directly drag these components to the Omniscript canvas and map them to their custom Lightning web component. These components include Step Chart, Save for Later Acknowledge, and Modal.

1. Create a custom LWC that extends the LWC and add the custom LWC to your Salesforce org. Here are the extendable components:
 - Step Chart: `omniscriptStepChart`
 - Save for Later: `omniscriptSaveForLaterAcknowledge`
 - Modal: `omniscriptModal`
2. From the Omniscript Setup, scroll to the Element Type to the LWC Component Mapping property and add mappings.
 - a. Enter the element type that you want to customize.
The element types include:
 - Step Chart: `StepChart`
 - Modal: `Modal`
 - Save for Later: `SaveForLaterAcknowledge`
 - b. In the Lightning Web Component field, select your custom Lightning web component for the element type.
 - c. Click **Add Mapping** to map another element type.

Element Type To LWC Component Mapping		
Element Type	Lightning Web Component	
StepChart	customomniscritStepC	
Modal	customomniscritModa	
SaveForLaterAcknc	customomniscritSavef	

3. From the Setup panel, click **Edit Properties As JSON**, and add this property: `"useCustomLMS": true`.

This property enables customized components to communicate with the Omniscript standard runtime components.

4. To close the property editor, click **Close JSON Editor**.
5. Activate the Omniscript.
6. Click **Preview**.

Create a Custom Lightning Web Component for Omniscript

Add custom HTML, JavaScript, and CSS to an Omniscript by creating a custom Lightning web component. The Custom LWC element enables custom components to either interact with the Omniscript or to act as a standalone component. Standalone components do not interact with the Omniscript or the Omniscript's data JSON.

For information on using the Lightning Web Component framework, see [Set Up Lightning Web Components](#).

Note Before previewing an Omniscript that uses custom Lightning web components, you must activate the Omniscript. If the Omniscript isn't active, custom Lightning web components don't render in the preview. If you deploy changes to a custom Lightning web component that's present in an active Omniscript, the Omniscript displays the changes.

To create a custom Lightning web component, review these component types:

- OmniscriptBaseMixin component: Enable a custom component to interact with an Omniscript by extending the OmniscriptBaseMixin component. See [Extend the OmniscriptBaseMixin Component](#).
- Standalone component: A custom standalone component. The component can't extend the OmniscriptBaseMixin component or an Omniscript element Lightning web component. See [Create a Standalone Custom Lightning Web Component](#).

After you create a custom Lightning web component, add it to an Omniscript by using the Custom LWC input element. See [Custom LWC Element](#).

[Requirements for Custom Lightning Web Components for Omniscripts](#)

Review the requirements for making a custom Lightning web component compatible with Omnistudio Lightning web components.

[Extend the OmniscriptBaseMixin Component](#)

Enable a custom Lightning web component to interact with an Omniscript by extending the OmniscriptBaseMixin component. The OmniscriptBaseMixin includes methods to update an Omniscript's data JSON, pass parameters, and more.

[Create a Standalone Custom Lightning Web Component](#)

Enable custom Lightning web components to act independently from an Omniscript by adding it to the Omniscript as a standalone component. Standalone custom Lightning web components cannot extend any Omniscript element component or the OmniscriptBaseMixin component. The standalone component supports custom functionalities, but cannot interact with an Omniscript or the Omniscript's data JSON.

[Communicate from Omniscript to a Lightning Web Component](#)

Send data from Omniscript actions and steps to other Lightning web components using the Publish and Subscribe property.

[Make Remote Calls Within Omniscripts from Lightning Web Components](#)

In the standard runtime, make remote calls from a customized Omniscript component or a custom Lightning web component.

[Add Validation to a Custom Lightning Web Component in Omniscript](#)

Add validation to custom Lightning web components that extend an Omniscript element's component by using Vlocity's built-in validation methods.

Requirements for Custom Lightning Web Components for Omniscripts

Review the requirements for making a custom Lightning web component compatible with Omnistudio Lightning web components.

- Custom Lightning web components extending the OmniscriptBaseMixin component cannot extend or override an Omniscript element Lightning web component.
- Custom Lightning web components extending the OmniscriptBaseMixin must use the **Custom LWC** input element in Omniscript. For more information about using the Custom LWC element, see [Custom LWC Element](#).
- Custom Lightning Web Components don't throw errors unless **Debug Mode** is enabled. For more information, see [Debug Lightning Web Components](#).
- A standalone component cannot extend an Omniscript element's Lightning web component.
- A standalone component cannot extend the OmniscriptBaseMixin component.
- A standalone component can only interact with Omniscript through the omniscripttaggregate event.
- To make the custom Lightning web component compatible with Omnistudio Lightning web components, set the **isExposed** metadata tag to `true` in your XML configuration file:

```
<isExposed>True</isExposed>
```

- To prevent JSON validation errors when passing data from Custom Lightning web components to Omnistudio Lightning web components, precede double quotes in the data with the backslash (\)

escape character. Single quotes don't need to be escaped.

- Install the package with the custom components. See [Set Up Your Environment to Customize Omniscript Elements](#).
- View all of the component's properties, attributes, and methods in the [Omniscript ReadMe Reference](#).

Extend the OmniscriptBaseMixin Component

Enable a custom Lightning web component to interact with an Omniscript by extending the OmniscriptBaseMixin component. The OmniscriptBaseMixin includes methods to update an Omniscript's data JSON, pass parameters, and more.

After extending the OmniscriptBaseMixin component, add a custom LWC input element to an Omniscript and specify the name of a component that extends the OmniscriptBaseMixin. See [Custom LWC Element](#).

1. Extend the Component

Learn how a custom Lightning web component extends the OmniscriptBaseMixin component to wrap a Salesforce Lightning Element.

2. Validate the Component

Learn how to add validation to the OmniscriptBaseMixin component by redefining the `checkValidity()` function.

3. Navigation Between Steps

Learn which mixins you can set on the steps of an Omniscript so that users can navigate between those steps.

4. Data Save Inside of a Custom Component

When the custom component doesn't exist in the DOM, store the data in a state. To store data in a state, use the `omniSaveState()` function. The `omniSaveState` function accepts three arguments: a data object, a key, and a boolean.

5. Retrieval of Custom Component Data from an Omniscript

Use the `omniGetSaveState()` function to retrieve custom component data from an Omniscript.

6. Clear a Saved State from a Previous Step

When a Custom LWC's saved state relies on data from a previous step, you can conditionally clear the saved state of the Custom LWC whenever a user navigates to a previous step. After configuration, the Step element passes a boolean flag to the custom LWC. The code in your Custom LWC must use the boolean to determine whether to instantiate a new saved state or continue using the existing saved state.

7. Access the Layout of the OmniscriptBaseMixin Component

Call the layout for your Custom LWC by passing the `data-omni-layout` parameter to the `getAttribute` method.

8. Remote Calls

Enable users to make remote calls from a Custom LWC by using the `omniRemoteCall()` method.

9. Data JSON Update

Update the custom Lightning web component element, map responses to the Omniscript's data JSON, and create and map data inside the custom Lightning Web component element.

See Also

[Create a Custom Lightning Web Component for Omniscript](#)

[Create a Standalone Custom Lightning Web Component](#)

Extend the Component

Learn how a custom Lightning web component extends the `OmniscriptBaseMixin` component to wrap a Salesforce Lightning Element.

See this code example for extending the `OmniscriptBaseMixin` component:

```
import tmpl from './customLightningElement.html';
import { LightningElement } from 'lwc';
import { OmniscriptBaseMixin } from 'c/omniscriptBaseMixin';

export default class customLightningElementWithMixin extends OmniscriptBaseMix
in(LightningElement) {
    handleBlur(evt) {
        this.omniUpdateDataJson(evt.target.value);
    }

    render() {
        return tmpl;
    }
}
```

Validate the Component

Learn how to add validation to the `OmniscriptBaseMixin` component by redefining the `checkValidity()` function.

This code example contains a `checkValidity()` function that returns false when the sum is less than zero.

```
import { api } from 'lwc';
import { OmniscriptBaseMixin } from 'c/omniscriptBaseMixin';
export default class MyCustomLwc extends OmniscriptBaseMixin(LightningElement)
{
    sum = 0;

    // defining custom validation conditions
    @api checkValidity() {
        return this.sum < 100;
    }
}
```

Navigation Between Steps

Learn which mixins you can set on the steps of an Omniscript so that users can navigate between those steps.

omniNextStep()

Advances users to the next step in the Omniscript.

JS Example	HTML Example
<pre>nextButton(evt) { if (evt) { this.omniNextStep(); } }</pre>	<pre><button onclick={nextButton}>Custom Next Button </button></pre>

omniPrevStep()

Takes users to the previous step in the Omniscript.

JS Example	HTML Example
<pre>prevButton(evt) { if (evt) { this.omniPrevStep(); } }</pre>	<pre><button onclick={prevButton}>Custom Prev Button </button></pre>

omniNavigateTo(step)

Takes users to a specific step in the Omniscript by passing an argument. The argument passed into the function can be a number that refers to the step's index in the Omniscript or a step's element name passed as a hardcoded string. The index for the current step is accessible using the property

`this.omniScriptHeaderDef.asIndex`.

 **Note** You can't auto-advance to a step more than one step ahead.

JS Example	HTML Example
<pre>goToStep(evt) {</pre>	<pre><button onclick={goToStep}>Custom St</pre>

JS Example	HTML Example
<pre>if (evt) { this.omniNavigateTo(this.omniScriptHeaderDef.asListIndex - 2); }</pre>	<pre>ep Button </button></pre>

Data Save Inside of a Custom Component

When the custom component doesn't exist in the DOM, store the data in a state. To store data in a state, use the `omniSaveState()` function. The `omniSaveState` function accepts three arguments: a data object, a key, and a boolean.

Method: `omniSaveState(input, key, usePubSub=false)` :

Arguments:

- `input` : The data to be saved.

```
let mySaveState = {"my":"data"};
this.omniSaveState(mySaveState);
```

- `key` : (Optional) A string value that the object parameter maps to as a key-value pair. If a key isn't sent as a parameter, the key defaults to the name of the component.

```
let mySaveState = {"my":"data"};
let key = 'customLwcKey';
this.omniSaveState(mySaveState, key);
```

- `usePubSub` : (Optional) Set the boolean to true to store data for use in a pubsub pattern. For more information on pubsub patterns, see [pubsub events](#). When left empty or set to false, events can use the data. For more information on events, see [LWC Events](#). When `omniSaveState` is in a `disconnectedCallback`, `usePubSub` must be set to `true` to pass the event in the pubsub instead of event bubbling.

Optionally save the state within a disconnected callback. Saving the state within a disconnected callback enables the state of the element to save when users leave the element automatically. In disconnected callbacks, the `usePubSub` argument must be set to `true`. Events in disconnected callbacks can only be passed using pubsub.

 **Example** `omniSaveState` being called in a `disconnectedCallback`:

```
disconnectedCallback() {
```

```
let mySaveState = {"my":"data", "here" : 8};  
let key = 'customLwcKey';  
let usePubSub = true;  
this.omniSaveState(mySaveState, key, usePubSub);  
}
```

Retrieval of Custom Component Data from an Omniscript

Use the `omniGetSaveState()` function to retrieve custom component data from an Omniscript.

`omniGetSaveState(key)` : The `omniGetSaveState` accepts a key as an optional argument. If a key is not sent, the key defaults to the name of the custom component. The key stores the data from the `omniSaveState()` function.

Example With a key argument:

```
let key = 'customLwcKey';  
const state = this.omniGetSaveState(key);
```

With no key:

```
// assumes the data is inside of the same custom component  
const state = this.omniGetSaveState()
```

Clear a Saved State from a Previous Step

When a Custom LWC's saved state relies on data from a previous step, you can conditionally clear the saved state of the Custom LWC whenever a user navigates to a previous step. After configuration, the Step element passes a boolean flag to the custom LWC. The code in your Custom LWC must use the boolean to determine whether to instantiate a new saved state or continue using the existing saved state.

1. In Omniscript, determine the Step that clears the Custom LWCs saved state.
2. In the Step's properties, click **Edit as JSON**.
3. Enter the property `"omniClearSaveState": []`.
4. In the `omniClearSaveState` property, enter the name of one or more custom LWC elements `"omniClearSaveState": ["CustomLWC1"]`. The Step sends data to the Custom LWC enabling the Custom LWC to determine if the Step rendered.
5. In your Custom LWC's code, retrieve the current state.

```
const state = this.omniGetSaveState()
```

6. In your Custom LWC's code, set a variable equal to

`this.omniGetSaveState(this.omniJsonDef.name + '-$vlocity.cleared')`. The function evaluates whether the previous Step containing the `omniClearSaveState` property rendered and returns a boolean.

```
let stateCleared = this.omniGetSaveState(this.omniJsonDef.name + '-$Vlocity.cleared');
```

7. Modify your code by using the boolean value to determine whether to keep the current saved state or instantiate a new saved state.

Access the Layout of the OmniscriptBaseMixin Component

Call the layout for your Custom LWC by passing the `data-omni-layout` parameter to the `getAttribute` method.

Use this method:

```
this.getAttribute('data-omni-layout')
```

Remote Calls

Enable users to make remote calls from a Custom LWC by using the `omniRemoteCall()` method.

Method: `omniRemoteCall(params, boolean)`

Arguments:

- The first argument must be an object with the properties `input`, `sClassName`, `sMethodName`, and `options`. The values of each of the properties must be in String data type.

 **Note** Methods called within a package must have the namespace prepended with `${this._ns}`. For example, `` ${this._ns}IntegrationProcedureService``.

```
const params = {
    input: JSON.stringify(this.omniJsonData),
    sClassName: 'SomeApexClass',
    sMethodName: 'someApexMethod',
    options: '{}',
};
```

- The second argument for `omniRemoteCall` is an optional boolean value that links a tracked property called `omniSpinnerEnabled`. The HTML must have some code to check the boolean's value.

Here's an HTML example:

```
<template if:true={omniSpinnerEnabled}>
    <div class="slds-spinner-container__wrapper">
```

```
<c-spinner variant="brand"
            alternative-text="Loading.."
            size="medium">
</c-spinner>
</div>
</template>
```

After `omniRemoteCall` executes, the response callback returns an object containing the two properties `result` and `error`.

- `result` : The result property contains a raw response from the server.
- `error` : The error property stores an error value from the Vlocity's GenericInvoke2 class.

Omnistudio provides properties that, when passed in the options, enable jobs to be queuable, future, chainable, and continuable.

Use Case Examples

- Remote call invoking an Apex class

```
const params = {
    input: this.omniJsonDataS,
    sClassName: 'SomeApexClass',
    sMethodName: 'someApexMethod',
    options: '{}',
};

this.omniRemoteCall(params, true).then(response => {
    window.console.log(response, 'response');
}).catch(error => {
    window.console.log(error, 'error');
});
```

- Integration Procedure call

```
const params = {
    input: this.omniJsonDataStr,
    sClassName: `${this._ns}IntegrationProcedureService`,
    sMethodName: 'test_RemoteAction', this will need to match the VI
    P -> type_subtype
    options: '{}',
};

this.omniRemoteCall(params, true).then(response => {
```

```

        window.console.log(response, 'response');
    }).catch(error => {
        window.console.log(error, 'error');
    });
}

```

- Chainable Integration Procedure call passing the property `chainable: true`. Use chainable when an Integration Procedure exceeds the Salesforce CPU Governor limit.

```

const options = {
    chainable: true,
};

const params = {
    input: this.omniJsonDataStr,
    sClassName: `${this._ns}IntegrationProcedureService`,
    sMethodName: 'test_RemoteAction', this will need to match the VI
P -> type_subtype
    options: JSON.stringify(options),
};

this.omniRemoteCall(params, true).then(response => {
    window.console.log(response, 'response');
}).catch(error => {
    window.console.log(error, 'error');
});

```

- Queueable remote call passing the property `useQueueableApexRemoting: true`. For more information on how queueable works, see [Queueable Apex](#).

```

const options = {
    input: this.omniJsonDataStr,
    vlcClass: 'SomeApexClass',
    vlcMethod: 'someApexMethod',
    useQueueableApexRemoting: true,
};

const params = {
    input: this.omniJsonDataStr,
    sClassName: `${this._ns}VFActionFunctionController.VFActionFunctionControllerOpen`,
    sMethodName: 'runActionFunction',
    options: JSON.stringify(options),
};

this.omniRemoteCall(params, true).then(response => {

```

```
        window.console.log(response, 'response');
    }).catch(error => {
        window.console.log(error, 'error');
});
```

- Future remote call passing the property `useFuture: true`. For more information on how Future Methods work, see [Future Methods](#).

```
const options = {
    useFuture: true,
};

const params = {
    input: this.omniJsonDataStr,
    sClassName: 'SomeApexClass',
    sMethodName: 'someApexMethod',
    options: JSON.stringify(options),
};

this.omniRemoteCall(params, true).then(response => {
    window.console.log(response, 'response');
}).catch(error => {
    window.console.log(error, 'error');
});
```

- Continuation remote call passing the parameter `useContinuation: true`. For more information on how continuation works, see [Continuation Class](#).

```
const options = {
    input: this.omniJsonDataStr,
    vlcClass: 'SomeApexClass',
    vlcMethod: 'someApexMethod',
    useContinuation: true,
};

const params = {
    input: this.omniJsonDataStr,
    sClassName: 'SomeApexClass',
    sMethodName: 'someApexMethod',
    options: JSON.stringify(options),
};

this.omniRemoteCall(params, true).then(response => {
    window.console.log(response, 'response');
}).catch(error => {
```

```
    window.console.log(error, 'error');
});
```

Data JSON Update

Update the custom Lightning web component element, map responses to the Omniscript's data JSON, and create and map data inside the custom Lightning Web component element.

1. Update of the Custom Lightning Web Component Element's Data

Apply the response of a Custom LWC callout to the Custom LWC element by using the `omniUpdateDataJson` method.

2. Mapping of Responses to the Omniscript's Data JSON

Apply responses from custom actions using the `omniApplyCallResp()` method.

3. Create and Map Data Inside the Custom LWC Element

Construct additional data JSON nodes in the Custom LWC element and pass values down from the element into the new JSON nodes using the `omniaggregate` event..

Update of the Custom Lightning Web Component Element's Data

Apply the response of a Custom LWC callout to the Custom LWC element by using the `omniUpdateDataJson` method.

Method: `omniUpdateDataJson(input, aggregateOverride = false)`

Arguments:

- `input` : The first argument, input, accepts data in an object or primitive data type. Undefined isn't accepted. The input data either replaces the current value or merges with the existing data depending on the setting of the second argument, `aggregateOverride` .
- `aggregateOverride` : The second argument, aggregateOverride, controls how the data saves to the data JSON. It's an optional argument that is set to false by default. When set to false, the input data merges with the existing data. If the data is not an object it doesn't merge into the data JSON. When set to true, the input data overrides any existing data in the Custom LWC element.

Example

- Pass the `input` argument with the default `aggregateOverride` behavior:

```
let myData = "myvalue" // any kind of data that is a string, object, number, etc
```

```
this.omniUpdateDataJson(this.myData);
```

- Pass the `input` argument with the `aggregateOverride` argument set to `false`:

```
// 1. first call to omniUpdateDataJson
// input for omniUpdateDataJson when aggregateOverride = false
{
    "prop1" : "data1"
}

// 2. output of data JSON after the first call
//
{
    "Step1" : {
        "CustomLWC1" : {
            "prop1": "data1"
        }
    }
}

// 3. second call to omniUpdateDataJson
// input for omniUpdateDataJson when aggregateOverride = false
{
    "prop2" : "data2"
}

// 4. output of data JSON after the second call
// notice that the value of CustomLWC1 contains both prop1 and prop2
// because the data is merged instead of replaced when aggregateOverride
is set to false
{
    "Step1" : {
        "CustomLWC1" : {
            "prop1": "data1",
            "prop2": "data2"
        }
    }
}
```

Mapping of Responses to the Omniscript's Data JSON

Apply responses from custom actions using the `omniApplyCallResp()` method.

The method accepts an object that it passes into the Omniscript's data JSON. If the root data node name

in the response matches an element name in the Omniscript, that data prefills the element, and any nested elements, when it renders in the Omniscript. If the root node name does not match an element in the data JSON, the node inserts into the data JSON immediately. Applying the response works similar to the Send/Response property. For information, see [Manipulate JSON with the Send/Response Transformations Properties](#).

-  **Note** The method can't support proxy objects. To include proxy inputs, you must first clone the input object by using `JSON.parse` and `JSON.stringify` and then pass it to the method, as shown in this example.

 **Example**

```
input = JSON.parse(JSON.stringify(input));
this.omniApplyCallResp(input);
```

Method: `omniApplyCallResp(response, usePubsub = false)`

Arguments:

- `response` : Object that merges into the data JSON.
- `usePubSub` : Controls how the response passes up to the Omniscript header to merge into the data JSON. This boolean is set to false by default. When set to `false`, `response` is sent to the Omniscript header via javascript events. When set to `true`, `response` is sent to the Omniscript header by the pubsub module.

The `usePubSub` argument enables users to call `omniApplyCallResp` asynchronously. When running asynchronously, the UI is not blocked waiting for the response.

Optionally perform a remote call by using the `omniNextStep()` method and calling the

`omniApplyCallResp()` method asynchronously.

 **How the `omniApplyCallResp()` method updates Data JSON**

1. View the current data JSON before running `omniApplyCallResp()`.

```
{
    "Step1" : {
        "Currency1" : 12345,
        "Text1" : "data1"
    },
    "Step2" : {
        "Text3": "data3"
    }
}
```

2. Call `omniApplyCallResp` and pass data as an argument.

```
let myData = {
    "Step1" : {
        "CustomLWC1" : "newprop"
    },
    "Anotherprop" : {
        "prop1" : "anothervalue"
    }
}

this.omniApplyCallResp(myData);
```

3. View the updated data JSON.

```
{
    "Step1" : {
        "Currency1" : 12345,
        "Text1" : "data1",
        "CustomLWC1" : "newprop"
    },
    "Step2" : {
        "Text3": "data3"
    },
    "Anotherprop" : {
        "prop1" : "anothervalue"
    }
}
```

Create and Map Data Inside the Custom LWC Element

Construct additional data JSON nodes in the Custom LWC element and pass values down from the element into the new JSON nodes using the `omniaggregate` event..

For more information on events, see [Create and Dispatch Events](#).

Event: `omniaggregate`

1. Create a custom aggregate function and set the `omniaggregate` event to a variable.

Example:

```
customAggregate() {
    const eventName = 'omniaggregate'
}
```

2. Create a variable to store the LWC element's input data.

Example:

```
customAggregate() {
    const eventName = 'omniaggregate'
    const data = {
        anyKeyName : 'myElementData'
    };
}
```

3. Construct an object by mapping the data to the key **data**, and an element name to the key **elementId**.
- Example:

```
customAggregate() {
    const eventName = 'omniaggregate'
    const data = {
        anyKeyName : 'myElementData'
    };
    const detail = {
        data: data,
        elementId: 'elementName2'
    };
}
```

4. (Optional) In the detail object, include the key-value pair `aggregateOverride: true` to override all existing data in the Custom LWC element.

Example:

```
customAggregate() {
    const eventName = 'omniaggregate'
    const data = {
        anyKeyName : 'myElementData'
    };
    const detail = {
        data: data,
        elementId: 'elementName2',
        aggregateOverride: true
    };
}
```

5. Create a new event object that passes the omniaggregate event and an object containing these four properties:

- **bubbles**: A boolean that determines if the event bubbles up to the DOM.
- **cancelable**: A boolean that determines if the event is cancelable.
- **composed**: A boolean that determines if the event can pass through the shadow boundary.
- **detail**: Data passed in the event.

Example:

```

omniAggregate() {
    const eventName = 'omniaggregate'
    const data = {
        anyKeyName : 'myElementData'
    };
    const detail = {
        data: data,
        elementId: 'elementName2'
    };
    const myEvent = new CustomEvent(eventName, {
        bubbles: true,
        cancelable: true,
        composed: true,
        detail: detail,
    });
}

```

6. Call `this.dispatchEvent()` and pass in the event object as a parameter.

Code Example	Data JSON Result Example
<pre> // code to call omniAggregate() { const eventName = 'omniaggregate' const data = { anyKeyName : 'myElementData' }; const detail = { data: data, elementId: 'elementName2' }; const myEvent = new CustomEvent(eventName, { bubbles: true, cancelable: true, composed: true, detail: detail, }); this.dispatchEvent(myEvent); } </pre>	<pre> "CustomLWC12": { "elementName1": { "prop": "prop1" }, "elementName2": { "anyKeyName": "myElementData" } } </pre>

Create a Standalone Custom Lightning Web Component

Enable custom Lightning web components to act independently from an Omniscript by adding it to the Omniscript as a standalone component. Standalone custom Lightning web components cannot extend any Omniscript element component or the OmniscriptBaseMixin component. The standalone component supports custom functionalities, but cannot interact with an Omniscript or the Omniscript's data JSON.

1. Review and complete the requirements for creating a standalone custom Lightning Web component. See [Requirements for Custom Lightning Web Components for Omniscripts](#).
2. Construct additional data JSON nodes in the Custom LWC element and pass values down from the element into the new JSON nodes using the `omniaggregate` event. See [Create and Map Data Inside the Custom LWC Element](#).
3. Add the standalone custom Lightning web component to an Omniscript. See [Custom LWC Element](#).

See Also

- [Communicate from Omniscript to a Lightning Web Component](#)
- [Custom LWC Element](#)
- [Extend the OmniscriptBaseMixin Component](#)

Create and Map Data Inside the Custom LWC Element

Construct additional data JSON nodes in the Custom LWC element and pass values down from the element into the new JSON nodes using the `omniaggregate` event.

For more information on events, see [Create and Dispatch Events](#).

Event: `omniaggregate`

1. Create a custom aggregate function and set the `omniaggregate` event to a variable.

Example:

```
customAggregate() {  
    const eventName = 'omniaggregate'  
}
```

2. Create a variable to store the LWC element's input data.

Example:

```
customAggregate() {  
    const eventName = 'omniaggregate'  
    const data = {  
        anyKeyName : 'myElementData'  
    };  
}
```

```
}
```

3. Construct an object by mapping the data to the key **data**, and an element name to the key **elementId**.

```
customAggregate() {
    const eventName = 'omniaggregate'
    const data = {
        anyKeyName : 'myElementData'
    };
    const detail = {
        data: data,
        elementId: 'elementName2'
    };
}
```

4. (Optional) In the detail object, include the key-value pair `aggregateOverride: true` to override all existing data in the Custom LWC element.

```
customAggregate() {
    const eventName = 'omniaggregate'
    const data = {
        anyKeyName : 'myElementData'
    };
    const detail = {
        data: data,
        elementId: 'elementName2',
        aggregateOverride: true
    };
}
```

5. Create a new event object that passes the omniaggregate event and an object containing these four properties:

- **bubbles**: A boolean that determines if the event bubbles up to the DOM.
- **cancelable**: A boolean that determines if the event is cancelable.
- **composed**: A boolean that determines if the event can pass through the shadow boundary.
- **detail**: Data passed in the event.

Example:

```
omniAggregate() {
    const eventName = 'omniaggregate'
    const data = {
        anyKeyName : 'myElementData'
    };
    const detail = {
        data: data,
```

```

        elementId: 'elementName2'
    };
const myEvent = new CustomEvent(eventName, {
    bubbles: true,
    cancelable: true,
    composed: true,
    detail: detail,
});
}

```

6. Call `this.dispatchEvent()` and pass in the event object as a parameter.

Code Example	Data JSON Result Example
<pre> // code to call omniAggregate() { const eventName = 'omniaggregate' const data = { anyKeyName : 'myElementData' }; const detail = { data: data, elementId: 'elementName2' }; const myEvent = new CustomEvent(eventName, { bubbles: true, cancelable: true, composed: true, detail: detail, }) this.dispatchEvent(myEvent); } </pre>	<pre> "CustomLWC12": { "elementName1": { "prop": "prop1" }, "elementName2": { "anyKeyName": "myElementData" } } </pre>

What's next: [Add a Custom Lightning Web Component to a Custom LWC Element.](#)

Communicate from Omniscript to a Lightning Web Component

Send data from Omniscript actions and steps to other Lightning web components using the Publish and Subscribe property.

Before you begin, review the documentation on using PubSub. See [Communicate Between Components](#).

-  **Note** The `pubsub` module doesn't operate when the Standard Runtime is enabled. Work around this limitation by using `window post message` (wpm) or `session storage message` (ssm). For more information, see [Message with Window Post Messages and Session Storage Messages](#).

The Pub/Sub property enables Action and Step elements to send data in key-value pairs to other Lightning web components. Lightning web components must register the Omniscript component's event name and add code to handle the data sent from the event.

1. In an Omniscript action or step, in the Properties panel, under Messaging Framework, select **Pub/Sub**.
2. In the Key and Value fields, configure which data to pass to another Lightning web component.
 - a. For Key, enter a name to store the value.
 - b. For Value, enter data to pass to the Lightning web component. The field accepts merge syntax.

Action Example	Step Example
Pass data from the action's response in the Value field using merge syntax. For example, to pass the response node " <code>accountId</code> ", enter <code>%accountId%</code> for the value.	Pass data from an element within the step using merge syntax. For example, to pass a text element named <code>Text1</code> , enter <code>%Text1%</code> for the value.

3. Activate the Omniscript.
4. In an Lightning web component that receives data from the Action or Step, import the `pubsub` module.

```
import pubsub from 'namespace/pubsub';
```

5. In the Lightning web component, register the `pubsub` event using this code without replacing anything. The even must register after the component renders.

Action pubsub Event	Step pubsub Event
<pre>pubsub.register('omniscript_action', { data: this.handleOmniAction.bind(this), }); }</pre>	<pre>pubsub.register('omniscript_step', { data: this.handleOmniStepLoadData.bind(this), }); }</pre>

6. In the Lightning web component, create a `pubsub` event handler.

Action Event Handler Example

```
handleOmniAction(data) {
    // perform logic to handle the Action's response data
}
```

Step Event Handler Example

```
handleOmniStepLoadData(data) {
    // perform logic to handle the pbsub data from the Step
}
```

7. (Optional) If different actions or steps require different logic, use a switch statement to describe how to handle the event. For information on switch statements, see[switch](#).

Action Example

Access the action's Element Name using
`data.name`.

```
handleOmniAction(data) {
    switch(data.name) {
        case 'SomeNameForAction':
            this.handleSomeNameForAction(data);
            break;
        case 'SomeNameForAction2':
            this.handleSomeNameForAction2(data);
            break;
        default:
            // handle default case
    }
}
```

```
handleSomeNameForAction(data) {
    // perform some logic specific to SomeNameForAction action
    // that has element name = "SomeNameForAction"
}
```

```
handleSomeNameForAction2(data) {
    // perform some logic specific to SomeNameForAction2 action
    // that has element name = "SomeNameForAction2"
}
```

Step Example

Access the step's Element Name using
`data.name`.

```
handleOmniStepLoadData(data) {
    switch(data.name) {
        case 'FirstStep':
            this.handleOmniFirstStepLoadData(data);
            break;
        case 'SecondStep':
            this.handleOmniSecondStepLoadData(data);
            break;
        default:
            // handle default case
    }
}
```

```
handleOmniFirstStepLoadData(data) {
    // perform some logic specific when a Step element which name is
    // FirstStep is loaded
}
```

```
handleOmniSecondStepLoadData(data) {
    // perform some logic specific when a Step element which name is
    // SecondStep is loaded
}
```

Action Example	Step Example
<pre>eNameForAction2" }</pre> <p>Access the action's Element Type using <code>data.type</code>.</p> <pre>handleOmniAction(data) { switch(data.type) { case 'Integration Procedure Action': this.handleIPActionPubsubEvents(data); break; case 'Remote Action': this.handleRemoteActionPubsubEvents(data); break; default: // handle default case } } handleIPActionPubsubEvents(data) { // perform some logic specific to Integration Procedure Actions } handleRemoteActionPubsubEvents(dat a) { // perform some logic specific to Remote Actions }</pre>	<pre>}</pre> <p>n/a</p>

Make Remote Calls Within Omniscripts from Lightning Web Components

In the standard runtime, make remote calls from a customized Omniscript component or a custom Lightning web component.

Before you invoke remote calls:

- Deploy the Omniscript customization npm package in the org. See [Set Up Your Environment to Customize Omniscript Elements](#).
- Migrate the Omniscript custom Lightning web components that extend the omniscriptBaseMixin component. See [Update Omniscript Custom Lightning Web Components and Omniscript Elements Overridden with Customized Components](#).
- Extend the omniscriptBaseMixin component. See [Extend the OmniscriptBaseMixin Component](#).

Call from a Customized Omniscript Component

If you've extended an Omniscript component (for example, omniscriptText) and not the omniscriptBaseMixin component, get the UUID by using:

```
this.omniScriptHeaderDef.uuid
```

 **Note** It isn't recommended to extend the omniscriptBaseMixin component and another Omniscript component to avoid unexpected behaviors.

Call an Integration Procedure From Lightning Web Components

The `OmniscriptActionUtilsForCore` utility calls the active Integration Procedure named `example_integrationprocedure`, where the type is `example` and the subtype is `integrationprocedure`.

```
import OmniscriptActionUtilsForCore from 'c/omniscriptActionUtilsForCore';

this.omniactions = new OmniscriptActionUtilsForCore(this.omniScriptHeaderDef.uid);
this.omniactions.runIntegrationProcedure({
    ipName: "example_integrationprocedure",
    options: {},
    input: {}
}).then((result) => {
    window.console.log("IP RESULT :");
    window.console.log(result);
});
```

Call a Data Mapper Extract From Lightning Web Components

The `OmniscriptActionUtilsForCore` utility calls the Data Mapper Extract named `getsomeaccounts`.

```
this.omniactions = new OmniscriptActionUtilsForCore(this.omniScriptHeaderDef.u
```

```
uid);
this.omnactions.runDataMapperExtract({
    dmName: "getsomeaccounts",
    options: {},
    input: {}
}).then((result) => {
    window.console.log("DME RESULT :");
    window.console.log(result);
});
});
```

Call a Data Mapper Load From Lightning Web Components

The `OmniscriptActionUtilsForCore` utility calls the Data Mapper Load named `createaccount`.

```
this.omnactions = new OmniscriptActionUtilsForCore(this.omniScriptHeaderDef.u
uid);
this.omnactions.runDataMapperLoad({
    dmName: "createaccount",
    options: {},
    input: {}
}).then((result) => {
    window.console.log("DML RESULT :");
    window.console.log(result);
});
});
```

Call a Data Mapper Transform From Lightning Web Components

The `OmniscriptActionUtilsForCore` utility calls the Data Mapper Transform named `transformdata`.

```
this.omnactions = new OmniscriptActionUtilsForCore(this.omniScriptHeaderDef.u
uid);
this.omnactions.runDataMapperTransform({
    dmName: "transformdata",
    options: {},
    input: {
        "one" : {
            "dataforone" : "ONEE"
        },
        "two" : {
            "datafortwo": "TWOOO"
        }
    }
});
```

```
        }
    }).then((result) => {
    window.console.log("DMT RESULT :");
    window.console.log(result);
}) ;
```

Call Apex From Lightning Web Components

Omniscript's custom lightning web components now support the Apex imports. See [Import Apex Methods](#).

If your custom LWC calls any Omnistudio managed package Apex classes, make sure to replace them with the standard runtime equivalents. Contact Salesforce support for more information.

Add Validation to a Custom Lightning Web Component in Omniscript

Add validation to custom Lightning web components that extend an Omniscript element's component by using Vlocity's built-in validation methods.

The `omniscriptValidation` component provides the Omniscript element's component with the functions to enable validation. For information on using validation in custom components that do not extend an Omniscript element's component, see [Extend the OmniscriptBaseMixin Component](#).

Omniscript Validation ReadMe

View validation examples for custom Lightning web components in the Omniscript Validation ReadMe. See [Omniscript ReadMe Reference](#).

Omniscript ReadMe Reference

This page lists the Lightning web components ReadMes available for Omniscripts. Extend Lightning web components to add custom behavior and styling.

All Omnistudio component HTML, CSS, and read-me files are available when you

[Set Up Your Environment to Customize Omniscript Elements](#). Check the folder for each component to find these files.

Base Component ReadMes

LWC Omniscript elements extend the base components in this table.

LWC	Description
Omniscript Atomic Element	The base component for Omniscript Input elements.
Omniscript Base Action	The base component for Omniscript Action elements.
Omniscript Base Element	The base component for LWC Omniscript.
Omniscript Group Element	The base component for Omniscript Group elements.

Mixin ReadMes

LWC	Description
Omniscript Base Mixin	Enables custom LWCs that do not override an Omniscript element's LWC to interact with Omniscript through the Custom LWC element. See Extend the OmniscriptBaseMixin Component .
Omniscript Options Mixin	Provides options logic for any LWC in Omniscript.
Omniscript Validation	Provides validation options to any LWC in Omniscript.

Preview and Test an Omniscript

View the Omniscript's appearance and functionality by previewing the Omniscript in the Omniscript Designer.

1. In your Omniscript, click **Preview**.
2. For additional preview options, select these properties:

Context ID	If your Omniscript doesn't have a Set Values Action that defines a ContextId, to preview the
-------------------	--

	<p>form with test data, enter a record ID.</p> <p>Using Set Values Action to set the context ID only works in preview.</p> <p>Confirm that a Context Id is defined in your data source. For example, if you have an Omnistudio Data Mapper Extract Action that gets account records, you must have an Input Parameter whose Data Source is <i>ContextId</i>, and whose Filter Value is the Data Mapper Extract's output name, such as <i>AccountId</i>.</p>
Preview Device	<p>To preview how your Omniscript appears on different devices, select <i>Mobile</i>, <i>Tablet</i>, or <i>Desktop</i>.</p> <p>Fields are arranged in a single column on mobile, even if you have put them side by side for the desktop version.</p>
Theme	Preview your Omniscript with a Lightning or Newport theme.
Reset data fields	To reset data fields in your Omniscript after making changes, click the reload icon.

3. In the Debug panel, view the JSON data in the Data JSON.
 4. If needed, click the clipboard icon to copy the full JSON.
 5. To open the debug console, click **Action Debugger**.
 6. Search for an action name in the Search field.
 7. To copy a specific node, click the clipboard icon next to the node. For example, copy Remote Options in an Omnistudio Data Mapper Extract Action.
 8. If needed, clear the debug console. To repopulate the debug console with new action request and response data, click .
 9. To hide the Data JSON and Action Debugger, click .
- To display the Data JSON and Action again, click the debug panel button that appears when the panel closes.

If you see the error "Omniscript is either inactive or has not been deployed," the Omniscript is too large. The Lightning web component for an Omniscript can't be larger than 4 MB because of a [payload data limit](#) with Aura components. In the designer on a managed package, download the Omniscript LWC to check its size.

When you preview an Omniscript with a child Omniscript with a custom CSS, the content for the child Omniscript doesn't show the custom styling.

Activate and Launch Omniscripts

Make Omniscripts available to Experience Sites, Lightning Pages, custom LWCs, and Lightning tabs by activating and launching the Omniscript. If an error occurs during deployment, the Omniscript is deactivated.

Before you begin, make sure that all components embedded in the Omniscript are also active.

1. From the Omniscript designer, activate the Omniscript.
2. After activating the Omniscript, click **How to launch?** to view how you can launch the Omniscripts. See [How to Launch Omniscripts](#).
3. After activating your Omniscript, launch your Omniscript from a Lightning page or Experience page in these ways:
 - Launch an Omniscript using the standard Omniscript component.
 - Launch an Omniscript using the Omniscript's generated LWC.
4. If an error occurs on the initial load of an Omniscript, in the Omniscript, click **Deactivate**, then click **Activate**.
5. If there's an issue with deploying the Omniscript in a Lightning or Experience page, redeploy the Omniscript by clicking **Deploy Standard Runtime Compatible LWC**.



Note This feature applies only to the managed package designer on standard runtime.

6. To reference your Omniscript on an Aura Experience Cloud site, or on a Lightning page, use the Omniscript wrapper component available from the Salesforce Lightning Component Library. See [Component Library](#) for more information. If you're referencing an Omniscript in a Lightning web component (LWC), you must use a wrapper component that takes in the name of the Omniscript you want to load at runtime, as shown. The type, subtype, and language attributes are mandatory. The Lightning theme is added by default. However, if you'd like to use the Newport theme instead, add the theme attribute as well.



Example

```
<namespace-omnistudio-standard-runtime-wrapper type={type}
                                                subtype={subtype}
                                                language={language}
                                                theme={theme}
                                                inline={inline}
                                                inlinevariant={inlinevariant}
                                                inlinelabel={inlinelabel}
                                                direction={direction}
                                                record-id={recordId}
```

```
prefill={prefill} >  
</namespace-omnistudio-standard-runtime-wrapper>
```

How to Launch Omniscripts

Make an active Omniscript available for launch in one of three ways: as a standalone Lightning Web Component in a Lightning Web Page or as a URL.

Add Your Omniscript to a Lightning Page

In Omnistudio standard runtime, add the standard Omniscript component to a Lightning page after activation.

Add Your Omniscript to an Experience Cloud Page

In the Omnistudio standard runtime, add the standard Omniscript component to an Experience Builder page after activation.

How to Launch Omniscripts

Make an active Omniscript available for launch in one of three ways: as a standalone Lightning Web Component in a Lightning Web Page or as a URL.

- ⓘ **Note** On the **How to Launch** page in the Omniscript designer, the Newport URL and the code samples beginning with `this[NavigationMixin.Navigate]` don't apply to Omnistudio Omniscripts.
- ⓘ **Note** It is recommended that you use a standard Omnistudio component on a Lightning or Experience Cloud page. If you migrate from the Omnistudio managed package runtime to the standard runtime, you can create new versions of existing Omniscripts for standard runtime. Then, you can add those Omniscripts via the Process Automation section on the Lightning App Builder or Aura-based Experience Builder page. In cases where you're unable to create standard versions of your Omniscripts, you may use a standalone or embedded Omniscript as described here.

Standalone

Open a Lightning Web Page in the Lightning App Builder, find the LWC for the Omniscript in the Custom Components section, and drag it onto the page.

- ⓘ **Note** This feature applies only to the designer on a managed package in standard runtime.

Embedded

If you're referencing an Omniscript on a Lightning Web Runtime (LWR) Experience Cloud site, you must use a wrapper component that takes in the name of the Omniscript you want to load at runtime, as shown. The type, subtype, and language attributes are mandatory. The Lightning theme is added by

default. However, if you'd like to use the Newport theme instead, add the theme attribute as well.



Example

```
<namespace-omnistudio-standard-runtime-wrapper type={type}
                                                subtype={subtype}
                                                language={language}
                                                theme={theme}
                                                inline={inline}
                                                inlinevariant={inlinevariant}
                                                inlinelabel={inlinelabel}
                                                direction={direction}
                                                record-id={recordId}
                                                prefill={prefill} >
</namespace-omnistudio-standard-runtime-wrapper>
```

For embedding an Omniscript in an LWC on an Aura page, use this format.



Example

```
<namespace:omnistudioStandardRuntimeWrapper
    type="type"
    subtype="subtype"
    language="language">
</omnistudio:omnistudioStandardRuntimeWrapper>
```

URL

Newport URL Example

```
https://myorg.lightning.force.com/lightning/page/omnistudio/omniscript?omniscript_type=Type&omniscript_subType=SubType&omniscript_language=English&omniscript_theme=newport&omniscript_tabIcon=custom:custom_tab_icon&omniscript_tabLabel=custom_tab_label
```

Lightning URL Example

```
https://myorg.lightning.force.com/lightning/page/omnistudio/omniscript?omniscript_type=Type&omniscript_subType=SubType&omniscript_language=English&omniscript_theme=lightning&omniscript_tabIcon=custom:custom_tab_icon&omniscript_tabLabel=custom_tab_label
```

Add Your Omniscript to a Lightning Page

In Omnistudio standard runtime, add the standard Omniscript component to a Lightning page after activation.

1. In your org, go to **Setup | Lightning App Builder** and click **Edit** for an existing Lightning app to open the Lightning App Builder. Or, on a Lightning record page, from the Setup menu, click **Edit Page**.
2. Drag the Omniscript component from the **Standard** section of the Components pane onto the canvas.

When you add the Omniscript component, it pulls in the first active Omniscript. The component shows up on the canvas as a placeholder, and the type and subtype are pre-filled in the dropdown from the first active Omniscript.

3. If the retrieved Omniscript isn't the script you must use, select another type and subtype option from the dropdown menu.
4. Select a **Theme**.
For example, the Salesforce Lightning Design System (SLDS) or Newport. You can switch themes at any time.
5. For **Display**, choose whether you want the Omniscript to display on the page, or use a button to open the Omniscript.

If you select a button to open Omniscript, you can apply one of the SLDS styles for the button variant.

6. Select the **Language** to use for Omniscript.
Depending on the language you select, you can select either Left to Right or Right to Left for your **Language Direction**.
7. To modify the component's visibility by adding filters, set **Component Visibility**.
See [Dynamic Lightning Pages](#).
8. To configure activation options, click **Activation**.
See [Activate Your Lightning App Page](#).

Add Your Omniscript to an Experience Cloud Page

In the Omnistudio standard runtime, add the standard Omniscript component to an Experience Builder page after activation.

You can add an Omniscript to an Aura-based Experience Cloud site, or a Lightning Web Runtime (LWR) Experience Cloud site. To add an Omniscript to an LWR site, you must first contact Salesforce Customer Support to enable this capability. Once done, you can perform these tasks.

1. To access the Experience Builder, see [Navigate Experience Builder](#).
2. To open the Components pane, click the lightning bolt icon.
3. Drag the Omniscript component from the **Process Automation** section onto the canvas, if you're using an Aura-based Experience Cloud site. On the LWR-based Experience Builder, find the Omniscript component in the Customer Interactions section.

When you add the Omniscript component, it pulls in the first active Omniscript. The component

shows up on the canvas as a placeholder, and the type and subtype are pre-filled in the dropdown from the first active Omniscript.

4. If the retrieved Omniscript isn't the script you must use, select another type and subtype option from the dropdown menu.

5. Configure your script by using the choices from the dropdown menu.

6. Select a **Theme**.

For example, the Salesforce Lightning Design System (SLDS) or Newport. You can switch themes at any time.

7. For **Display**, choose whether you want the Omniscript to display on the page, or use a button to open the Omniscript.

If you select a button to open Omniscript, you can apply one of the SLDS styles for the button variant.

8. Select the **Language** to use for Omniscript.

Depending on the language you select, you can select either ltr (left to right), or right to left for your **Language Direction**.

9. To preview, click **Preview**.

See [Preview Your Experience Builder Site](#).

Considerations for Using Omniscripts with Lightning Web Runtime Sites

Both existing and newly created Omniscripts can be used on Lightning Web Runtime (LWR) Experience Cloud sites. Learn about some key considerations before you choose an Omniscript to embed in your LWR sites.

You can use the Omniscripts through the LWR site builder under the Customer Interactions category. To make sure that your Omniscripts display correctly and function well, note these pointers and choose only compliant Omniscripts for your LWR sites.

- If you make changes to your Omniscript, make sure that you republish the sites that use it. Child Omniscripts must also be republished after a change is made.



Note Before republishing your sites via the site builder, consider other elements from other products that you've embedded in your site. Only republish if everything production-ready.

- Lightning pages, Experience Cloud Aura sites and LWR sites all implement their own styling. To reuse an Omniscript with a modified UI appearance, enter the token overrides in the Omniscript corresponding to the consuming page or site.
- To use the Open Omniscript option on an LWR site, make sure that you select the **Load Omniscript from URL** checkbox in the Properties section of the page. This checkbox is added to the Omniscript component and loads the Omniscript present in the URL configured via the Action Type. You must also create an `lwcos` page for this action to work. For more information, see [Launch an Omniscript from an Omniscript Action on a Flexcard in Experience Cloud](#).
- Custom LWCs are supported. However, due to a technical limitation, nested custom LWCs require a workaround to work as expected. Because LWR sites work on cached data, they look for Flexcards and Omniscripts nested within your custom LWCs on the site. If present, they load the appropriate component at runtime. To do this, you can pick one of the two methods outlined here.
 - Create a page on your site and add all nested components in your main LWC component to this page. Hide the page so it doesn't show up for users as a navigation option.

On the page that you're adding your custom LWC, add the dependent Flexcards and Omniscripts. Then, for each component, control the visibility via the Visibility tab. For instance, you can switch the Show Component on Desktop toggle on and add a condition that says "User ID" equals "Invalid_User". In most orgs, this condition is never true, so the dependent components stay hidden for the user but are available to the LWR site to read from.

- The Style and Visibility tabs on LWR sites don't work with Omniscript elements. You can continue to control the style and visibility of the Omniscript through the options available within the Omniscript designer, but they aren't compatible with the settings native to the LWR site builder.
- When you open an LWR site from the Digital Experiences page, the site considers you an unauthenticated user and shows only Flexcards that have data available to guest users. To make sure that you see the full data, implement a login page on your site. Then, go to the **Workspaces** link next to the site name. Click **Administration** and find the **Login & Registration** tab in the left navigation pane. Make sure that the Login Page Type dropdown has a value and that the **Allow employees to log in directly to an Experience Cloud site** checkbox is selected.

The screenshot shows the 'Login & Registration' section of the Experience Cloud administration. It includes a 'Login Page Setup' heading, a note about choosing a login page type, and two configuration options: 'Login Page Type' set to 'Default Page' and a checked 'Allow employees to log in directly to an Experience Cloud site' checkbox.

Omniscript Element Reference

The available elements are Omniscripts are organized as groups, Data Mapper actions, standard actions, display elements, functions, input, and Omniscripts (to embed reusable Omniscripts). Each element uses an extendable Lightning web component (LWC).

When you drag an element to the canvas in an Omniscript, it shows the field label in the designer. In the designer for a managed package, it shows the element name.

Common Omniscript Element Properties

Each Omniscript element has settings, some of which are unique to that element. Some settings are common among the elements, though some of these setting aren't available in all elements. To edit these settings, use the Properties pane on the Designer.

Activate Elements in the Omniscript Designer

Activate an inactive element by selecting the element in the Navigation Panel of the Omniscript Designer.

Omniscript Group Elements

From the group category, you can organize Omniscript elements. Use step elements to divide the Omniscript into sections. Use edit blocks to group input elements for users to add, edit, or delete

records. Create a questionnaire with the radio group element. Use the type ahead block to display a list of results when a user begins typing in an input field.

Omniscript Data Mapper Action Elements

Use Data Mapper actions to retrieve, write, or restructure data from one or more related Salesforce objects.

Omniscript Action Elements

Use action elements in an Omniscript for executing actions, such as get or update data from one or more Salesforce objects, call a series of actions, send emails or documents for signature, redirect to different pages. Action elements can be either rendered as a button when placed in a Step or Block or run remotely if placed between steps. In either case, you can specify a redirect page, where the user can proceed to the next step or action, or if a button, back to the source step.

Omniscript Display Elements

Use the Omniscript Display elements to display rich text and images on the screen to enhance the user interface. The display elements include Line Break and Text Block.

Omniscript Function Elements

From the Omniscript function elements, use the formula element to perform calculations that require complex calculations. For array input, use the aggregate element for complex calculations. The Messaging element displays comments, requirements, success, and warning messaging depending on whether the validate expression is true or false.

Omniscript Input Elements

When users to change, add, or delete data, add an Omniscript element for a specific type of data, such as email addresses, files, dates, passwords. You can also show information and allow users to select from a list of options, such as radio, checkboxes, multi-select, select, and disclosure elements.

Activate Elements in the Omniscript Designer

Activate an inactive element by selecting the element in the Navigation Panel of the Omniscript Designer.

When an element is deactivated, it disappears from the canvas. View inactive and active steps and actions placed outside of steps in the Slide View of the Navigation Panel. View all inactive and active elements, including those inside steps, in the Tree View of the Navigation Panel.

1. From Omniscript, in the Navigation panel, click the Tree View icon.
2. (Optional) If you are activating an action outside of a step or a step, you may also click the Slide View icon to activate the element.
3. Click the action, step, or element inside of a step that you want to activate.
4. In the Properties panel, click **Activate**.

See Also

[Create an Omniscript](#)

Omniscript Group Elements

From the group category, you can organize Omniscript elements. Use step elements to divide the Omniscript into sections. Use edit blocks to group input elements for users to add, edit, or delete records. Create a questionnaire with the radio group element. Use the type ahead block to display a list of results when a user begins typing in an input field.

[Run Multiple Actions in an Action Block](#)

Run multiple Omniscript Actions asynchronously by grouping them in an Action Block. Actions within the Action Block run in parallel and inherit the Action Block's settings.

[Combine Elements Logically in a Block](#)

Combine logical groups of elements in an Omniscript Step using a Block element. Blocks help to group information to create nested JSON data. Blocks are contained within steps and can be repeated to capture an array of data. For example, street, city, state, and postal code elements can be combined into an address block.

[Manage Records with an Edit Block](#)

Create, edit, and delete multiple Salesforce or external records on one page using Edit Block. For example, if you're collecting an Account's Contact information, you can add an Edit Block to your Omniscript to enable users to add a record for each Contact.

[When to Use Omniscript Edit Blocks and Blocks](#)

To decide when to use edit blocks and blocks, check if you need to limit repeating the block, directly modify sObjects, show fields when the block is collapsed, display the block in different layouts, or use the custom LWC element.

[Create an Omniscript Questionnaire with a Radio Group](#)

With an Omniscript's radio group, create and display questions in a questionnaire format.

[Add Steps to an Omniscript](#)

Each Step element in an Omniscript presents a page to the user that prompts for input or displays information. For navigation, all Steps except the first have a Next button, and all except the last have a Previous button.

[Add a Type Ahead Block in an Omniscript](#)

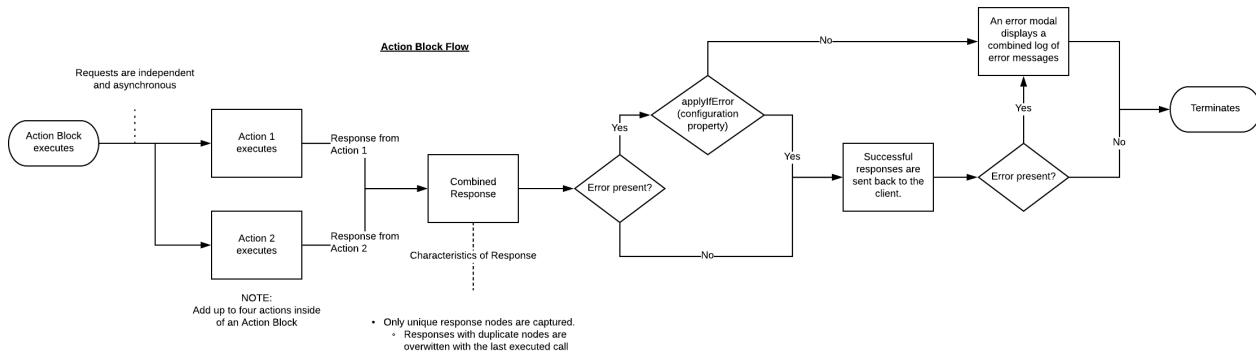
In edit blocks, suggest possible entries as a user types in a field, as an autosuggest or autocomplete.

Run Multiple Actions in an Action Block

Run multiple Omniscript Actions asynchronously by grouping them in an Action Block. Actions within the Action Block run in parallel and inherit the Action Block's settings.

View the Action Block flow chart to understand how actions in the Action Block execute.

Action Block flow chart:



- From the elements panel, drag the **Action Block** element to the canvas.
- Place the Action Block inside a step, before the first step, or between steps.
- Drag up to four action elements into the Action Block.
- Select an **Invoke Mode** to control how actions inside the Action Block run.

Default	The Action blocks the UI with a loading spinner.
Non-Blocking	The Action runs asynchronously, and the response applies to the UI. Pre and Post Omnistudio Data Mapper transforms, and large attachments aren't supported.
Fire and Forget	The Action runs asynchronously with no callback to the UI. Pre and Post Data Mapper transforms, and large attachments aren't supported. A response still appears in the debug console but doesn't apply to the Data JSON.

When Invoke Mode is set to non-blocking, elements using default values don't receive the response because the element loads before the response returns. You must configure these properties to map the response to an element:

- **Response JSON Node:** VlocityNoRootNode
 - **Response JSON Path:** The name of the Omniscript Element receiving the value.
- If needed, select **Apply if error** to enable actions that run successfully to apply their responses to the Omniscript's data JSON even when other actions in the Action Block fail.
 - When the Action Block is inside of a Step, set validation to **Step** to prevent the Action Block from running when errors exist in the Step.
 - Preview the Omniscript to test the Action Block behavior.

Combine Elements Logically in a Block

Combine logical groups of elements in an Omniscript Step using a Block element. Blocks help to group information to create nested JSON data. Blocks are contained within steps and can be repeated to capture an array of data. For example, street, city, state, and postal code elements can be combined into

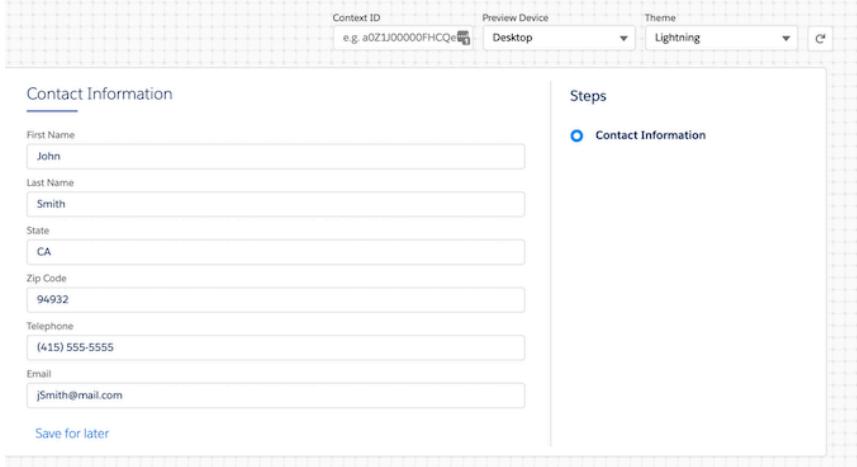
an address block.

 **Note** You can nest Blocks within other Blocks, but Blocks don't support the Custom LWC Element.

1. From the elements panel, drag a **Block** element to the canvas and name the element.
2. In **Field Label**, enter a label that displays to users.
3. To render a collapsed block in the Omniscript, select **Collapse**.
4. In the **Heading Level** field, enter the nesting level of the element as a number.
5. Enable users to add multiple block entries:
 - a. In Clone Elements, select **Copy elements**.
 - b. In the designer on a managed package, select **Enable Repeat** under **Repeat Settings**.
Repeatable blocks store data in an array format, where each block entry is indexed in an array under the Block's element name.
 - If you plan to access data from an element in a repeatable Block, see [Access to Data Within and Outside a Repeatable Block](#).
 - For information about prefilling blocks, see [Prefill Repeatable Blocks](#).
6. To clone values when a Block is repeated, select **Clone When Repeating**.
7. In the **Limit Repeat** field, enter a number to set the number of times a Block can be repeated.
8. Preview the Omniscript and test the functionality.

 **Example** Data JSON output format based on your configuration to map data JSON from blocks:

- No block JSON:



The screenshot shows a contact form with the following fields:

- First Name: John
- Last Name: Smith
- State: CA
- Zip Code: 94932
- Telephone: (415) 555-5555
- Email: jsmith@mail.com

At the bottom of the form is a "Save for later" button.

```
Object
1 > object
2   layout: "lightning"
3   nei: "oui_dailylwc"
4   lwc: "omniscriptPreview"
5   sfdcIframeHost: "web"
6   id: "a2470000000X18qAO"
7   id: "a2470000000X18qAO"
8   sfdcIframeOrigin: "https://guidailylwc-dev-ed.lightning.force.com"
9   timestamp: "2020-05-19T19:33:59.358Z"
10  userProfile: "System Administrator"
11  userRoleId: "005t00000000000000"
12  userCurrencyCode: "USD"
13  userName: "oui_dailylwc@velocity.com"
14  userId: "005t00000000000000"
15  > ContactInformation: Object
16    > ContactInformation: Object
17    > ContactInformation: Object
18    > ContactInformation: Object
19    > ContactInformation: Object
20    > ContactInformation: Object
21    > ContactInformation: Object
```

- Block JSON

```

1  ► Object
2    layout: "lightning"
3    ns: "oui_dailylwc"
4    lwc: "omniscriptPreview"
5    sfdciframeHost: "web"
6    iddpri: "p1"
7    id: "a22470000000UKtRQAO"
8    sfdciframeOrigin: "https://ouidailylwc-dev-ed.lightning.force.com"
9    timestamp: "2020-05-19T19:18:54.951Z"
10   userProfile: "System Administrator"
11   userTimeZone: "+420"
12   userCurrencyCode: "USD"
13   userName: "oui_dailylwc@velocity.com"
14   userId: "00547000000YB1QAM"
15   > ContactInformation: Object
16     > BasicInformation: Object
17       FirstName: "John"
18       LastName: "Smith"
19       State: "CA"
20       ZipCode: "94342"
21       Telephone: "4155555555"
22       Email: "jSmith@gmail.com"
  
```

- Repeatable block JSON

```

1  ► Object
2    layout: "lightning"
3    ns: "oui_dailylwc"
4    lwc: "omniscriptPreview"
5    sfdciframeHost: "web"
6    iddpri: "p1"
7    id: "a22470000000UKtRQAO"
8    sfdciframeOrigin: "https://ouidailylwc-dev-ed.lightning.force.com"
9    timestamp: "2020-05-19T19:23:782Z"
10   userProfile: "System Administrator"
11   userTimeZone: "+420"
12   userCurrencyCode: "USD"
13   userName: "oui_dailylwc@velocity.com"
14   userId: "00547000000YB1QAM"
15   > BasicInformation: Object
16     > 0: Object
17       FirstName: "John"
18       LastName: "Smith"
19       State: "CA"
20       ZipCode: "94932"
21       Telephone: "4155555555"
22       Email: "jSmith@gmail.com"
23     > 1: Object
24       FirstName: "Jane"
25       LastName: "Smith"
26       State: "NY"
27       ZipCode: "94955"
28       Telephone: "2125555555"
29       Email: "janedsmith@mail.com"
30   
```

See Also

[Access to Data Within and Outside a Repeatable Block](#)

[Manage Records with an Edit Block](#)

Manage Records with an Edit Block

Create, edit, and delete multiple Salesforce or external records on one page using Edit Block. For example, if you're collecting an Account's Contact information, you can add an Edit Block to your Omniscript to enable users to add a record for each Contact.

You can add other types of elements to an Edit Block, such as Formulas and Actions. For a complete list, see *Compatible Elements* in [Configure an Edit Block in Omniscripts](#).

-  **Note** You can't nest Edit Blocks within other Edit Blocks, and you can add only one level of Block inside an Edit Block. Also, Edit Blocks don't support the Custom LWC Element.

In addition to this, these elements are supported inside an edit block. However, you can't toggle their visibility on the initial Omniscript screen via the **Elements Inside Edit Block** edit block property.

- Image
- Block
- File
- Disclosure
- Line Break
- Text Block
- Radio Group
- Type Ahead Block

1. From the elements panel, drag an **Edit Block** to the canvas.

2. Click the Edit Block's **Enable Edit Mode** icon.

3. From the elements panel, drag the elements you need into the Edit Block.

For example, for a Phone field, drag a Telephone element into the Edit Block.

-  **Note** An Edit Block with an Edit Block Mode of Table can have up to six elements, or columns.

Including more than six elements makes the columns small and uneven, resulting in odd word wrapping.

4. Where applicable, name the elements to match the field names.

For example, for a Phone field, change the Telephone element's **Name** property from the `Telephone1` default to `Phone`.

Configure an Edit Block in Omniscripts

Edit Blocks enable a user to add and edit multiple entries in a single array. For example, you can use an Edit Block to list and edit an Account's list of Contacts. Edit Blocks are repeatable blocks by default. Prefilling repeatable blocks with data is possible using an array format.

Add, Edit, and Delete Records with Edit Block Actions

Enable users to add, edit, and delete records by using Edit Block's default remote actions. To add custom functionality, configure additional actions.

Display Sprites in an Edit Block

Display Sprites to indicate information about an Edit Block record entry using the **Default Svg Sprite** and **Default Svg Icon** properties. For example, a User icon can indicate if the entry is a Contact record. Omnistudio supports the Lightning Design System icons. To dynamically display different sprites, map values of the **SVG Controlling Element** to SLDS icons in the **SVG Controlling Element Map**.

Set a Select Mode in Edit Block

Select modes enable users to change an Edit Block entry's boolean value in the data JSON by clicking on Edit Block entries.

Work with Formulas in Edit Block

Perform equations and combine strings by using Formula Elements. Place formulas within an Edit Block to access element values within a specific Edit Block entry. Place Formula outside of the Edit

Block in the same Step to access an element value for all of the Edit Block entries.

Overwrite Edit Block Values in Omniscripts

Overwrite the number of children in an Edit Block in an Omniscript. You can also set the Edit Block's value to `null` in between steps.

Override an Edit Block LWC in Omniscripts

Apply custom code to an Edit Block by extending and overriding the Edit Block's lightning web components. Edit Blocks contain up to three LWC override fields based on the Edit Block's Edit mode.

Edit Omniscript Block Properties

This section contains information on the Edit Block properties.

See Also

[Access to Data Within and Outside a Repeatable Block](#)

Configure an Edit Block in Omniscripts

Edit Blocks enable a user to add and edit multiple entries in a single array. For example, you can use an Edit Block to list and edit an Account's list of Contacts. Edit Blocks are repeatable blocks by default. Prefilling repeatable blocks with data is possible using an array format.

Setting Display Modes and Templates

Enable Omniscripts to display different Edit Block user interfaces by using Modes.

 **Note** You can't nest Edit Blocks within other Edit Blocks, and you can add only one level of Block inside an Edit Block. Also, Edit Blocks don't support the Custom LWC Element.

Edit Block Modes

Select the mode of the Edit Block to change its display and functionality.

Inline

Lightning example:



Newport example:

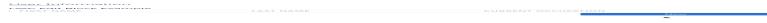


Table

The maximum number of elements that you can show as columns is 6.

Currently, edit blocks configured in table mode in Omniscripts aren't compatible with screen readers. If you use a screen reader, we recommend that you use Flexcards instead of Omniscripts. In Flexcards, you can use data tables or Lightning datatables that are compatible with screen readers. See [Show Data in a Table on a Flexcard](#) and [Lightning Web Component – Datatable](#).

Lightning example:



Newport example:



Finance Statement

The total combined width of the elements must be less than or equal to 12. Salesforce recommends displaying fewer than twelve elements. Hidden elements don't affect the total display width

Lightning example:



Newport example:



Cards

The maximum number of elements that you can display in Cards mode varies by element type. For example, you can display up to three Text Elements. Visually inspect the Edit Block to ensure that elements are rendered correctly.

To add more than the maximum number of elements, set the `height` attribute of the Edit Block to `auto` in Omniscripts, for example:

```
.omni-edit-block-card.omni-edit-block-short-cards[omnistudio-  
omniscriptEditBlock_omniscriptEditBlockCards] { height: auto !important }
```

Lightning example:



Newport example:



LongCards

The width of LongCards must be set to 12.

The maximum number of elements that you can display in LongCards mode varies by element type. For example, you can display up to four Text Elements. Visually inspect the Edit Block to ensure that elements are rendered correctly.

To add more than the maximum number of elements, set the `height` attribute of the Edit Block to `auto` in Omniscripts, for example:

```
div.slds-grid.slds-grid_vertical.slds-grid_align-center.slds-p-left_small.slds-max-small-size_5-of-12.slds-small-size_8-of-12 { height: auto !important; }
```

Lightning example:



Newport example:



Compatible Elements

The following elements are compatible with Edit Block:

- Aggregate
- Checkbox
- Currency
- Date
- Date/Time
- Disclosure
- Email
- File
- Formula
- Headline
- HTTP Action
- Image
- Integration Procedure Action
- Line Break
- Lookup
- Matrix Action
- Multi-select
- Number
- Password

- Radio
- Radio Group
- Range
- Remote Action
- Select
- Signature
- Telephone
- Text
- Text Area
- Text Block
- Type Ahead Block
- Time
- URL

What's Next

Add actions to an Edit Block or use sObject mapping to create, edit, and delete records. See [Add, Edit, and Delete Records with Edit Block Actions](#).

See Also

- [Apply Custom Styling to Omniscripts with Static Resources](#)
- [Access to Data Within and Outside a Repeatable Block](#)

Add, Edit, and Delete Records with Edit Block Actions

Enable users to add, edit, and delete records by using Edit Block's default remote actions. To add custom functionality, configure additional actions.

Edit Block supports these actions:

- HTTP Action
- Integration Procedure Action
- Remote Action



Note Word wrap isn't supported in an Edit Block.

[Create, Edit, and Delete Records with Default Actions and sObject Mapping](#)

Add sObject Mapping functionality by adding Remote Actions to the Edit Block for each option you want to make available to the User. These Remote actions will access Apex classes built for the Edit Block and the sObject Mapping tool. The Default Actions use the sObject mapping property to determine how the entries map to an sObject.

[Add Custom Actions to an Omniscript Edit Block](#)

Add custom actions to include additional functionality in your Edit Block.

[Add Global Actions to an Omniscript Edit Block](#)

Run actions on the entire JSON for the Edit Block by adding global actions. Global actions display

outside of the Edit Block entries.

Create, Edit, and Delete Records with Default Actions and sObject Mapping

Add sObject Mapping functionality by adding Remote Actions to the Edit Block for each option you want to make available to the User. These Remote actions will access Apex classes built for the Edit Block and the sObject Mapping tool. The Default Actions use the sObject mapping property to determine how the entries map to an sObject.

1. To allow a user to create entries, edit entries, or delete entries:
 - a. Add a **Remote Action** to the Edit Block.
 - a. Configure the Edit Block using the Element naming convention, Class, and Method for each of these table entries:

Element Name	Class	Method
(Edit Block Element Name)-New	DefaultOmniScriptEditBlock	new
(Edit Block Element Name)-Edit	DefaultOmniScriptEditBlock	edit
(Edit Block Element Name)-Delete	DefaultOmniScriptEditBlock	delete

2. Map the Edit Block entries to Salesforce objects by using the SOBJECT MAPPING property. Fields hidden from the user, such as an Id, remain available for mapping. To add mapping entries to the Edit Block:
 - a. Click **SObject Mapping** to expand the section.
 - b. Click **Select Object** and select a Salesforce object.
 - c. Add an SObject Field Mapping.
 - d. In the Edit Block Element field, select the name of an element that is in the Edit Block.

 **Note** By default, new entries are added, and existing entries are updated. To create a record instead of updating an existing one, select **Duplicate Key**.

- e. In the sObject field, select an sObject field. The element will map to this field.

 **Note** You must add a Text element to store the ID of the record in the Edit Block. The element containing the `record Id` must map to the sObject's ID field in the SObject Mapping section. To hide the element from displaying in the UI, see [Using the Hide Property on Omniscript Components](#).

Add Custom Actions to an Omniscript Edit Block

Add custom actions to include additional functionality in your Edit Block.

To add a custom action:

1. Add an action element into the Edit Block.
2. Name the element. Actions in the Edit Block will appear in the dropdown beneath the default actions by default.
3. To have an action replace the default New, Edit, or Delete actions, name the Action element, and append the action type.

Replace Action	Example Element Name
New	AddUserNew
Edit	ChangeAddressEdit
Delete	RemoveAccountDelete

4. Add a **Label** to the Action.
5. Configure the Action.

Add Global Actions to an Omniscript Edit Block

Run actions on the entire JSON for the Edit Block by adding global actions. Global actions display outside of the Edit Block entries.

1. Add an action element into the Edit Block.
2. Name the element and append *Global* to the element name.
3. Add a label to the action.
4. Configure the action.

Display Sprites in an Edit Block

Display Sprites to indicate information about an Edit Block record entry using the **Default Svg Sprite** and **Default Svg Icon** properties. For example, a User icon can indicate if the entry is a Contact record. Omnistudio supports the Lightning Design System icons. To dynamically display different sprites, map values of the **SVG Controlling Element** to SLDS icons in the **SVG Controlling Element Map**.

 **Note** Dynamic SVG Icon display works only if the SVG Controlling Element is of type Text.

Replace the Default SVG Sprite and Icon

Replace the default SVG sprite and icon.

1. Search for an [SLDS icon](#), and note the category and the icon name.
2. In the **Default SVG Sprite** field, enter the icon's category.
3. In the **Default SVG Icon** field, enter the name of the icon.
4. Preview the Omniscript to view the new default SVG icon.

Display SVG Icons Dynamically

An Edit Block can dynamically display different SVG Icons for some records based on a Text element value.

1. In the Edit Block's **SVG Controlling Element** property, enter the name of a Text element in the Edit Block.
-  **Note** Dynamic SVG Icon display works only if the SVG Controlling Element is of type Text.
2. In the Svg Controlling Element Map, click **Add SVG Icon**.
3. In the **Value** property, enter a possible value for the SVG Controlling Element.
4. Search for an [SLDS icon](#), and note the category and icon name.
5. In the **Svg Sprite** property, enter the icon's category.
6. In the **Svg Icon** property, enter the icon's name.
7. Preview the Omniscript and enter the value for the SVG Controlling Element to test the icon display.
8. (Optional) Repeat the steps for additional SVG Controlling Element values.

Set a Select Mode in Edit Block

Select modes enable users to change an Edit Block entry's boolean value in the data JSON by clicking on Edit Block entries.

Select Mode is only available for Cards, LongCards, and Table.

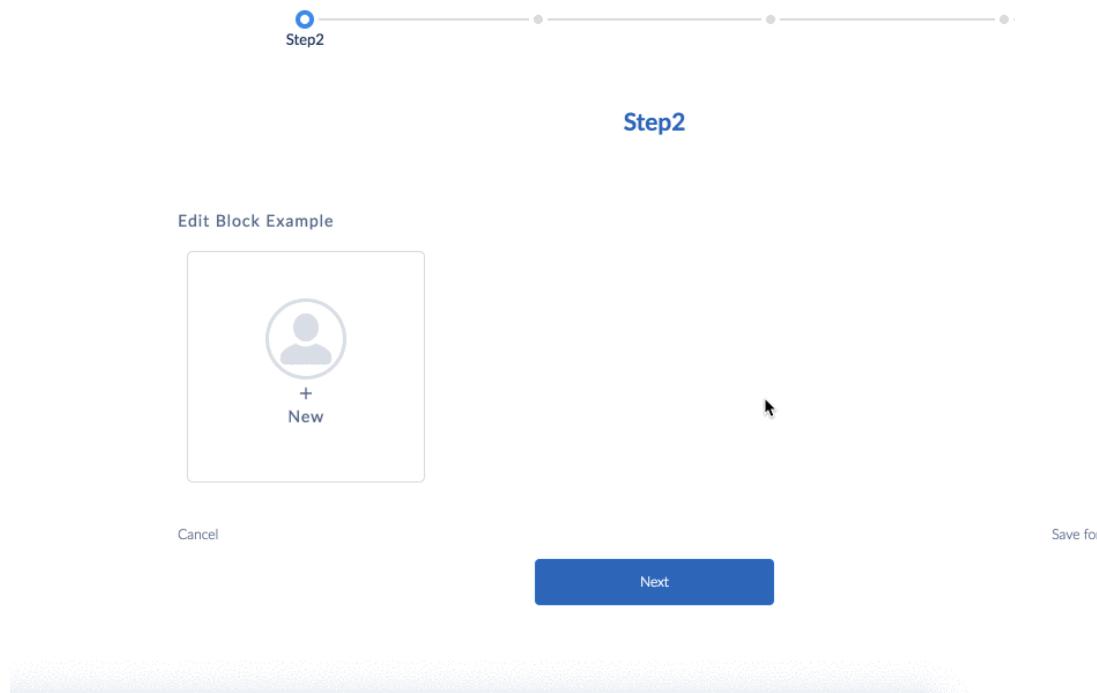
1. Add a Checkbox element into the Edit Block and name the element.
For example, Element Name = PaidInFull.
2. Hide the element from the Edit Block by clicking **Edit as JSON**, and setting *Hide* equal to *true*.
3. In the Edit Block, set **Select Mode** equal to **Single** or **Multi**.
4. Enter the Checkbox element name in the **Checkbox Element Name** field.
5. Preview the Omniscript, create Edit Block entries, and select the entries.
6. Click **{ Data }** to open the data panel and view the boolean values for the Edit Block entry.

Work with Formulas in Edit Block

Perform equations and combine strings by using Formula Elements. Place formulas within an Edit Block to access element values within a specific Edit Block entry. Place Formula outside of the Edit Block in the

same Step to access an element value for all of the Edit Block entries.

1. Add a Formula element to the Edit Block.
2. Using the merge field syntax, add the Edit Block's Element Name and the Element Name for the element you want to access in the Formula. For example, `%EditBlock:FirstName%`. For more information on merge fields, see [Access Omniscript Data JSON with Merge Fields](#).
3. Add a `| n` after the Edit Block element name to set the formula to only access the merge field's value for its instance. Using the `|n` syntax is necessary because the Edit Block stores entry values in an array format. The resulting syntax is `%EditBlock|n:FirstName%`.
4. Concatenate values by combining them using the `|n` syntax. For example,
`%EditBlock|n:FirstName% + " " + %EditBlock|n:LastName%`.



You can easily validate the Edit Blocks by using formulas. For example, EditBlock1 has fields Name, Xfield, Yfield, and sizefield.

To validate EditBlock1, use these formulas:

- Number of rows in EditBlock1: `COUNT(%EditBlock1%)`
- Verify if there's a row with value `X = Test : IF(CONTAINS(%EditBlock1:Xfield%, "Test"), true, false)`
- Verify if all `Y` columns are null: `IF(%EditBlock1:Yfield%==NULL, true, false)`
- Total sum of sizefield: `SUM(%EditBlock1:sizefield%)`

 **Note** When working with validations, formulas, or messaging elements in Omniscripts, always use the `|n` notation to reference fields in Edit and Repeat blocks. Without it, only the first row is evaluated. For example, instead of:

```
%ChildList:RadioChildCitzPick% ==  
%ChildList:RadioChildRelParPick%
```

use:

```
%ChildList|n:RadioChildCitzPick% ==  
%ChildList|n:RadioChildRelParPick%
```

This ensures the validation checks the current row, not just the first one, preventing incorrect results.

Access Index of an Edit Block Array in a Formula

Access each index of an Edit Block array in a formula.

1. Place a Formula outside of the Edit Block.
2. In the Formula editor, add the Element Name using the merge field syntax. For example, to return the sum of all the entries for a Number element in an Edit Block, you would use the following syntax in the formula editor: `SUM(%EditBlock1:NetIncome%)`, where NetIncome is the name of a Number element.
3. (Optional) When using the `vlcEditBlockFS.html` template, display the formula at the top of the Edit Block by adding the Formula element's name to the Edit Block's **Sum Element** property.

Overwrite Edit Block Values in Omniscripts

Overwrite the number of children in an Edit Block in an Omniscript. You can also set the Edit Block's value to `null` in between steps.

1. In the Setup panel of an Omniscript, click **Edit Properties As JSON**, and enter JSON node `"allowOverwrite": true`.
2. Add an Edit Block inside a Step.
3. To populate the Edit Block from an action, add an action, such as Set Values inside or before a step and configure the action.
4. To enable an action to set the Edit Block's value to `null` (empty) in between steps, select **Allow Clear** in the Properties pane of the Edit Block. The action must be in between, before, or after a step.

For example, add a Set Values action in between two steps where the Element Value is `{EditBlock1: null}`. If EditBlock1 is on the first step, when the user reaches the second step and goes back to the first, the contents of EditBlock1 is emptied.

5. To enable the action to remove the Edit Block, select **Allow Delete**.

Override an Edit Block LWC in Omniscripts

Apply custom code to an Edit Block by extending and overriding the Edit Block's lightning web

components. Edit Blocks contain up to three LWC override fields based on the Edit Block's Edit mode.

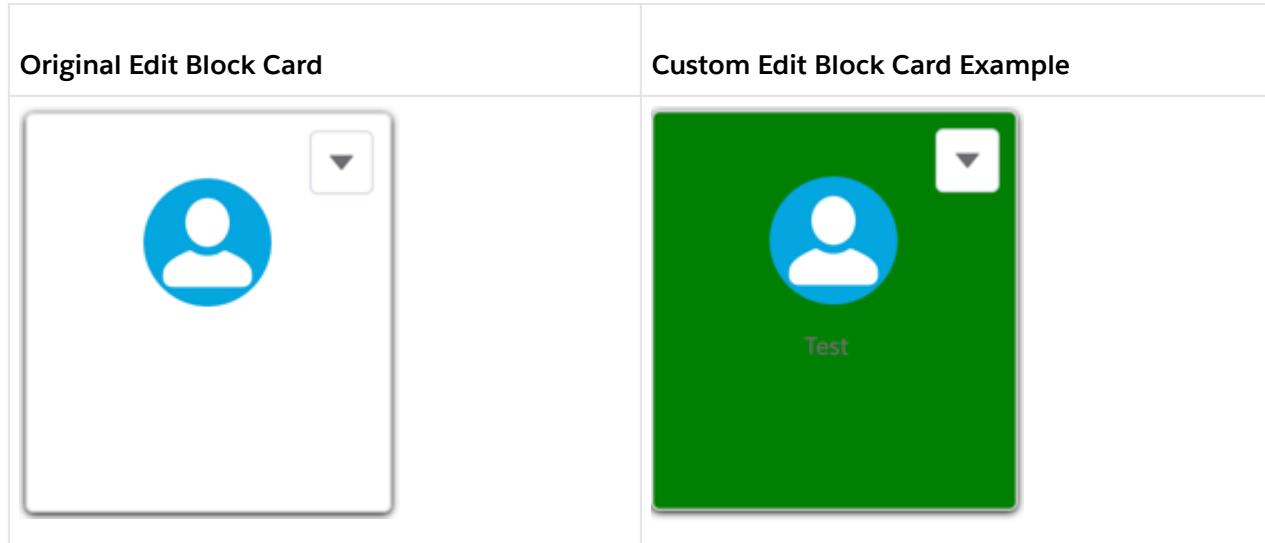
Property	Description	Compatible Edit Modes	Example
LWC Component Override	Customize the Edit Block component by extending the Edit Block component.	All	Extend the Edit Block LWC
Edit Block Label LWC Component Override	Customize the Edit Block's label by overriding the label component.	Cards, Long Cards	Extend the Edit Block Label LWC
Edit Block New LWC Component Override	Customize the Edit Block Card users click to create new entries by overriding the New LWC component.	Cards	Extend the Edit Block New LWC

Extend the Edit Block LWC

Customize an LWC Omniscript's Edit Block by extending the Edit Block LWC.

The Edit Block LWC overrides the entire Edit Block LWC unless the Edit Mode is Cards or Long Cards. Cards and Long Cards include additional extendable LWCs. For information on Cards and Long Cards LWC options, see [Override an Edit Block LWC in Omniscripts](#).

1. In VisualStudio, create a custom LWC or download a custom Edit Block LWC example by clicking [here](#). The code in the custom LWC example changes the background of the Edit Block card to green.



- In the LWC's JavaScript file, extend the Edit Block LWC. Using this code example, replace the NS with the namespace of the package. For information on finding the namespace, see [View the Namespace and Version of Managed Packages](#).

Example:

```

import omniscriptEditBlock from "VLOCITYNAMESPACE/omniscriptEditBlock";
import template from "./editblockCardCustomize.html"
export default class editblockCardCustomize extends omniscriptEditBlock{
    // your properties and methods here

    render()
    {
        if (this.jsonDef) {
            this._hasChildren = this.jsonDef.children.length > 0;
            this._isFirstIndex = this.jsonDef.index === 0;

            if (this._isCards && this._isFirstIndex) {
                // hides the first short card with no children
                if (!this._hasChildren) {
                    this.classList.add(this._theme + '-hide');
                } else {
                    this.classList.remove(this._theme + '-hide');
                }
            }
        }
        return template;
    }
}

```

- Add code to the custom LWC.

4. Deploy the custom LWC to Salesforce. See [Deploy Lightning Web Components](#).
5. Go to the Omniscript that contains the Edit Block LWC.
6. In the Edit Block's **LWC Component Override** field, enter the name of the custom LWC.
7. Activate the Omniscript, and preview your changes.

Extend the Edit Block Label LWC

Customize an LWC Edit Block's Label for the Cards and Long Cards edit modes by extending the Edit Block Label LWC. Exclusive to Cards and Long Cards edit mode, the Edit Block Label LWC overrides the original labels, including global actions.

1. In VisualStudio, create a custom LWC or download a custom Edit Block LWC example by clicking [here](#). The code in the custom LWC example changes the label of a global action to green.

Original Edit Block Label	Custom Edit Block Label
	

2. In the LWC's JavaScript file, extend the Edit Block Label LWC. Using this code example, replace the NS with the namespace of the package. For information on finding the namespace, see [View the Namespace and Version of Managed Packages](#).

Example:

```
import omniscriptEditBlockLabel from "NS/omniscriptEditBlockLabel";
import template from "./editblockLabelCustomize.html"
export default class editblockLabelCustomize extends omniscriptEditBlockLabel{
    // your properties and methods here

    render(){
        {
            return template;
        }
    }
}
```

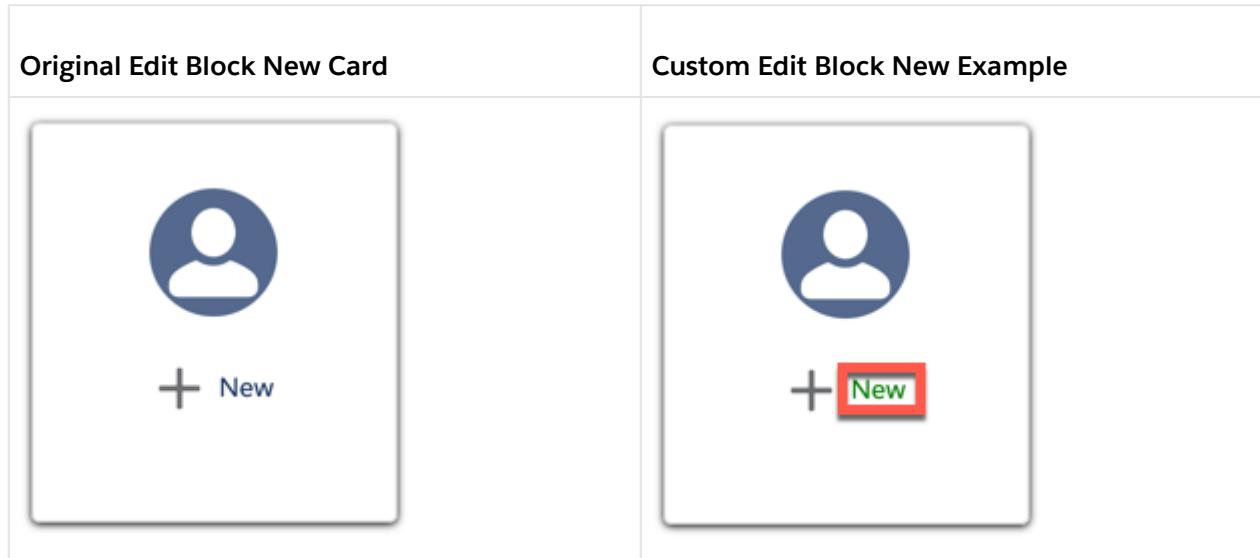
3. Add code to the custom LWC.
4. Deploy the custom LWC to Salesforce. See [Deploy Lightning Web Components](#).
5. Go to the Omniscript that contains the Edit Block LWC.
6. In the Edit Block's properties, select **Edit Block Mode**, and click **Cards** or **LongCards**.
7. In the Edit Block's **Edit Block Label LWC Component Override** field, enter the name of the custom LWC.
8. Activate the Omniscript, and preview your changes.

Extend the Edit Block New LWC

Customize an LWC Edit Block's new entry card by extending the Edit Block New LWC. Exclusive to the Card edit mode, the Edit Block New LWC overrides the new entry card.

1. In VisualStudio, create a custom LWC.

Download a custom Edit Block New LWC example by clicking [here](#). The code in the custom LWC example changes the text color to green.



2. In the LWC's JavaScript file, extend the Edit Block New LWC.

Using this code example, replace the NS with the namespace of the package. For information on finding the namespace, see [View the Namespace and Version of Managed Packages](#).

Example:

```
import omniscriptEditBlockNew from "NS/omniscriptEditBlockNew";
import template from "./editblockNewCustomize.html"
export default class editblockNewCustomize extends omniscriptEditBlockNe
w{
    // your properties and methods here
    render()
    {
        return template;
    }
}
```

3. Add code to the custom LWC.
4. Deploy the custom LWC to Salesforce. See [Deploy Lightning Web Components](#).
5. Go to the Omniscript that contains the Edit Block LWC.
6. In the Edit Block's properties, click **Edit Block Mode** and select **Cards**.
7. In the Edit Block's **Edit Block New LWC Component Override** field, enter the name of the custom

LWC.

8. Activate the Omniscript, and preview your changes.

Edit Omniscript Block Properties

This section contains information on the Edit Block properties.

Property	Description
Allow Clear	Allows values in Edit Block to be set to null using the Set Values action.
Allow Delete	Enables users to access an Action that deletes an entry.
Allow Edit	Enables users to access an Action that edits an entry.
Allow New	Enables users to access an Action that creates a new entry in the Edit Block.
Delete Label	The text label that appears for the Delete Action.
Edit Block Mode	Exclusive to LWC Omniscripts; sets the styling for Edit Block entries.
Edit Label	The text label that appears for the Edit Action.
New Label	The text label that appears for the New Action.
Default SVG Sprite	The name of the Default SVG Sprite category. By default, this field is set to the utility category.
Default SVG Icon	The name of the Default SVG Icon that is within the Default SVG Sprite's category. By default, this field is set to the user icon.
SVG Controlling Element	The name of a Text element in the Edit Block with a value that dynamically controls whether an alternative SVG Icon displays.

Property	Description
Sum Element	The Sum Element displays the value for a Formula placed outside of the Edit Block.
Select Mode	Exclusive to LWC Omniscript, Select Mode enables users to set a Checkbox 's boolean value to <i>true</i> by selecting Edit Block entries. Select Mode is only available for Cards, LongCards, and Table Edit Blocks.
Checkbox Element Name	Enter the name of a Checkbox element in the Edit Block. This element enables Select Mode to function.
sObject Mapping	Enables users to use Vlocity's default actions to map Edit Block entries to Salesforce records.

For information on common properties, see [Common Omniscript Element Properties](#).

When to Use Omniscript Edit Blocks and Blocks

To decide when to use edit blocks and blocks, check if you need to limit repeating the block, directly modify sObjects, show fields when the block is collapsed, display the block in different layouts, or use the custom LWC element.

Omnistudio offers a few block types that meet different needs based on the data that must be grouped together. Edit blocks and blocks are two such types that offer the flexibility you need to structure information in an Omniscript. They both group sets of fields or records and can repeat for however many records there are. They both support extensions to standard Omniscript elements.

Here are some key considerations based on various features and their availability or behavior in each block.

Limit Records

Blocks have a parameter that allows you to limit how many repeats you can have of that block. Edit blocks don't have this capability. However, you can enforce this capability with messaging, formulas, and set errors elements with either block type.

Link to SObjects

Edit blocks allow you to link the recordset directly to an SObject, allowing you to directly modify data in that object while using the edit block. Blocks don't have this capability.

Fields Shown on Collapse

Edit blocks allow you to have different modes of how the record looks in a collapsed mode and allows you to choose fields to display when collapsed. Blocks don't show any fields when collapsed, only its label.

LWC Modes

Edit blocks have various LWC modes that can be used to change the look and feel of the cards. For instance, use the Cards mode to show information in a square-shaped card and the LongCards mode to show it in a rectangular card. Learn more about modes in the LWC Modes section in [Configure an Edit Block in Omniscripts](#).

Create an Omniscript Questionnaire with a Radio Group

With an Omniscript's radio group, create and display questions in a questionnaire format.

Radio Group provides the same options for each question. To provide different options for each question, see [Add Options for Selects, Multi>Selects, and Radio Buttons](#).

1. In the Radio Group's properties, under Options, add *Values* and *Labels*. These labels represent the user's selectable options.
 **Note** Using a value from a Radio Group as a conditional in another element requires adding the Radio Group's element name as the parent node. Example Syntax: *ParentNode:ChildNode*
2. To select all of the fields at once when an option is selected, select **Set All**.
3. Under Radio Group Questions, add questions into the **Label** field and enter **Values** to store the user's response. The Labels represent the questions that display to the left of the selectable options.
4. If needed, set the **Radio Labels Width**.
Changing the width of Radio Group labels changes the width of the options labels.

Add Steps to an Omniscript

Each Step element in an Omniscript presents a page to the user that prompts for input or displays information. For navigation, all Steps except the first have a Next button, and all except the last have a Previous button.

You can drag any Omniscript element into a step except a child Omniscript or another step. An action element in a step renders as a button that performs the action when clicked. Whether fields are required

or validated isn't set at the step level, but in the properties for the elements within the Step. However, if a required field has no value or a field fails validation, the user can't proceed to the next step.

1. From the elements panel, select the default step or drag a **Step** element to the canvas.
2. From the properties panel of a step, update these properties:

Chart Label	Label for the step in the Step Chart, which lists all the steps and shows which step the user is in. If no Chart Label is specified, the Field Label is used.
Instruction	Add instructions to explain to the user what to do in the step. Supports merge fields, rich text, and images.
Allow Save for Later	Displays the Save for later button on the page if selected, which is the default.

3. Configure the buttons in the step:

Previous Label	Label for the button to move to the previous page. The default label is Previous.
Next Label	Label for the button to move to the next page. The default label is Next. You can change it to a value such as Add to Cart or Place Order.
Previous Width, Next Width	Width of the button on the page, with a range of 1-12. The default value is 3.
Save Label	Label for the Save for Later button to save and resume an Omniscript. The default label is Save for later.
Save Message	Label for the Save for Later confirmation prompt. The default is <i>Are you sure you want to save it for later?</i>

4. Configure Knowledge Options, Error Messages, Messaging Framework, and Conditional View, as required.

Hide the Next and Previous Buttons

Hide a step's Next and Previous buttons when using auto-advance options or when you do not want the user to navigate steps.

1. In the Step's properties, expand the **Button Properties** section.
2. Drag the Previous or Next **Width** slider to **0**.
3. Save your Omniscript and preview the new behavior.

Add a Type Ahead Block in an Omniscript

In edit blocks, suggest possible entries as a user types in a field, as an autosuggest or autocomplete.

1. Add the **Type Ahead Block** element to the Omniscript canvas.



Note If a Type Ahead Block exists in a Block element, only one Block element is permitted. Adding multiple nested Blocks results in errors.

2. Add an Action element into the Type Ahead Block to retrieve data or seed the Type Ahead Block with data.

Type Ahead Blocks only support using a single action.

Supported Data Sources	Description
Data JSON	Accesses an array of data present in the Omniscript's data JSON.
Omnistudio Data Mapper Extract Action	Retrieves internal or external Salesforce data.
Google Maps Auto Complete	Calls a Google Maps API to retrieve google maps data.
HTTP Action	Retrieves external data.
Remote Action	Retrieves data from an Apex class.

3. Add input elements into the Type Ahead Block to map response data.

To hide an element, such as an Id, from the user, prefill it with data from the TypeAhead using these steps:

- a. From the element properties panel, click **Edit as JSON**. The JSON definition of the element is displayed.

- b. Set the **hide** property's value to **true**.
 - c. Check the box labeled **Available for Prefill When Hidden**.
4. Configure additional properties in the Edit Block to set a user's data access.
5. Add custom styling and behavior by overriding LWC Type Ahead Block components.

The **LWC Component Override** field only overrides the Type Ahead Component.

Type Ahead Block LWCs	Description
omniscryptypeahead	Overrides the Type Ahead component that lists the data results.
omniscryptypeaheadBlock	Overrides the Type Ahead Block component that embeds the Type Ahead component.
progressBar	LWC Type Ahead Blocks display a progress-bar when retrieving data results. For information on customizing the progress-bar, see Base Omnistudio LWC ReadMe Reference .

Use a Type Ahead Block with an Omnistudio Data Mapper

Retrieve an object record using a Type Ahead Block with a Data Mapper. For example, type a few letters of an account name in the Type Ahead Block. The Data Mapper finds account records with names containing those letters.

Use Type Ahead Block With Data JSON

A Type Ahead Block can use a JSON node as its source instead of an Omnistudio Data Mapper, API, HTTP, or Remote action. This JSON node can store data from any action that outputs a list. The response from a Type Ahead Block that uses a JSON node is made client-side, eliminating the need to make server calls.

Use Google Maps Autocomplete in Omniscripts

Retrieve Google Maps data and display a location in the Type Ahead Block by using the Google Maps API.

Make an HTTP Call from a Type Ahead Block

Retrieve data from a URL in your Type Ahead Block using an HTTP Action.

Use a Type Ahead Block with an Omnistudio Data Mapper

Retrieve an object record using a Type Ahead Block with a Data Mapper. For example, type a few letters of an account name in the Type Ahead Block. The Data Mapper finds account records with names containing those letters.

-  **Note** To prevent users' type-ahead selections from propagating to other fields, make sure to map type-ahead data to individual elements. Create one Data Mapper for each Type Ahead Block, or put one Data Mapper inside a Block or Edit Block element and select **Copy elements (Enable Repeat)** in the designer for a managed package.

Type Ahead Blocks usually use Name but can use another field. The following task steps use Name.

[Download an example](#) Type Ahead Omniscript and its Data Mapper Extract that you can import.

1. Create a Data Mapper Turbo Extract or a Data Mapper Extract with a filter of `Name LIKE Key`. If needed, add other filters.
2. Add a Step component to your Omniscript, or use the first Step.
3. [Add a Type Ahead Block](#) to the Step in your Omniscript.
4. In the Type Ahead Block Properties, set the **Typeahead Key** to `Name`.
5. Drag a Omnistudio Data Mapper Turbo or Omnistudio Data Mapper Extract element into the structure of the Type Ahead Block.
6. In the Data Mapper Action Properties, set the **Data Mapper Interface** to the name of the Data Mapper Turbo Extract or Extract you created.
7. To pass the Type Ahead Block input to the Data Mapper, in the Data Mapper Extract Action Properties, go to **Input Parameters**. Set one **Data Source** to the **Name** of the Type Ahead Block, for example `TypeAhead1`. Set the corresponding **Filter Value** to `Key`.
8. Set any other **Input Parameters** that the Data Mapper Extract requires for other filters.
9. For a Data Mapper Turbo Extract, make sure the **Response JSON Path** in the Omniscript's Data Mapper Turbo Action matches the Data Mapper Turbo Extract's **Extract Output Path**.
10. If needed, add fields to the Type Ahead Block.

To populate the fields, set the element names to match the Data Mapper Turbo Extract or Extract's Output JSON Path fields.

For more information on populating elements, see [Load Salesforce Data into an Omniscript Using an Omnistudio Data Mapper](#).

11. To populate subsequent Omniscript elements with data from the Type Ahead Block's Data Mapper, use `%$StepName:TypeAheadName-Block:DROutputField%`, for example
`%Step1:TypeAhead1-Block:AnnualRevenue%`.

Use Type Ahead Block With Data JSON

A Type Ahead Block can use a JSON node as its source instead of an Omnistudio Data Mapper, API, HTTP, or Remote action. This JSON node can store data from any action that outputs a list. The response from a Type Ahead Block that uses a JSON node is made client-side, eliminating the need to make server calls.

For example, suppose you want to use an Integration Procedure that [eliminates duplicates in the list](#) the Type Ahead Block displays. You can't add an Integration Procedure to the Type Ahead Block directly as you can do with a Data Mapper. However, you can save the response from the Integration Procedure

Action to a JSON node that the Type Ahead Block can reference. To download an example of an Omniscript with a Type Ahead Block that uses data from this Integration Procedure, click [here](#).

1. Use the **Send/Response Transformations** properties of the action with the source list to save the list to a root level JSON node.

For example, the Integration Procedure that eliminates duplicates outputs a list named `contactMerge`.

In the Integration Procedure Action that calls it, set the **Response JSON Path** to `contactMerge` and the **Response JSON Node** to the node the Type Ahead Block references, such as `NoDupList`.

2. In the Type Ahead Block, select **Use Data JSON**.

3. In the **Data JSON Path**, enter the name of a root level JSON node returned by the response of the action with the source list.

Continuing the example of the Integration Procedure that eliminates duplicates, set the **Data JSON Path** to `NoDupList`.

 **Note** The Data JSON Path only accepts data at the root level of the JSON.

4. To return unfiltered results, select **Disable Data Filter** in the Type Ahead Block's properties.

By default, user input is filtered using a `LIKE` match of the **Typeahead Key** field.

5. To make the Type Ahead Block behave like the Lookup element, select **Lookup Mode**.

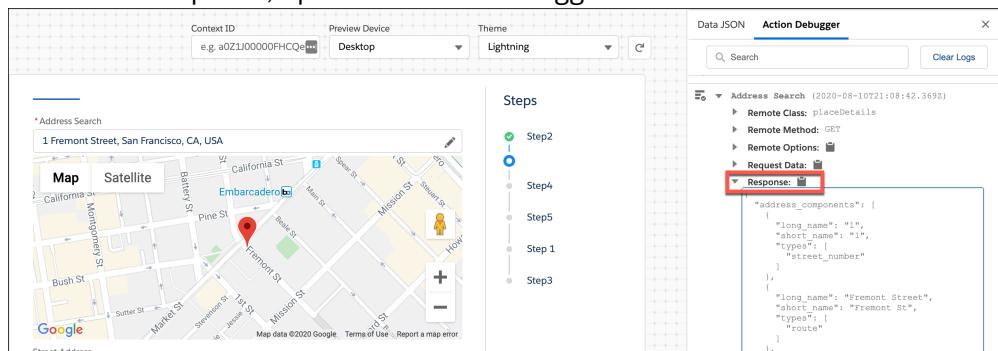
Instead of typing to return a set of values, users can select from a set of values. However, users can enter text using an Input Method Editor (IME) in the lookup mode.

Use Google Maps Autocomplete in Omniscripts

Retrieve Google Maps data and display a location in the Type Ahead Block by using the Google Maps API.

Before you begin, you require a Google Maps API Key to use the Type Ahead Block's Google Maps functionality. See [Google](#).

1. Add a Type Ahead Block to your Omniscript.
2. In Type Ahead Block's **Field Label**, enter the text to display in the Google Maps search field.
3. Select **Enable Google Maps Autocomplete** to display the Google Maps Autocomplete section.
4. In the Google Maps API Key field, enter your Google Maps JavaScript API key.
If you don't have a Google Maps JavaScript API Key, obtain one from [Google](#).
5. If you want to hide the Google Map widget, select **Hide Map**.
6. Select a country in the Country Filter field.
7. To run a search and return a Google response, preview the Omniscript.
8. To view the response, open the Action Debugger.



 **Note** To map the returned Google address with Google Maps Autocomplete to a Salesforce custom address field, under Response, make sure Country and State are in the same format.

Here are the available response nodes:

Responses return unique sets of data based on the configuration.

In addition to the default response nodes, this table shows response nodes that can also be available in the Google response. For detailed information on each response node, see [Google Geocoding API](#) (Google documentation).

Response Node	Description
<code>street_address</code>	A street address
<code>route</code>	A named route, for example, I-5
<code>intersection</code>	A major intersection
<code>political</code>	A political entity
<code>country</code>	A national political entity
<code>geometry</code>	An object that contains longitude and latitude
<code>administrative_area_level_1</code>	A first-order civil entity below the country level, for example, states
<code>administrative_area_level_2</code>	A second-order civil entity, for example, counties
<code>administrative_area_level_3</code>	A third-order civil entity
<code>administrative_area_level_4</code>	A fourth-order civil entity
<code>administrative_area_level_5</code>	A fifth-order civil entity
<code>colloquial_area</code>	A common alternative name for the entity
<code>locality</code>	An incorporated city

Response Node	Description
sublocality	A first-order civil entity below locality. The sublocality can contain additional civil entities in nodes from sublocality_level_1 to sublocality_level_5
neighborhood	A neighborhood
premise	A location, for example, a building
subpremise	A first-order entity below a location, for example, a building
postal_code	A postal code
natural_feature	An important natural feature
airport	An airport
park	A park name
point_of_interest	A point of interest, usually an important local entity, for example, the Golden Gate Bridge
types	An array that contains the response's type, for example, San Francisco is a type of locality

9. To map the data from a Google response node, add an element to the Type Ahead Block. Elements in the Type Ahead Block appear as a selectable Child Element in the Google Transformation section.
10. In the Google Transformations section, click **Add New Mapping**.
11. Select a child element and map it to a Google Response Node.

Refer to the Action debugger's response, and map these additional nested data types by entering the correct notation in the Google Response Node field:

Response Node	Data Type	Notation Example
geometry	Object	<i>geometry:location:lat</i>

Response Node	Data Type	Notation Example
types	Array	<i>types</i> 1

12. To view your mappings, save the Omniscript and run it in preview.

Make an HTTP Call from a Type Ahead Block

Retrieve data from a URL in your Type Ahead Block using an HTTP Action.

1. Add a Type Ahead Block to your Omniscript.
2. Drag the HTTP action into the Type Ahead Block.
3. Configure your HTTP Action by entering your HTTP Path and your HTTP Method.
4. Add fields to your Type Ahead Block for the items that are being returned by the HTTP action.
5. To return unfiltered results select **Disable Data Filter** in the Type Ahead Block's properties.
By default, user input is filtered against a LIKE match.

Omniscript Data Mapper Action Elements

Use Data Mapper actions to retrieve, write, or restructure data from one or more related Salesforce objects.

To add any of these elements, drag the element to the canvas. Add the Data Mapper element in the appropriate logical order, depending on the action. For example, if you're retrieving data, you'll likely want to add it at the beginning of a step. Similarly, writing data to an object is likely done lower or even at the end of a step or in the last step in a sequence.

In the properties for the element, select the appropriate Data Mapper name to call. If it doesn't exist, create a Data Mapper from the properties panel.

[Retrieve Data From Multiple Objects With an Omnistudio Data Mapper Extract Action](#)

To load an Omniscript with data from Salesforce objects, call a Data Mapper Extract. It can get data from one or more related Salesforce objects. You can also use formulas and complex field mappings.

[Write Data to Objects With an Omnistudio Data Mapper Post Action](#)

Create and update sObjects from an Omniscript with an Omnistudio Data Mapper Post Action. The Data Mapper Post Action sends data from the Omniscript to a Data Mapper Load that updates or adds to one or more Salesforce objects.

[Restructure Data With an Omnistudio Data Mapper Transform Action](#)

To restructure, rename, and convert data, use a Data Mapper Transform element. It sends the full Omniscript Data JSON to the Data Mapper and returns the mapping from the Data Mapper. Unlike Data Mapper Extracts and Loads, Transforms don't read or write Salesforce data.

[Retrieve Data From an Object with an Omnistudio Data Mapper Turbo Action](#)

To get data from a single Salesforce object to use in an Omniscript, use a Data Mapper Turbo Extract

Action. Unlike a standard Data Mapper Extract, a Data Mapper Turbo Extract doesn't support formulas or complex field mappings. Because the returned data type is Map<String, Object> and not SObject, you can't use it to update Product2 objects without reserializing. A Data Mapper Turbo is simpler to set up and can run more quickly than a Data Mapper Extract.

Retrieve Data From Multiple Objects With an Omnistudio Data Mapper Extract Action

To load an Omniscript with data from Salesforce objects, call a Data Mapper Extract. It can get data from one or more related Salesforce objects. You can also use formulas and complex field mappings.

1. From the elements panel, drag the **Data Mapper Extract Action** element to the canvas.
2. From the properties panel, select the Data Mapper name to be called.
You can type in the field to filter the list. If the Data Mapper Extract doesn't exist, create a Data Mapper from the properties panel.
3. Provide the name of a JSON node and the value of the JSON node as input parameters for the Data Mapper Extract.
Each Data Source field must match a Data Mapper Extract input parameter. Each Filter Value must match values in the sObject field to which the input parameter refers.
4. Use the Response Transformations properties to trim or rename the output JSON.
See [Manipulate JSON with the Send/Response Transformations Properties](#).

Write Data to Objects With an Omnistudio Data Mapper Post Action

Create and update sObjects from an Omniscript with an Omnistudio Data Mapper Post Action. The Data Mapper Post Action sends data from the Omniscript to a Data Mapper Load that updates or adds to one or more Salesforce objects.

To use a Data Mapper Post Action in an Omniscript, you have three key items to set up:

- The data to add or update
- The object(s) to add or update the data to
- The mapping of the data from the Omniscript JSON to the object and field

When multiple upsert keys are used, the record must match all keys for a match. If multiple actions are needed, use an Integration Procedure to host the Data Mapper Post Action and handle the requests.

First, the Omniscript must have access to or provide the data to update or add to an object. For example, if the data is user input, the Omniscript should have the input fields for that data, such as fields for an account name and industry and for an account contact's first name, last name, and email address.

Second, the Data Mapper Load named in the action must include the Salesforce objects so you can write the data to those objects' fields. For example, the Data Mapper Load has the Account object and the Contact as a linked object via the AccountId field.

Third, in your Data Mapper, map data from the Omniscript to object fields to create or update. With the

account contact example, the mapping includes the first name, last name, and email address. On the right, the JSON node reference is mapped to the object and field on the left.

JSON Node reference	object and field
AccountDetails:AcctName	Account.Name
AccountDetails:Industry	Account.Industry
AccountDetails:ContactFirstName	Contact.FirstName
AccountDetails:ContactLastName	Contact.LastName
AccountDetails:ContactMail	Contact.Email

To trim or rename the input JSON so it has the structure the Data Mapper Load expects, use the Send Transformations properties. See [Manipulate JSON with the Send/Response Transformations Properties](#).

Previewing the Omniscript has the JSON for the example input:

```
{
  "omniscriptId": "0jNSG000000Fqnt2AC",
  "language": "English",
  "type": "Content",
  "subType": "Audit",
  "sId": "0jNSG000000Fqnt2AC",
  "runMode": "preview",
  "theme": "lightning",
  "LanguageCode": "en_US",
  "userProfile": "System Administrator",
  "timeStamp": "2025-05-19T15:43:49.752Z",
  "userTimeZoneName": "America/Los_Angeles",
  "userTimeZone": "-420",
  "userCurrencyCode": "USD",
  "userName": "exampleuser123",
  "userId": "005SG00000JaEzCYAV",
  "omniProcessId": "0jNSG000000Fqnt2AC",
  "localTimeZoneName": "Asia/Calcutta",
  "AccountDetails": {
    "AcctName": "Acme",
    "Industry": "Other",
    "ContactFirstName": "Jon",
    "ContactLastName": "Doe",
    "ContactMail": "Jon_Doe@gmail.com"
  }
}
```

{}

Restructure Data With an Omnistudio Data Mapper Transform Action

To restructure, rename, and convert data, use a Data Mapper Transform element. It sends the full Omniscript Data JSON to the Data Mapper and returns the mapping from the Data Mapper. Unlike Data Mapper Extracts and Loads, Transforms don't read or write Salesforce data.

Use the Send/Response Transformations properties to trim or rename the input JSON so it has the structure the Data Mapper Transform expects. Or you can trim or rename the output JSON for a subsequent Omniscript step.

See [Manipulate JSON with the Send/Response Transformations Properties](#).

Retrieve Data From an Object with an Omnistudio Data Mapper Turbo Action

To get data from a single Salesforce object to use in an Omniscript, use a Data Mapper Turbo Extract Action. Unlike a standard Data Mapper Extract, a Data Mapper Turbo Extract doesn't support formulas or complex field mappings. Because the returned data type is Map<String, Object> and not SObject, you can't use it to update Product2 objects without reserializing. A Data Mapper Turbo is simpler to set up and can run more quickly than a Data Mapper Extract.

The Data Mapper Turbo should extract the fields needed in the Omniscript.

1. Provide the name and value of a JSON node as input parameters that the Data Mapper Turbo Extract requires.
Each Data Source field must match a Data Mapper Turbo Extract input parameter. Each Filter Value must match values in the sObject field to which the input parameter refers.



Note Data Mapper Turbo Extracts return objects in a nested array format.

For example, to set the Context ID to the record ID for accounts, the **Data Source** field is set to Context ID, and **Filter Value** is set to the variable name as defined in the Data Mapper Turbo Extract.



2. Define the response transformations.
 - a. In the **Response JSON Path** field, enter the name of the object returned in the Action's response.

To prefill elements, append a `|` delimiter and a *number* to access the correct instance in the nested array.

Data Mapper Turbo Extract JSON Example	Response JSON Path Example
<pre>{ "Account": [{ "RecordTypeId": "0125w000001 NgyvAAC", "Phone": "5105551223", "Name": "Aileen Lee", "Id": "0015w00002D8iCAAAZ" }] }</pre>	<i>Account</i> 1

- b. To ensure the object isn't nested, enter *VlocityNoRootNode* in the Response JSON Node field. See [Manipulate JSON with the Send/Response Transformations Properties](#).

Omniscript Action Elements

Use action elements in an Omniscript for executing actions, such as get or update data from one or more Salesforce objects, call a series of actions, send emails or documents for signature, redirect to different pages. Action elements can be either rendered as a button when placed in a Step or Block or run remotely if placed between steps. In either case, you can specify a redirect page, where the user can proceed to the next step or action, or if a button, back to the source step.

You can prefill values to multiple fields in Omniscript by using Integration Procedure or Data Mapper action elements. If the result of an Integration Procedure or Data Mapper is an update to the Omniscript JSON, it prefills the fields in the Omniscript at run time.

-  **Note** Use unique names for Omniscript elements and Omnistudio Data Mapper response nodes.

Common Action Element Properties

Review information about Omniscript action element properties.

Calling a Decision Matrix from an Omniscript

Call a decision matrix with specified inputs and return the matching results to an Omniscript.

Delete an Object Record from an Omniscript

Enable users to delete one or more sObject records by using the Delete Action. Use an Object's Record Id to determine which record to delete. The best practice is to use a merge field to refer to an Id or a list of IDs in the data JSON.

Calling Expression Sets from an Omniscript

From an Omniscript, call an expression set created in the Business Rules Engine, and return its results to the Omniscript. Expression sets evaluate conditions, perform mathematical operations, look up

decision tables and matrices, and perform multiple transformations simultaneously.

[Sending Email from Omniscripts](#)

Use emails to send alerts or notifications as part of a workflow, such as sending a summary of a customer's changes to their account information or a report of new customer cases to agents. An Omniscript can send an email from a template or a custom message, and you can use data from Salesforce objects or other data sources in the email address or body fields.

[Fill a PDF with a PDF Action](#)

You can fill an existing PDF form with the PDF action and an Omnistudio Data Mapper.

[Calling Web Services with the Omniscripts HTTP Action](#)

With the HTTP Action, call internal and external web services from an Omniscript without coding or making Salesforce API calls. You can call an HTTP API that allows Apex, Named Credentials, SOAP/XML, or Web, and use the Omniscript's JSON as input.

[Retrieve Data with an Integration Procedure in an Omniscript](#)

From an Omniscript, call an Integration Procedure to retrieve Salesforce data and external data. You can run multiple actions as a headless service (lacking a UI) through JavaScript or Apex Service.

[Open Other Pages from Omniscripts with the Navigate Action](#)

To open various Salesforce pages, apps, and resources from Omniscript, use the Navigate element.

[Calling Apex from Omniscripts with the Remote Action](#)

Call Apex classes from Omniscript using the Remote element, and use the Omniscript's JSON as input.

[Use the DocuSign Signature Action to Sign Documents From Within an Omniscript](#)

Users can sign a document from an Omniscript and download the signed document for their records. After you prepare the DocuSign template and map the fields from the Omniscript to the template using an Omnistudio Data Mapper Transform, you can create a DocuSign Signature Action in the Omniscript. When the action runs, a DocuSign window opens containing the prefilled document. The user must sign or decline to sign the document before continuing the Omniscript.

[Use the DocuSign Envelope Action to Email Documents for Signature](#)

Send an email with prefilled documents for signing or reviewing. It can be sent to one or more recipients. After you prepare the DocuSign template and map the fields from the Omniscript to the template using an Omnistudio Data Mapper Transform, you can create a DocuSign Envelope Action in the Omniscript. When the action runs, a DocuSign Envelope containing the prefilled document is emailed to one or more recipients for signing or reviewing.

[Set Errors in an Omniscript](#)

To handle potential end-user errors, add an error or validation message on one or more elements in a previous step based on conditions from future steps with the Set Errors element. For example, an email address isn't required during the initial step, but in a future step, the user states that they wish to be contacted by email. The Set Errors element runs after the step and returns the user to the initial step with custom error messages.

[Set Values in an Omniscript](#)

Use the Set Values action to set element values in subsequent steps, rename JSON nodes, create dynamic values, and concatenate data. Set Values actions use merge fields to access JSON data in other Omniscript elements.

[Emailing an Omniscript](#)

To email a link to an Omniscript to a Contact, Lead, or User, you create an Integration Procedure that uses a remote action to send the email and call the Integration Procedure from an Omniscript or from Apex code.

[Set Up Access to Remote Action APIs](#)

Configure access to Apex classes used by the remote actions (Vlocity Open Interface Apex classes) called from an Omniscript, Flexcard, Classic Card, or REST API, by profiles.

Common Action Element Properties

Review information about Omniscript action element properties.

Common Action Properties

Property	Description
Label	The label on the button, or, if running remotely, the heading of the Action block.
Validation	Determines whether validation runs on the step before the action is invoked. When set to Step , the action button will be clickable once all required fields within the step contain valid input. Select None to bypass validation.
Invoke Mode	Configure the response behavior of the action. Default: The Action blocks the UI with a loading spinner. Non-Blocking: The Action runs asynchronously, and the response is applied to the UI. Pre and Post Omnistudio Data Mapper transforms, and large attachments are not supported. Fire and Forget: The Action runs asynchronously with no callback to the UI. Pre and Post Data Mapper transforms, and large attachments are not supported. A response will still appear in the debug console but will not be applied to the Data JSON.
Conditional View	All Action elements support conditional view. See

Property	Description
	Conditionally Display Omniscript Elements.
Pub/Sub	Communicate with Omniscript from a custom LWC by sending key-value pairs. Use the events passed in the key-value pairs to trigger custom behavior in a component. See Messaging Framework for Omniscripts .
Session Storage	Send information to a Session Storage object using key-value pairs. A session storage object clears when a page session ends. See Messaging Framework for Omniscripts .
Window Post Message	Send information from an element to a window object in key-value pairs. See Messaging Framework for Omniscripts .

User Message Properties

Actions that run remotely and between steps, or before the first Step is run, contain these User Message properties.

Property	Description
Show a message during execution	Enables an input of custom action messages.
Enable Action Message in the designer for a managed package	
Next Label on Failure	Label of the continue button if the Action fails.
Failure Next Label in the designer for a managed package	
Previous Label on Failure	Label of the Go Back button if the Action fails. The default behavior enables users to return to the previous Step unless there are no previous steps.
Failure Go Back Label in the designer for a managed package	

Property	Description
Success Message	Message displayed upon success.
Post Message in the designer for a managed package	

Send and Response Transformations

Send and Response Transformations are properties available on a variety of Omniscript remote actions. They provide flexibility in trimming and reparenting the request and response JSON. For more information on how to use Send and Response Transformations, see [Manipulating JSON with the Send and Response Transformations Property](#).

Property	Description
Send JSON Path	This property enables one node of the Omniscript Data JSON to be sent, rather than the entire JSON. Specify the node name in this property.
Send JSON Node	This property allows you to specify the root node name in the outgoing response. For example, if your Omniscript had a <i>Customer</i> node at the root, you could relabel the node to <i>CustomerAccount</i> .
Pre-Transform Data Mapper	In situations where a more complete transformation of the JSON is required, you can specify a Data Mapper transform interface. The service will run on the server, and then return the request body to the client for sending.
Post-Transform Data Mapper	Works identically to the Pre-Transform, except the transformation is applied to the response body before merging into the Data JSON.
Response JSON Path	This property allows you to trim the incoming JSON. For example, if the response has a three-level hierarchy, and you only want one of the nodes, you can specific the path you like to extract. The syntax is node:node:node (colon-

Property	Description
	separated).
Response JSON Node	This property allows you to reparent the incoming JSON to a node within the Omniscript JSON. Specific the Omniscript element name which will be the new root node for the response JSON.

Selecting an Invoke Mode

Enable actions to run asynchronously, block users from advancing to a future Step, and modify by selecting an invoke mode.

Invoke Mode enables you to configure the response behavior of an action. It is available in:

- Remote Action
- Integration Procedure Action

Property	Description
Default	The Action blocks the UI with a loading spinner.
Non-Blocking	The Action runs asynchronously, and the response is applied to the UI. Pre and Post Data Mapper transforms, and large attachments are not supported.
Fire and Forget	The Action runs asynchronously with no callback to the UI. Pre and Post Data Mapper transforms, and large attachments are not supported. A response will still appear in the debug console but will not be applied to the Data JSON.

Extra Payload

Extra payload is an additional property available for the actions listed below. The Omniscript can set up extra payload to be sent while making the call. This property supports merge fields. It is available in:

- Remote Action
- Integration Procedure Action

- HTTP Action

Add an Action Message

Display an action message beneath the loading spinner while the action is running.

1. In an Action, expand the User Messages property.
2. Check **Enable Action Message**.
3. In **Action Message**, enter a custom message, or leave the default message.

Calling a Decision Matrix from an Omniscript

Call a decision matrix with specified inputs and return the matching results to an Omniscript.

If you have a decision matrix in the Business Rules Engine, you can call it from an Omniscript with the Decision Matrix element. See [Call a Decision Matrix from an OmniScript](#).

Property	Description
Matrix Input Parameters: Data Source	Name of a JSON node in the Omniscript that contains a value to pass to the Decision Matrix.
Matrix Input Parameters: Filter Value	Name of the Decision Matrix input parameter that accepts the data source value.
Default Matrix Result	Define key/value pairs that specify the default result if the Decision Matrix doesn't return a result.
Execution Date Time	Control the timing of Decision Matrix execution. If this property is blank, the Decision Matrix is executed immediately. Merge field syntax is supported.
Remote Properties: Matrix Name	Name of the Decision Matrix to call.
Remote Properties: Post-Transform Data Mapper Interface	Specify an optional Data Mapper transform to reformat output data from the Decision Matrix.
Remote Options	Define key/value pairs that specify additional options for the Decision Matrix.
Send/Response Transformations: Response JSON Path	Name of the response node in which you want to capture the results of the Decision Matrix. For example, <i>matrixResult</i> .
Send/Response Transformations: Response JSON Node	Name of the Omniscript step where you added the input fields for the Decision Matrix and the Decision Matrix action.

See Also

[Calling Expression Sets from an Omniscript](#)

Delete an Object Record from an Omniscript

Enable users to delete one or more sObject records by using the Delete Action. Use an Object's Record Id to determine which record to delete. The best practice is to use a merge field to refer to an Id or a list of IDs in the data JSON.

1. Preview the Omniscript and identify the JSON path for the record.

For example, this data JSON contains a list of accounts.

```
{  
    "Account": [  
        {  
            "accId": ["001f400000EPQ8o", "001f40000BAkbn"]  
        }  
    ]  
}
```

2. In the Delete Action properties, click **Add sObject to Delete**.
 - a. From the Type of sObject dropdown, select an Object.
 - b. In the Path to Id field, enter the *JSON path* for the record id using merge field syntax.
Using the Account example, the path to delete the array of Account ids is: `%Account:accId%`.
 - c. If required, select **Fail operation if any records aren't deleted**.
In the designer for a managed package, the check box is **All or None**.
3. Enable users to confirm the deletion of a record by configuring these fields in the Confirmation Modal section:
 - a. To display a confirmation modal before deleting the record, select **Display confirmation modal**.
In the designer for a managed package, select **Confirm**.
 - b. Enter the message to be displayed in the confirmation modal.
 - c. If necessary, provide button labels for confirm and cancel actions.
4. Display messages when the operation succeeds or fails by configuring these fields in the Error Messages section:
 - a. In the **Deleted Record Message** field, enter a message that displays when a record is deleted.
In the designer for a managed package, enter the message in the **Entity Is Deleted Message** field.
 - b. In the **Invalid ID Message** field, enter a message that displays when an invalid Id is sent to the Delete Action.
 - c. In the **Deletion Failed Message** field, enter a message that displays when the action fails to delete a record.
 - d. In the **Configuration Error Message** field, enter a message that displays when the Delete Action has a configuration error.

Calling Expression Sets from an Omniscript

From an Omniscript, call an expression set created in the Business Rules Engine, and return its results to the Omniscript. Expression sets evaluate conditions, perform mathematical operations, look up decision tables and matrices, and perform multiple transformations simultaneously.

Property	Description
Remote Properties: Expression Set Name	Name of the Expression Set.
Remote Properties: Configuration Name (in the designer for a managed package)	
Remote Properties: Include inputs	Check whether the Omniscript elements sent to the Expression Set are at the Step level of the Omniscript and not in a block.
Remote Properties: Match input variables	Check whether the Omniscript elements sent to the Expression Set exactly match the Variable IDs in the procedure.
Remote Properties: Pre-Transform Data Mapper Interface	Specify a Data Mapper transform to reformat input data for the Expression Set.
Remote Properties: Post-Transform Data Mapper Interface	Specify an optional Data Mapper transform to reformat output data from the Expression Set.
Remote Properties: Remote Options	Define key/value pairs that specify additional class invocation options.
Remote Properties: Extra Payload	Send additional key/value pairs. Merge field syntax is supported.
Remote Properties: Send only extra payload	Send the key/value pairs for Extra Payload without the Omniscript's JSON.

See Also

[Calling a Decision Matrix from an Omniscript](#)

Sending Email from Omniscripts

Use emails to send alerts or notifications as part of a workflow, such as sending a summary of a customer's changes to their account information or a report of new customer cases to agents. An Omniscript can send an email from a template or a custom message, and you can use data from Salesforce objects or other data sources in the email address or body fields.

For email templates, you need an email template and the object ID to send the email to (such as a

contact, lead, or user). For more information on the What ID, see the [Salesforce Email API documentation](#).

To send a custom email, you can use HTML or plain text for the email body. Also, you can use email addresses or object IDs for the recipient fields with these limits:

- To: 100 addresses
- BCC: 25 addresses
- CC: 25 addresses

For template or custom emails, you can send them from a Salesforce organization-wide email address.

For more information, see [Organization-Wide Email Addresses](#). For attachments, you have access to several sources. If your Omniscript has uploaded or retrieved a file, use the JSON node of a single or list of Salesforce sObject IDs.

- **File Attachments From Omniscript:** The Omniscript JSON node that contains a single or list of Salesforce File sObject IDs.
- **Document Attachments From Omniscript:** The Omniscript JSON node that contains a single or list of Salesforce Document sObject IDs.

You can also attach files in other locations, such as the Salesforce library or an object with a file attached:

- **Content Version ID:** Accepts the ID of a ContentDocument that is sent as an attachment with the Email Action.
- **Document Attachments:** Specifies the Attachment sObjects to add to the email.
- **Attachment List:** Supports attachment IDs merged from Data JSON. For example, `%DocGenId%`. When the email is sent, the attachment will be added to the email.

 **Important** When you embed an Omniscript with an Email action in a Lightning or Experience Cloud page, the response from the action isn't included in the Data JSON. This means that if you add a messaging element after the Email action, it won't accurately show whether the action was successful.

How do you use data from another source?

Let's say you have a Data Mapper extract action that gets information from the contract object for a list of email recipients: In the Omnistudio Data Mapper used in the Omniscript, it maps `Contact:Email` to `ContactEmail`, `Contact:FirstName` to `ContactFirstName`, and so on for other fields. In the Email action, add the merge field `%ContactEmail%` in the address fields. Use the merge fields in the custom email body, such as `Hello, %ContactFirstName%`.

Fill a PDF with a PDF Action

You can fill an existing PDF form with the PDF action and an Omnistudio Data Mapper.

PDF Requirements:

- Adobe Acrobat Pro
- An existing fillable form (either unsecured or you have the password)
- PDF must be a Linearized PDF, Version 1.5 or above. You can save a PDF as Linearized Version 1.7 by clicking **Save as Other**, and then **Reduced Size PDF**, and then **Acrobat 10.0 and later**.
- Salesforce doesn't support the List Box PDF Form field type (Multi-Select fields) in the PDF.
- The PDF Form Field Values for Radio Buttons and Dropdowns must be logically mappable. For example, a Radio Button on a PDF form can't have two options with the same name.



Note Firefox and Safari browsers can't fill some fields without taking additional steps. To use Firefox and Safari, see [Configuring your browser to use the Adobe PDF plug-in](#). Omniscript PDF generation supports only Latin-1 characters. For a full list of these characters, see page 997 of the [PDF Reference, Sixth Edition](#).

- In the PDF, use the base fonts included with Adobe Acrobat, or, if you use other fonts, specify a substitute base font to use when those fonts aren't available.
- The PDF can be launched in a Firefox, Safari, or Chrome browser.

1. Prepare your PDF.
2. Upload the PDF to Salesforce as a document.

Create an Omnistudio Data Mapper Interface to Map Omniscript Data to a PDF

After you create the Omniscript and upload your PDF, you must create a Data Mapper to map the information from the Omniscript to the PDF.

1. From Omniscript, click Preview and fill in test data to populate the JSON of the Omniscript.
2. Copy the Omniscript's JSON data.
3. From the Data Mapper tab, click **New**.
4. For **Interface Type**, select **Transform**.
5. Enter an Interface name.
6. In the **Input Type** field, select **JSON**.
7. In the **Output Type** field, select **PDF**.
8. In the **Target PDF** field, select an existing PDF, and click **Save**.

Map Omniscript Fields to a PDF

Populate a PDF with Omniscript data by mapping the Omniscript JSON data to the PDF.

1. In your Omnistudio Data Mapper Interface, click **Transforms**.
2. Click **Input JSON**, and paste in your Omniscript's JSON.
3. Click **Quick Match** and map nodes in your Input JSON to PDF fields.
4. (Optional) Add mappings one at a time by clicking the + symbol, and mapping an **Input JSON Path** to a **PDF Output Field**.

Add a PDF Action to an Omniscript

Populate a PDF with Omniscript data by using the PDF Action.

-  **Note** The PDF Action is not supported in IE 11 browsers. Edit block doesn't support this action.

Beginning Spring '22, enable older PDF viewers to display text fields in a PDF element using the appearance object property. By default, the `useAppearanceObject` is undefined. To enable, add `"useAppearanceObject": true` in the JSON editor of the PDF Action Properties pane. PDFs larger than 250KB generate slowly. PDFs larger than 1MB can take several minutes to generate and sometimes time out.

1. From the Omniscript Designer, drag the **PDF Action** to the Omniscript.
2. From the Properties pane, in **Document**, select your PDF document.
3. If needed, attach the PDF to a parent record:
 - a. For **Attachment Name**, name the attachment.
 - b. For **Attachment Parent Id**, provide the id of the attachment's parent record id.
4. In the Send Transformations section, from the **PreTransform Omnistudio Data Mapper Interface** picklist, select the Data Mapper that you created earlier.
5. Select **Read Only** to make the filled PDFs read-only.
6. If necessary, change the default date and time formats.

-  **Note** Use `moment.js` formatting. If left blank, the defaults are Time Format: **h:mm a**, Date Format: **MM/DD/YYYY**, Date Time Format: **MM/DD/YYYY h:mm a**.

7. Launch the form in Preview mode, and fill out each field that should map to the PDF.
8. Check the PDF to confirm that the mappings are correct.

-  **Note** To view the generated PDF, users need read access to the Data Mapper Bulk Data object. This access is provided by default, but in case of issues, make sure that users have access to the object.

Calling Web Services with the Omniscripts HTTP Action

With the HTTP Action, call internal and external web services from an Omniscript without coding or making Salesforce API calls. You can call an HTTP API that allows Apex, Named Credentials, SOAP/XML, or Web, and use the Omniscript's JSON as input.

To make sure that the HTTP Action functions correctly without errors related to Content Security Policy (CSP) restrictions, you must add a trusted URL in the Setup, and specify CSP directives and Permissions-Policy directives. When adding this URL, it's important to select all checkboxes under CSP Directives to fully configure the directives. See [Add or Edit a Trusted URL](#).

-  **Note** You must add the Salesforce org's domain into a Remote Site's **Remote Site URL** field. For more information, see [Adding Remote Site Settings](#).

Use the HTTP Action to:

- Send GET, POST, PUT, or DELETE requests to standard REST Endpoints.
- Call Apex REST.
- Access Salesforce Named Credential (OAuth) identity services.

Configure the HTTP settings to call web services.

Property	Description
HTTP Path	The request URL of the API. For example, <code>/NS/v1/application</code> . NS = namespace of the Apex REST API. The path can contain merge fields . For example, to pass a UserId node from the data JSON, add the node name with percentage signs on either side, i.e., %UserId%.
HTTP Method	<p>The method name, which has two parameters. For example, <code>doPost(String fullJSON, String filesMap)</code>.</p> <p>GET, POST, PUT, PATCH, or DELETE.</p>
Endpoint Type	<p>Specifies the endpoint type:</p> <ul style="list-style-type: none"> • Apex – Invokes Apex REST. You must add the Salesforce org's domain to a Remote Site's Remote Site URL field. • Named Credential – Access Salesforce Named Credential (OAuth) identity services. • SOAP/XML – Send requests to standard REST endpoints using XML format for the request and response bodies. • Web – Send requests to standard REST endpoints (unauthenticated or credentials in HTTP headers).
Named Credential	Salesforce named credential. For Named Credential and optionally SOAP/XML endpoint types.
HTTP Headers	HTTP headers specified as key-value pairs. Supports merge field syntax.
Parameters	URL parameters specified as Key/Value pairs.

Property	Description
	Supports merge field syntax.
Encode URI	Uses HTML escape codes in the URI.
Cache response	Caches the response.
Require credentials	Requires security credentials such as a username and password.
Extra Payload	Sends additional Key/Value pairs. Supports merge field syntax.
Send body	Sends the Extra Payload as the request body for a POST action.
Send only extra payload	Enables the action to send Extra Payload's key-value pairs without including the Omniscript's data JSON.

Retrieve Data with an Integration Procedure in an Omniscript

From an Omniscript, call an Integration Procedure to retrieve Salesforce data and external data. You can run multiple actions as a headless service (lacking a UI) through JavaScript or Apex Service.

Before you begin, create an [Integration Procedure](#) to handle Omniscript data.

-  **Note** When you add the same Integration Procedure action to two or more Omniscripts that have a parent-child relationship, set the values in the parent Omniscript, even if the value is intended for use in the child Omniscript.

1. After adding an Integration Procedure element, in the remote properties, enable the Integration Procedure's Queueable Chainable settings:

Property	Value
Future Method	Specifies that the Integration Procedure runs asynchronously, as a Salesforce future method, which can return no data to the calling
Use Future (in the designer for a managed	

Property	Value
package)	Omniscript.
Chainable	Enables the Chainable settings of the subordinate Integration Procedure.
Use Continuation	Enables a subordinate Integration Procedure that calls long-running actions to use Apex continuations.
Queueable	Enables chainable steps in the subordinate Integration Procedure to start queueable jobs.
Queueable Chainable	Enables the Queueable Chainable settings of the Integration Procedure.

2. In **Remote Options**, add this key-value pair:

Key	Value
useQueueableApexRemoting	true

3. Use the Extra Payload property to send additional key-value pairs to the Integration Procedure as input data.
4. Specify any additional Integration Procedure Action properties you need.
5. Use the Send/Response Transformations properties to specify the input and to trim or rename the output JSON.
 - Send JSON Path – Specifies the JSON node that contains the input for the Integration Procedure. This is typically the name of a previous element in the Omniscript.
 - Send JSON Node – Renames the JSON node that contains the input for the Integration Procedure. This is typically the name of one of the Integration Procedure's input parameters.
 - Response JSON Path – Specifies the JSON node that contains the output to return to the Omniscript. By default, all Integration Procedure data is returned.
 - Response JSON Node – Renames the JSON node that contains the output to return to the Omniscript. This is typically the name of a subsequent element in the Omniscript.

See [Manipulate JSON with the Send/Response Transformations Properties](#).

6. Select the response behavior of the action by selecting an **Invoke Mode**.
 - Default – Blocks the UI with a loading spinner.
 - Non-Blocking – Runs asynchronously, and the response is applied to the UI. Pre and Post Omnistudio Data Mapper transforms and large attachments aren't supported. When Invoke Mode is

set to non-blocking, elements using default values won't receive the response because the element loads before the response returns. To map the response to an element, you must set Response JSON Node to VlocityNoRootNode and Response JSON Path to the name of the element.

- Fire and Forget – Runs asynchronously with no callback to the UI. Pre and Post Data Mapper transforms and large attachments aren't supported. A response will still appear in the debug console but won't be applied to the Data JSON.
- When Invoke Mode is set to non-blocking, elements using default values don't receive the response because the element loads before the response returns. You must configure these properties to map the response to an element:

Property	Value
Response JSON Node	VlocityNoRootNode
Response JSON Path	The name of the Omniscript Element receiving the value.

See [Manipulate JSON with the Send/Response Transformations Properties](#).

-  **Note** When an Omniscript invokes an Integration Procedure asynchronously, the Integration Procedure continues running if the user goes to the next step, but not if the user navigates away from the Omniscript.

Open Other Pages from Omniscripts with the Navigate Action

To open various Salesforce pages, apps, and resources from Omniscript, use the Navigate element.

- From the elements panel, drag the **Navigate** action element to the canvas.
 - If the Navigate Action renders in a Step, it becomes a clickable button.
 - If the Navigate Action is between Steps, it fires automatically.
 - If the Navigate Action renders in a Step, select the button style.
 - Determine the style of the button by selecting an [SLDS Button variant](#).
In addition to SLDS Button variants, you can select [Link](#) to render the Navigate Action as an HTML anchor tag.
 - Display an [SLDS Icon](#) in the button.
For example, to add the standard_account Icon, enter *Standard:Account*.
 - If desired, replace the existing entry in the browser history by enabling the Replace property.
If the Replace property is enabled, users can't navigate back to the Omniscript.
-  **Note** By default, a Navigate Action in a Console App opens the page reference type in a new subtab. If you use more than one Navigate Action, only the first navigation action redirects to a new tab. Even if the Replace property is disabled, subsequent Navigate Actions replace the tab. But if the Replace property is disabled, the page reference doesn't replace existing entries in the browser history. Users can navigate back and forth between Omniscripts in the same tab. After

a Navigate Action completes, the Steps and other Omniscript elements that follow it run. However, if the user navigates away from the Omniscript, an asynchronously invoked Integration Procedure stops running.

4. To send messages from the Omniscript to separate Lightning web components, use Pub/Sub messages and add a message key-value pair. Each Navigate Action must pass all required parameters to subsequent Lightning web components. For example, the first Navigate Action passes the `c__ContextId` parameter to a second Omniscript as `c__ContextId=ContextId`. A Navigate Action in the second Omniscript passes the parameter to a third Omniscript as `c__ContextId=%ContextId%`. All Navigate Actions must continue to pass the parameter to subsequent Omniscripts for as long as the parameter is needed.
5. Determine the Salesforce experience that the Navigate Action directs to by selecting a page reference type.

PageReference Type	Description	Task Documentation
App	Direct users from an Omniscript to a standard App, a custom App, or a page within an App. Connected apps aren't supported.	Navigate to an App
Component	Navigate from an Omniscript to a Lightning web component.	Navigate to a Component
Current Page	Trigger a page update on the Current Page after a user completes an Omniscript.	Navigate to the Current Page
Knowledge Article	Display a Knowledge Article in an Omniscript.	Navigate to a Knowledge Article
Login or Logout	Direct users from an Omniscript to a Community Login or Logout page.	Navigate to an Experience Cloud Login or Logout
Named Page	Navigate from an Omniscript to a Named Page or a Community Named Page.	Navigate to a Named Page or Community Named Page
Navigation Item	Navigate from an Omniscript to a page that displays mapped	Navigate to a Navigation Item

PageReference Type	Description	Task Documentation
	content on a CustomTab.	
Object	Navigate from an Omniscript to a standard or custom Object page.	Navigate to an Object Page
Omniscript by pageReference Type	Navigate to an Omniscript by using a page reference type.	Navigate to an Omniscript by Using a Page Reference Type
Omniscript by custom Lightning web component	Navigate to an Omniscript by using a custom Lightning web component.	Navigate to an Omniscript by Using a Custom Lightning Web Component
Record	Navigate from an Omniscript to a Record page.	Navigate to a Record Page
Record Relationship	Navigate from an Omniscript to a page that interacts with a Record Relationship.	Navigate to a Record Relationship Page
Restart Omniscript	Navigate to a new instance of the same Omniscript that resets the Omniscript data.	Restart an Omniscript
Web Page	Open an external Web page from an Omniscript.	Navigate to a Web Page

Navigate to an App

Direct users from an Omniscript to a standard app, custom app, or a page within an app with the App PageReference Type. Verify the app is available in the Salesforce App Launcher.

property	description
Page Reference Type	App
App Name	Enter the appId, or enter the <i>appDeveloperName</i> with the appropriate namespace.

property	description
	<p>For appId, use the AppDefinition object's DurableId.</p> <p>For appDeveloperName, use a concatenation of the app's namespace and developer name. The app's namespace is standard, custom, or a managed package namespace. The Developer name is viewable in the App Manager. For more information on appDeveloperName, see PageReference Type</p> <p>Syntax Examples:</p> <p>Standard: <i>standard__Account</i></p> <p>Custom: <i>vlocity_cmt__CustomAccountApp</i></p>
Additional Parameters	<p>Enter JSON to specify the app page and additional attributes.</p> <p>In the designer for a managed package, enter the JSON in the Target Parameters field. The Target Page field represents the App's pageRef property.</p> <pre data-bbox="842 1174 1445 1300" style="border: 1px solid black; padding: 10px;">{"type": "standard__navItemPage", "attributes": {"apiName": "standard__Account"}}</pre>

Navigate to an Experience Cloud Login or Logout

Direct users from an Omniscript to a Community Login or Logout page for authentication.

To preview an Omniscript with the community login, activate the Omniscript and place it in a Experience Cloud page. For more information, see [Add Your Omniscript to a Lightning Page](#) and [Add Your Omniscript to an Experience Cloud Page](#).

property	description
Page Reference Type	Login
Action	Login or Logout

Navigate to a Component

Navigate to Lightning web components from an Omniscript with the Component PageReference Type.

Property	description
Page Reference Type	Component
Component Name	<p>Enter a component name in the format <code>namespace__componentName</code>, where <code>namespace</code> is the namespace in standard runtime.</p> <p>For Omnistudio, the namespace is <code>omnistudio</code>. This field supports merge fields.</p> <p>This field supports merge fields.</p>
Additional Parameters	<p>Pass additional target parameters by using a URL query string format. Parameters must have a <code>c__</code> prefix. This field supports merge fields.</p> <p>When navigating to a component in a console app, add a Console Tab Icon and a Console Tab Label by setting the <code>c__tabIcon</code> and <code>c__tabLabel</code> parameters. The <code>c__tabIcon</code> accepts SLDS Icon and <code>c__tabLabel</code> accepts plain text. For example, <code>&c__tabLabel=Custom Label&c__tabIcon=standard:account</code> renders the console tab shown in this example image.</p>

Navigate to the Current Page

Update and navigate to the current page from an Omniscript by using the Current Page PageReference Type in a Navigate Action.

property	description
Page Reference Type	Current Page
Additional Parameters	<p>Apply updates to the current page by passing parameters that use a URL query string format.</p> <p>In the designer for a managed package, add</p>

property	description
	<p>parameters to the Target Params field.</p> <p>Parameters must have a <code>c_</code> prefix. This field supports Merge fields.</p> <p>For example, an Omniscript has 10 steps, and you insert a Navigate Action between Step 1 and Step 2 with the page reference type as <i>Current Page</i> and the target as <code>c_step=Step7</code>. At run time, clicking the Next button on Step 1 navigates the user to Step 7.</p>

Navigate to a Knowledge Article

Display a Knowledge Article in an Omniscript by selecting the Navigate Action's Knowledge Article page reference type.

property	description
Page Reference Type	Knowledge Article
Article URL	Enter the Knowledge Article's <code>urlName</code> . This field supports merge fields.
Article Type	<p>Enter the <code>articleType API name</code>. Find the articleType API Name by removing <code>_kav</code> from the API Name of the Knowledge Object.</p> <p>Communities ignore Article Type.</p>

Navigate to an Omniscript by Using a Page Reference Type

Navigate to an Omniscript page reference type. Omniscripts on Experience Cloud site pages must be accessed by using the Community Named Page page reference type.

property	description
Page Reference Type	Omniscript
Omniscript Layout	Select how the Omniscript is styled.
Type	As an Omniscript's name is defined by its Type, SubType, and Language, enter these value for the

property	description
Sub Type Language	specific Omniscript to open.
Fields to Prefill Omniscript	<p>Pass parameters by using a URL query string format.</p> <p>Parameters must have a <i>c__</i> prefix. This field supports merge fields.</p> <p>Example: <code>c__firstName=%firstName%</code></p>

Navigate to an Omniscript by Using a Custom Lightning Web Component

Navigate to an Omniscript by embedding the Omniscript in a custom Lightning web component. You can navigate to an Omniscript this way whether the Managed Package Runtime setting is enabled or disabled.

1. Create a custom Lightning web component (LWC).
2. Add this code to the appropriate button click handler in the component, replacing the type, subtype, language, and context ID with your values.

```
pageReference = {
    type: 'standard__featurePage',
    attributes: {
        featureName: 'omnistudio',
        pageName: 'omniscript'
    },
    state: {
        omniscript__type: os_type,
        omniscript__subType: os_subtype,
        omniscript__language: os_language,
        omniscript__recordId: contextId
    }
};

this[NavigationMixin.GenerateUrl](pageReference).then(url => {
    window.open(url, "_blank");
});
```

Navigate to a Named Page or Community Named Page

Direct users to a standard or Experience Cloud page.

property	description
Page Reference Type	Named Page For an Experience Cloud page, select Community Named Page.
Page Name	Enter the API for the page name. API Names for custom pages must include __c in the API Name. This field supports merge fields.

Navigate to a Navigation Item

Navigate from an Omniscript to a custom tab displaying mapped content with the Navigate Action Navigation Item PageReference Type.

property	description
Page Reference Type	Navigation Item
Tab Name	Enter the Custom Tab's Tab Name and Namespace prefix with the format <code>NS__TabName</code> , where NS is the namespace of the Tab. For more information on tabs, see Create Lightning Page Tabs . This field supports merge fields.

Navigate to an Object Page

Navigate from an Omniscript to an Object page by using the Navigate Action's Object page reference type.

property	description
Page Reference Type	Object
Object API Name	Enter the <i>API name</i> of the object. This field supports Merge fields.
Action	Select an action to invoke. The New action opens

property	description
	a modal dialogue.
Filter Name	Enter the Id of the Object page. This field supports merge fields.

Navigate to a Record Page

Navigate from an Omniscript to a Record page by using Navigate Action's Record PageReference Type.

property	description
Page Reference Type	Record
Record ID	Enter the record ID to navigate to. This field supports Merge fields.
Record Action	Select a record action to invoke. Clone and Edit actions open a modal dialogue.
Object API Name	Enter the API name of the object. This field supports merge fields.

Navigate to a Record Relationship Page

Navigate from an Omniscript to pages interacting with a record relationship by using the Record Relationship page reference type.

property	description
Page Reference Type	Record Relationship
Record ID	Enter the record ID to navigate to. This field supports Merge fields.
Object API Name	Enter the API name of the object defining the Relationship. This field supports Merge fields.
Relationship Object API Name	Enter the API name of the relationship.

Navigate to a Web Page

Navigate to a web page from an Omniscript with the Web Page PageReference Type.

property	description
Page Reference Type	Web Page
URL	<p>Enter the full URL for a web page.</p> <p>For example, to navigate to Google, enter <code>https://google.com</code>.</p> <p>The referring tab with the Omniscript is replaced. To refresh the web page, navigate to or reload the page.</p>

Restart an Omniscript

Navigate a user to a new instance of the same Omniscript. Use this option to enable a user to start over or repeat a process. Place the Navigate Action in a step to render it as a button or outside of a step to run it automatically or conditionally.

property	description
Page Reference Type	<p>Web Page</p> <p>Time tracking and the Messaging Framework aren't supported options for this page reference type.</p>
Conditional View	<p>Add a condition to run, or display, the action conditionally. See Conditionally Display Omniscript Elements.</p>

Calling Apex from Omniscripts with the Remote Action

Call Apex classes from Omniscript using the Remote element, and use the Omniscript's JSON as input.

Before you begin, create an Apex class in Salesforce to call from Omniscript. Include the ID in the SOQL query. That field is required because the Apex class retains the queries and serializes data according to what's actually executed.

1. Add a **Remote** element before a Step, in a Step, or after a Step to load data or send data in the Omniscript.
The Omniscript's data JSON is sent as the input.
2. In the **Remote Class** field, enter the name of the Apex class that the action calls.
3. In the **Remote Method** field, enter a method of the class that the action calls. For example—`validateAddress`.

4. If required, enable the subordinate Integration Procedure that calls long-running actions to use Apex continuations.
5. In the **Pre- and Post-Transform Omnistudio Data Mapper Interface**—optionally select a Data Mapper Transform interface to run before or after the Remote Action.
See [Omnistudio Data Mappers](#).
6. Change the amount of time before the action times out using the Remote Timeout property. In the element, click **Edit Properties as JSON**, and in the `remoteTimeout` property, set a time in milliseconds.
The default timeout is `30,000`, or 30 seconds. The maximum is `120,000`, or 120 seconds.
7. Configure additional properties in the Remote Action.
8. Select the response behavior of the action by selecting an **Invoke Mode**.
 - Default – Blocks the UI with a loading spinner.
 - Non-Blocking – Runs asynchronously, and the response is applied to the UI. Pre and Post Omnistudio Data Mapper transforms and large attachments aren't supported. When Invoke Mode is set to non-blocking, elements using default values won't receive the response because the element loads before the response returns. To map the response to an element, you must set Response JSON Node to `VlocityNoRootNode` and Response JSON Path to the name of the element.
 - Fire and Forget – Runs asynchronously with no callback to the UI. Pre and Post Data Mapper transforms and large attachments aren't supported. A response will still appear in the debug console but won't be applied to the Data JSON.
9. When Invoke Mode is set to non-blocking, elements using default values will not receive the response because the element loads before the response returns. You must configure these properties to map the response to an element:
 - **Response JSON Node:** `VlocityNoRootNode`
 - **Response JSON Path:** The name of the Omniscript Element receiving the value.

 **Troubleshooting Omniscript** In the Troubleshooting Omniscript, the `callOut1` Remote Action element calls an external system to check if the asset is still under warranty. This example references the Troubleshooting Omniscript and the `OmniCallout` and `DataObjectService` Apex classes. It implements the Remote `OmniCallout` class and the remote `checkWarrantyStatus` method. Since this is a sample Omniscript, the method contains a hard-coded response:

Apex Class Detail

Name	OmniCallout	Edit	Delete	Download	Security	Show Dependencies
Namespace Prefix		Status	Active			
Created By	Dave Stone , 6/24/2015 11:03 AM	Code Coverage	0% (0/43)			
Last Modified By	Dave Stone , 6/25/2015 10:44 AM					

Class Body

```

1 global class OmniCallout implements vvelocity_cmt.VlocityOpenInterface{
2     public Boolean invokeMethod(String methodName, Map<String, Object> input, Map<String, Object> output, Map<String, Object> options){
3         System.debug('called correct class');
4         //List<Object>inputVals = input.values();
5         Set<String> keys = input.keySet();
6         for(String key:keys){
7             System.debug('*****Key ' + key + ' value ' + input.get(key));
8         }
9         if(methodName.equals('returnItemOrder')){
10            returnItemOrder(input, output, options);
11        }else if(methodName.equals('checkWarrantyStatus')){
12            checkWarrantyStatus(input, output, options);
13        }else if(methodName.equals('returnItemOrderTest')){
14            returnItemOrderTest(input, output, options);
15        }
16        return true;
17    }
18    public Boolean checkWarrantyStatus(Map<String, Object> input, Map<String, Object> output, Map<String, Object> options){
19        System.debug('called checkWarrantyStatus');
20        String jsonMsg = '';
21        Map<String, Object> lookupItem = (Map<String, Object>)input.get('troubleshootAsset');
22        Map<String, Object> assetItem = (Map<String, Object>)lookupItem.get('assetLookup');
23        String asset = (String)assetItem.get('value');
24        System.debug('*****lookupItem ' + asset);
25        if(asset == 'Cisco Router 800'){
26            jsonMsg = '{"item":"Cisco Router 800", "inwarranty":"T"}';
27            //output.put('warrantyStat', 'This item is still in warranty');
28        }else{
29            jsonMsg = '{"item":"' + asset + '", "inwarranty":"F"}';
30            //output.put('warrantyStat', 'This item is not under warranty');
31        }
32        System.debug(jsonMsg);
33        String serviceReturn = DataObjectService.checkWarranty(jsonMsg);
34    }

```

The callOut2 Remote Action calls the `returnItemOrder` method, which in turn calls the `DataObjectService` class that calls out to a Heroku instance to return an order number:

Apex Class

DataObjectService

[Help for this Page](#)

[« Back to List: Apex Classes](#)

Apex Class Detail

Name	DataObjectService	Edit	Delete	Download	Security	Show Dependencies
Namespace Prefix		Status	Active			
Created By	Dave Stone , 6/24/2015 11:02 AM	Code Coverage	0% (0/51)			
Last Modified By	Dave Stone , 6/24/2015 11:02 AM					

Class Body

```

1 public class DataObjectService {
2     public static final String BASESERVICEENDPOINT = 'https://blooming-anchorage-1503.herokuapp.com'; //upgrade to custom setting
3
4     public static String updateObject(String jsonString){
5         System.debug('*****json passed in ' + jsonString);
6         Http connection = new Http();
7         HttpRequest req = new HttpRequest();
8         req.setEndpoint(BASESERVICEENDPOINT + '/api/v1/echoobject');
9         req.setMethod('POST');
10        req.setHeader('Content-Type','application/json');
11        req.setBody(jsonString);
12        HttpResponse response = connection.send(req);
13        System.debug(response.getBody());
14        return response.getBody();
15    }
16
17    public static String selectObjects(){
18        Http connection = new Http();
19        HttpRequest req = new HttpRequest();
20        req.setEndpoint(BASESERVICEENDPOINT + '/api/v1/echoobject');
21        req.setMethod('GET');
22        HttpResponse response = connection.send(req);
23        System.debug(response.getBody());
24        return response.getBody();
25    }

```

The following class structure can be used to call out to an external system:

```

global with sharing class CustomClassName implements NS.VlocityOpenInterface
{
}

global CustomClassName () { }

```

```
global Boolean invokeMethod(String methodName, Map<String, Object> inputMap, Map<String, Object> outMap, Map<String, Object> options) {

    Boolean result = true;

    try{

        if(methodName.equals('customMethodName')){

            // your implementation, use outMap to send response back to Omniscript

        }

    }

    else if(methodName.equals('customMethodName2')){

        // your implementation, use outMap to send response back to Omniscript

    }

    } catch(Exception e){

        result = false;

    }

    return result;

}

}
```

Use the DocuSign Signature Action to Sign Documents From Within an Omniscript

Users can sign a document from an Omniscript and download the signed document for their records. After you prepare the DocuSign template and map the fields from the Omniscript to the template using an Omnistudio Data Mapper Transform, you can create a DocuSign Signature Action in the Omniscript. When the action runs, a DocuSign window opens containing the prefilled document. The user must sign

or decline to sign the document before continuing the Omniscript.

Before you begin:

- [Prepare DocuSign Documents to Use with Omniscripts](#)
- [Configure a Salesforce Org for DocuSign Integration](#)
- [Prepare a DocuSign Template and Omnistudio Data Mapper Transform for Omniscript Use](#)

1. From the Omniscript elements panel, drag a **DocuSign Signature** action element to a Step or block element, where it renders as a button.

 **Note** Edit Block doesn't support the DocuSign Signature action.

2. In the DocuSign Signature Action properties, click **Add Template**. The Edit Template window opens.
3. Select your DocuSign **Template**.
4. Select a template role.
Roles are defined in the DocuSign template.
5. Enter the name of the Data Mapper Transform.
If the Data Mapper doesn't exist, create a Data Mapper Transform.
6. Click **Save**.
7. Enter a **Signer Name** and **Signer Email**.

The Signer Name and Signer Email fields support merge fields. For example `%FirstName%` `%LastName%` merges the contents of the **FirstName** and **LastName** fields into the **Signer Name** field during runtime.

8. Add an **Email Subject** that recipients receive after signing.
9. If needed, enter a **DocuSign Return Url** that replaces the DocuSign window after signing is complete.
10. If needed, enter display formats for date and time values.

 **Note** To use an Omniscript with a DocuSign Signature Action in an Experience Cloud site, add the **OmniScriptLwcDocuSignViewPdf** Visualforce page permission to any profiles using the Omniscript.

When the DocuSign window opens, a node is written to the Omniscript data JSON. Preview the Omniscript to see the response node in the data JSON:

```
"DocuSignElementName": [
  {
    "status": "Declined",
    "envelopeId": "xyz123"
  },
  {
    "status": "Completed",
    "envelopeId": "xyz123"
  }
]
```

A new `status` and `envelopeId` are generated every time the DocuSign window launches. Statuses include `Completed`, `Declined`, and `In Process`.

! **Important** The DocuSign element doesn't return the response data JSON node when it's added to a Lightning page or Experience Builder site.

If you're a developer overriding the `OmniScriptDocuSignReturnPage` in your Omniscript, post the DocuSign Return Page status back to the Omniscript using `window.parent.postMessage`. For example:

```
window.addEventListener('DOMContentLoaded', function(event) {  
  var domClass = '';  
  var searchStr = event.currentTarget.location.search;  
  
  searchStr = searchStr.substring(searchStr.indexOf('&event='), searchStr.length);  
  searchStr = searchStr.substring(searchStr.indexOf('=') + 1, searchStr.length);  
  
  if(searchStr === 'signing_complete') {  
    domClass = document.getElementById('signing_complete');  
    domClass.style.display = 'block';  
  } else {  
    domClass = document.getElementById('signing_failed');  
    domClass.style.display = 'block';  
  }  
  _window.parent.postMessage(searchStr, '*');  
});
```

Use the DocuSign Envelope Action to Email Documents for Signature

Send an email with prefilled documents for signing or reviewing. It can be sent to one or more recipients. After you prepare the DocuSign template and map the fields from the Omniscript to the template using an Omnistudio Data Mapper Transform, you can create a DocuSign Envelope Action in the Omniscript. When the action runs, a DocuSign Envelope containing the prefilled document is emailed to one or more recipients for signing or reviewing.

Before you begin:

- [Prepare DocuSign Documents to Use with Omniscripts](#)
- [Configure a Salesforce Org for DocuSign Integration](#)
- [Prepare a DocuSign Template and Omnistudio Data Mapper Transform for Omniscript Use](#)

To create a DocuSign Envelope Action:

1. From the Omniscript Designer Build tab, drag a **DocuSign Envelope Action** element from the Actions section onto the canvas. To run the action automatically, place it between steps. To run the action

when the user clicks a button, place it within a step.

 **Note** Edit Block doesn't support the DocuSign Envelope action.

2. In the DocuSign Envelope Action properties, click **Add Template**.
The Edit Template window opens.
3. Select your DocuSign **Template**.
4. Enter the name of the Data Mapper Transform.
5. Enter a **Signer Name** and **Signer Email**.

The Signer Name and Signer Email fields support merge fields. For example `%FirstName%` `%LastName%` merges the contents of the **FirstName** and **LastName** fields into the **Signer Name** field during runtime.

6. Select a template role.
Roles are defined in the DocuSign template.
7. If you have more than one recipient, you can override the routing order set in the template by entering a new order in the **Routing Order** field.
Leave the field blank to use the default routing order.
8. Click **Save**.
The Add Template window closes.
9. Add an **Email Subject** and **Email Body** that recipients receive along with the envelope.
10. If needed, enter display formats for date and time values.

Set Errors in an Omniscript

To handle potential end-user errors, add an error or validation message on one or more elements in a previous step based on conditions from future steps with the Set Errors element. For example, an email address isn't required during the initial step, but in a future step, the user states that they wish to be contacted by email. The Set Errors element runs after the step and returns the user to the initial step with custom error messages.

- The name of a Set Errors element must be unique within an Omniscript and all the reusable Omniscripts it uses.
- The Set Errors action element doesn't support File, Image, Messaging, and Formula elements and doesn't support custom labels.
- Use the Set Errors element if the validation depends on multiple steps in the Omniscript. If you want to use the required field or a formula or messaging element for validation, you can only set requirements and conditions on elements in the current step.
- Avoid using the Set Errors element for validations in a repeat block. In such cases, use a Messaging element. See [Display Messages in Omniscripts](#).

 **Note** When Set Errors is used, it results in two distinct behaviors. First, the error message on an input is transient, which means, it vanishes when the user clicks away from the field but reappears if they try to proceed by selecting Next, if not fixed. Second, if Set Errors is used on multiple inputs, the errors are shown sequentially rather than all at once. The subsequent set of errors will only be

shown after the first set of triggered errors has been resolved.

1. From the elements panel, drag the **Set Errors** element to the canvas.
2. Add a conditional to control when the Set Errors message displays.
Without a conditional the error persists in the Omniscript. See [Conditionally Display Omniscript Elements](#).
3. To set the validation for the Set Errors element:
 - a. In the Element Error Map section, click **Add Element Value**.
 - b. Select the name of the element for which you want to set up validation.
 - c. In the **Value** field, enter the error message that displays to the user.
 - d. If necessary, select an expression for the element value and enter the validation expression in the **Value** field.

Expressions can include:

- Merge fields from a previous step (%elementName%)
- Literal value
- Concatenated values
- Results of formulas and functions
- Expressions that combine the options, such as "Case Status: %caseStatus%"

Set Values in an Omniscript

Use the Set Values action to set element values in subsequent steps, rename JSON nodes, create dynamic values, and concatenate data. Set Values actions use merge fields to access JSON data in other Omniscript elements.

Action elements either render as a button when in a step or run remotely when between steps. Set Values actions are usually between steps.

Use the Set Values action to:

- Access and rename JSON data in other Omniscript elements by using merge fields
- Populate subsequent elements with values
- Concatenate values
- Set values sent into the JSON data from a response or parameter
- Use Omniscript functions and formula operators to set values

Don't use the Set Values action to:

- Transform data
- Relabel incoming JSON data

Instead, use Integration Procedures and Omnistudio Data Mappers.

 **Note** Any text between two percent (%) signs in a Set Values formula is treated as a [merge field](#). To

represent literal percent (%) signs, use the `$Vlocity.Percent` environment variable. The Set Values action doesn't support custom labels.

Download this [datapack](#) and test the behaviors the examples on the following pages provide.

Rename Data JSON Nodes

Simplify and rename JSON node names by accessing the values in an Omniscript's data JSON. Access data sent into the Omniscript's data JSON from a parameter, an action's response data, or an element using merge fields.

Populate Input Elements with Set Values Action

Populate subsequent input elements with values from data JSON, formula operators, functions, concatenations, and static values. Prefill an element directly from an Omnistudio Data Mapper or action response whenever possible. When a previous element assigns a value to a subsequent input element's Name, the subsequent input element is populated with the assigned value.

Concatenate Values with Set Values

Create dynamic values by concatenating JSON node values.

Formulas and Functions with Set Values

Generate dynamic values by evaluating data with formulas and functions to return specific values.

Set and Access Nested Data

Set and access nested data by editing the Set Values JSON directly and using specific syntax.

See Also

[Load Data into Omniscript Elements](#)

Rename Data JSON Nodes

Simplify and rename JSON node names by accessing the values in an Omniscript's data JSON. Access data sent into the Omniscript's data JSON from a parameter, an action's response data, or an element using merge fields.

1. From Omniscript Designer, click **Preview**.
2. Preview the Omniscript and view the resulting data JSON.
3. Locate the JSON node in the data JSON that you wish to rename.
4. Add a Set Values Action to the Omniscript.
5. In the Set Values Action, click **Add Element Value**.
6. In the **Element Value Map**, in the **Element Name** field, enter the new JSON node name.
7. In the **Element Value Map**, in the **Value** field, enter the name of the JSON node you wish to rename using merge field syntax to map it to the new JSON node.

Here's an example of a Set Values action's **Element Value Map** that renames responses and parameters:

Element Name	Value
FirstName	%FirstNameHTTPResponse%
LastName	%LastNameHTTPResponse%
Age	%AgeParameter%

- Preview the Omniscript again and view the new JSON node name.

Populate Input Elements with Set Values Action

Populate subsequent input elements with values from data JSON, formula operators, functions, concatenations, and static values. Prefill an element directly from an Omnistudio Data Mapper or action response whenever possible. When a previous element assigns a value to a subsequent input element's Name, the subsequent input element is populated with the assigned value.

For example, suppose a Set Values action has an **Element Value Map** that assigns an **Element Name** of `FirstName` with a **Value** of `John`. A subsequent Text element with a **Name** of `FirstName` populates with the value `John`.

-  **Note** When populating an element using Set Values, data types must match. For example, you cannot set a Text element with a value from a Number element.

- In the Set Values Properties, click **Add Element Value**.
- In the **Element Value Map**, in the **Element Name** field, enter the name of an input element in a subsequent Step element.

-  **Note** When the Element Name maps to an existing element in the Omniscript, the element's Type appears between the Element Name and the Value fields.

For example, the subsequent Step element has Text elements `FirstName` and `LastName`, a Number element named `Age`, and a Date element named `Date`.

- In the **Element Value Map**, in the **Value** field, enter a literal value, a formula, or the name of another JSON node using merge syntax.

Examples:

Element Name	Value
FirstName	"John"
LastName	%LastNameHTTPResponse%
Age	%AgeParameter%
Date	=TODAY()

- Preview the Omniscript to ensure the fields populate correctly.

Concatenate Values with Set Values

Create dynamic values by concatenating JSON node values.

After you add an element value in a set values action, you must map a JSON node or element name to a value. Concatanate the nodes, elements, or literal values with the + operator, and use " " to add spaces where needed.

Element name	Value
Enter a JSON node or element name.	<p>Click fx and enter multiple merge fields or a combination of merge fields and literal values.</p> <p>Example of two merged fields: =%FirstName% + " " + %LastName%</p> <p>Example of a merged field and literal value: =%FirstName% + " " + "Smith"</p>

Formulas and Functions with Set Values

Generate dynamic values by evaluating data with formulas and functions to return specific values.

After you add an element value in a set values action, define the value for a JSON node or element name with a formula or function using **Use Expression For Value**. Add static values, or dynamic values using merge fields, and add an operator.

 **Note** Operators must have a space on each side for the operation to run correctly.

To assign a null value in JSON or to check for null values in a data field, use `$Vlocity.NULL`. Use `$Vlocity.NULL` in these scenarios:

Unset a field value

To remove a value in a field by setting it to null. Use `$Vlocity.NULL` to make sure that the value is explicitly recognized as null by the system.

Conditionally check for null

Use `$Vlocity.NULL` in formulas to detect when a field is null. You can then trigger certain actions accordingly. Example: `IF(%Step1:Multi-select1% == null, "$Vlocity.NULL", %Step1:Multi-select1%)`

Examples of Formulas and Functions in Set Values

Example type	Example Value Formula Syntax
Merge values	= %GrossIncome% - %NetIncome%
Merge And literal values	= %GrossIncome% - 5000
Literal values	= 2 - 2

Set and Access Nested Data

Set and access nested data by editing the Set Values JSON directly and using specific syntax.

After you add an element value in a set values action, use **Edit Properties as JSON** to map a valid JSON structure to the node name created in **Element Name**.

For example, ParentArray in the Element Name is mapped to a **NumberMap** array with three values:

```
"elementValueMap": {
    "ParentArray": {
        "NumberMap": [
            1000,
            2000,
            3000
        ]
    }
},
```

You can also access nested data, and set it to an element name or JSON node.

In this case, the element for the set values action is a new JSON node name or element name.

For the element's value, apply merge field syntax to access the appropriate value. For more information, see [Access Omniscript Data JSON with Merge Fields](#).

Emailing an Omniscript

To email a link to an Omniscript to a Contact, Lead, or User, you create an Integration Procedure that uses a remote action to send the email and call the Integration Procedure from an Omniscript or from Apex code.

For example, an insurance agent who requires details about a prospect can email an Omniscript link to

the prospect. When the recipient of the email clicks the link, their browser displays the Omniscript, containing whatever data the sender has entered. The recipient fills in the required information, enabling the agent to create a quote for them.

The recipient does not need to have a Salesforce account to access the Omniscript but must have a Contact, Lead, or User record containing their email address and the permissions required to view the script. The recipient might be required to log in to view the email.

To email an Omniscript, create these tasks:

- An email template with the required merge fields
- An Integration Procedure containing a remote action that invokes the email-sending method, passing in the required parameters
- In the Omniscript, an Integration Procedure action that calls the email Integration Procedure

The following sections provide details about creating the required components.

Creating the Email Template With the Omniscript Link

Define an email template containing the desired text, which includes a link to the Omniscript. The email template type must be **Custom (without using Letterhead)**.

Before you create the email template, determine the link of the Omniscript to email. See the URL examples in [How to Launch Omniscripts](#).

After giving the new template a name and subject, enter the text of the email in **HTML Value** and include the Omniscript link. Use Salesforce-style variables (formatted as `{!object.field}`) to incorporate Salesforce data, and Omnistudio-style merge fields (formatted as `%fieldname%`) to incorporate data provided by the Integration Procedure with the Email action.

Example of an email body with variables for the recipient's name and a merge field for the sender's name.

```
Hi, {!Contact.Name}:
```

```
Here's a link to the OmniScript you requested:
```

```
https://MyDomainName.lightning.force.com/lightning/page/omnistudio/omniscript?omniscript__type=%Type%&omniscript__subType=%Subtype%&omniscript__language=English&omniscript__theme=lightning
```

```
Thanks,  
%SenderName%
```

Creating the Email Integration Procedure

To send the email, create an Integration Procedure containing a remote action with the following settings:

- Remote Class: DefaultOmniScriptSendEmail
- Remote Method: emailOmniScriptLink

In the **Remote Options** section, define the following required key/value pairs, plus any merge fields required by the email template:

- **Language** : Omniscript language. For multi-language Omniscripts, set to "Multi-Language" and set the **LanguageCode** parameter to the [Salesforce language code](#) .
- **Type** : Omniscript type.
- **Subtype**: Omniscript subtype.
- **emailTargetObjectId**: The Id of the recipient (contact, lead, or user).
- **emailTemplateName** : The Salesforce Unique Template Name.
- **saveAsActivity**: Set to true if you want an activity record created when the email is sent.
- **LanguageCode** : For multi-language Omniscripts, the Salesforce language code.

This remote action creates an Omniscript record with the specified Type, Subtype and Language. Before the Omniscript is saved, all actions prior to the first step are run, which enables you to prefill information about the Contact, Lead, or User. The Remote Action merges the data into the email template and sends the email.

To test the Integration Procedure, go to the **Preview** tab and define an input parameter named "contextId" that contains the Id of a valid Contact, Lead, or User who has a valid email address. When you are satisfied that the Integration Procedure works as desired, activate it.

Calling the Email Integration Procedure

In the Omniscript, at the point where you want the email sent, add an Integration Procedure action with the following settings:

- Integration Procedure Key: Email_Action
- Extra Payload:
 - Key: contextId
 - Value: the Id of the Contact to whom you want to send the email.

The Omniscript must provide the contact Id in the contextId node of the JSON payload that is sent to the Integration Procedure.

To verify that you have configured all settings correctly, preview the Omniscript and check the resulting email.

Set Up Access to Remote Action APIs

Configure access to Apex classes used by the remote actions (Vlocity Open Interface Apex classes) called from an Omniscript, Flexcard, Classic Card, or REST API, by profiles.

For example, after creating a Force.com site, users with the **Site User** profile can access some APIs based on the profile's Apex class access to RestResource Apex classes. Restricting Apex class access ensures that unauthorized users, such as a [guest user](#), can't access classes through the **Vlocity/V1/Invoke** API.

See [Set Apex Class Access from Profiles](#).

Omniscript Display Elements

Use the Omniscript Display elements to display rich text and images on the screen to enhance the user interface. The display elements include Line Break and Text Block.

[Adding a Line Break in an Omniscript](#)

To help separate other elements in a step, you can create a line break in the Omniscript wherever it exists. Elements placed after a line break start on the next line regardless of another element's width. Place the Line break element directly above the element that should begin on a new line.

[Format Text in an Omniscript with a Rich Text Editor](#)

Use a rich text editor to add text, images, and other rich content to your Omniscripts and Flexcards.

Adding a Line Break in an Omniscript

To help separate other elements in a step, you can create a line break in the Omniscript wherever it exists. Elements placed after a line break start on the next line regardless of another element's width. Place the Line break element directly above the element that should begin on a new line.

From the properties panel, add additional padding in pixels below the line break.

A line separates the elements surrounding the Line Break element.

Format Text in an Omniscript with a Rich Text Editor

Use a rich text editor to add text, images, and other rich content to your Omniscripts and Flexcards.

1. From the App Launcher, find and select **Omniscripts**.
 2. Select the Omniscript version that you want to edit.
 3. If needed, deactivate the Omniscript for editing the Omniscript.
 4. To use a rich text editor:
 - From the elements panel, drag a text block to the canvas.
-  **Note** You can't add a table inside a text block element.

- In the Step properties, edit the Instruction text.
 - From the elements panel, drag a Disclosure element to the canvas.
5. In the properties panel, click the pencil icon to open the rich-text editor and add text and images, as needed.

Omnistudio uses the TinyMCE Editor. However, if needed, you can switch to Lightning Rich Text Editor.

See [Differences Between Lightning and TinyMCE Rich Text Editors](#) and [Enable Lightning Rich Text Editor in Omniscripts and Flexcards](#).

Differences Between Lightning and TinyMCE Rich Text Editors

Omnistudio integrates the TinyMCE editor in the standard designer and managed package designer. However, you can switch to Lightning Rich Text Editor, if needed. Review the key differences between these editors. Before implementing any use cases with the Lighting Rich Text Editor, contact Salesforce Customer Support.

Enable Lightning Rich Text Editor in Omniscripts and Flexcards

Omnistudio uses the TinyMCE editor by default in both Omniscripts and Flexcards. You can switch to Lightning Rich Text Editor, if needed.

Differences Between Lightning and TinyMCE Rich Text Editors

Omnistudio integrates the TinyMCE editor in the standard designer and managed package designer. However, you can switch to Lightning Rich Text Editor, if needed. Review the key differences between these editors. Before implementing any use cases with the Lighting Rich Text Editor, contact Salesforce Customer Support.

Rich Text Editor Feature	Available in the TinyMCE Rich Text Editor?	Available in the Lightning Rich Text Editor?
HTML source code	Yes	No
Write HTML of Text Block to Data JSON	Deprecated	Unsupported
Merge fields to reference data JSON	Yes	Yes
Link	Yes	Yes Can only link to public sites.
Images	Yes	Yes

Rich Text Editor Feature	Available in the TinyMCE Rich Text Editor?	Available in the Lightning Rich Text Editor?
	<ul style="list-style-type: none"> Upload files and images to Salesforce documents (up to 1 MB). Files upload to the public Document Uploads folder and are set to Externally Available. Link to files and images from Documents. Omniscripts and Flexcards support all image file types by default. If set, the lightning-file-upload <code>accept</code> attribute filters file types in the user's file browser. If set, the "Don't allow HTML uploads as attachments or document records" security setting disallows <code>.svg</code> image files. See Lightning Web Component File Upload. 	<ul style="list-style-type: none"> Only supports pasting of images. Supports image size up to 1 MB. Support all image types except <code>.svg</code>. Custom URL, selecting an image from image list, alt text, and width and height aren't supported.
Tables	Yes	<p>Yes</p> <p>Only supports pasting of tables. Doesn't support inserting and formatting tables.</p>
Anchor	Yes	<p>Yes</p> <p>Only supports pasting of anchors.</p>
Horizontal Line	Yes	<p>Yes</p> <p>Only supports pasting of horizontal lines.</p>
Special characters	<p>Yes</p> <p>Note: When you want to use percentage inside the rich text editor, type <code>\$v{velocity.Percent}</code> instead of <code>%</code> to render the output properly.</p>	<p>No</p> <p>String containing JSON, such as characters with {}, (), **, aren't supported in text blocks for Flexcards.</p>
Date and time formats	Yes	No
Bulleted list	Yes	Yes

Rich Text Editor Feature	Available in the TinyMCE Rich Text Editor?	Available in the Lightning Rich Text Editor?
Numbered List	<p>Yes</p> <p>Supports numbers, alphabets, and roman letters.</p>	<p>Yes</p> <p>Supports only numbers</p>
Insert Smart Links to Salesforce Knowledge articles	Yes	No
Headers and footers	Yes	<p>Yes</p> <p>Supports only pasting of headers and footers.</p>
Fonts	<p>Yes</p> <p>Font size in pt (points).</p>	<p>Yes</p> <p>Unsupported fonts are removed automatically.</p> <p>Font size in px (pixels).</p>
Fields (FlexCard)	<p>Yes</p> <p>Select fields or enter the field name in the {field name} format, such as {Type}.</p>	No
Styles (FlexCard)	Yes	No
New Document (Clear existing data from the editor and add content from scratch)	Yes	<p>Yes</p> <p>Use Clear All.</p>
Blocks	<p>Yes</p> <p>The rich text editor adds an HTML div tag for each text section. To add a CSS class at the</p>	<p>Yes</p> <p>Supports only paragraphs.</p>

Rich Text Editor Feature	Available in the TinyMCE Rich Text Editor?	Available in the Lightning Rich Text Editor?
	paragraph level, select a class from the Div picklist.	
Reset	Yes	Yes
Custom labels	<p>Yes</p> <p>Enter custom labels in the <code>{Label.customLabelName}</code> format. For example, to add a custom label for AccountName, enter <code>{Label.AccountName}</code>.</p>	No

Enable Lightning Rich Text Editor in Omniscripts and Flexcards

Omnistudio uses the TinyMCE editor by default in both Omniscripts and Flexcards. You can switch to Lightning Rich Text Editor, if needed.

Review the differences between the TinyMCE editor and Lightning Rich Text Editor to see which one suits your needs. See [Differences Between Lightning and TinyMCE Rich Text Editors](#).

1. From Setup, find and select **Omni Interaction Configuration**.
2. Click **New Omni Interaction Configuration** and add these values to the fields:
 - a. Label: EnableLightningRTE
 - b. Name: EnableLightningRTE
 - c. Value: true
3. Test your Omniscript by creating a new version to see if it works as expected with Lightning Rich Text Editor.

Omniscript Function Elements

From the Omniscript function elements, use the formula element to perform calculations that require complex calculations. For array input, use the aggregate element for complex calculations. The Messaging element displays comments, requirements, success, and warning messaging depending on whether the validate expression is true or false.

[Create a Formula or Aggregate in an Omniscript](#)

Create expressions to set calculated values and evaluate data across multiple fields using a Formula or Aggregate element. For example, if you are accepting information for a qualifying life event you may want to create a formula that determines if the date entered for the qualifying life event is within thirty days of today's date.

Display Messages in Omniscripts

Display comments, requirements, and success and warning messaging depending on whether the validate expression returns True or False. Configure the Messaging element after creating an element, such as a formula or aggregate, to display validation messages in the OmniScript. Merge fields are supported in messages.

Create a Formula or Aggregate in an Omniscript

Create expressions to set calculated values and evaluate data across multiple fields using a Formula or Aggregate element. For example, if you are accepting information for a qualifying life event you may want to create a formula that determines if the date entered for the qualifying life event is within thirty days of today's date.

Formula and Aggregate elements support the following constants and types:

- Omniscript elements and JSON nodes, passed in as merge fields—for example, %Element1% or %JSONnode1%
- Numbers and integers—for example, 5 + 3.145
- String literals wrapped in quotes—for example, "ABC" + "DEF"
- Booleans—for example, true || false
- Arrays—for example, "[1,2, 3, 4, 5]"
- Dates—for example, 2/3/24

Formulas with a **Data Type** of *Date* are interpreted according to the **Date Format**, using the [Day.js](#) library. To prevent a date from being interpreted, set the **Data Type** to *Text*.

- Null

The elements behave similarly in an Omniscript. However, the Aggregate element should be used for aggregation purposes, such as averaging or summing elements. Formula elements can use both [Supported Formula Operators](#) and [Omniscript Functions](#). For example, you may want to determine a contract end date by adding 365 days to a date element. You could do this by using the example formula below:

 **example** DATE(YEAR(%DateElement%), MONTH(%DateElement%), DAYOFMONTH(%DateElement%) +365)

If you would like to set a date one week from today's date, you could do so by using the example formula below:

 **example** DATE(YEAR(TODAY()), MONTH(TODAY()), DAYOFMONTH(TODAY()) +7)

1. From the elements panel, drag a Formula or Aggregate element onto the script structure.
2. In the properties panel, under Expression, add Omniscript functions by clicking on the function name; The function pre-populates in the Expression section. To add Omniscript elements inside of the function, use merge fields by enclosing the element in percentage (%) signs.
3. If you plan to use this element in a repeatable block, see [Evaluate Elements in Repeatable Blocks](#).

4. If you are using a Formula element, continue to build the formula using [Supported Formula Operators](#) in the Expression text box.
5. If the element should not appear on the Omniscript UI, click **Hide**.
6. Create additional elements that depend on the results of the Formula.
See [Create Dynamic Forms Using the Conditional View Property](#) and [Display Messaging in Omniscripts](#).
7. To test the Formula or Aggregate, click **Preview**.

[Supported Formula Operators](#)

Formula fields in Omniscript support the following operators:

[Omniscript Functions](#)

You can add functions to the Aggregate and Formula elements in Omniscripts.

Supported Formula Operators

Formula fields in Omniscript support the following operators:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Power
=	Equals
<>	Not equals
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Operator	Meaning
&&	And
	Or
()	Parentheses

For information on supported functions in Omniscript, see [Omniscript Functions](#). For information on adding a Formula field to an Omniscript, see [Create a Formula or Aggregate in an Omniscript](#).

Omniscript Functions

You can add functions to the Aggregate and Formula elements in Omniscripts.

For a list of functions for Omnistudio Data Mappers and Integration Procedures, see [Supported Data Mapper and Integration Procedure Functions](#).

To view a list of supported formula operators that can be used with the Formula element, see [Supported Formula Operators](#).

The tables below detail the different available Omniscript functions and how they work.

Function Name	Example	Details
ABS(number)	ABS(%Friends%) == 6.52534	Returns the absolute value.
BOOLEAN(value)		Returns true or false.
CURRENCY(value)		Returns value formatted for currency. 12345 would display as 12,345.00. If you add a decimal value such as 1.2345, by default, the field shows the value 1.23, with two decimal points. If you require all decimal values to be shown, add a value to the Mask field. In this example, the value #.#### in the Mask field will show all four decimal values.

Function Name	Example	Details
INTEGER(value)	INTEGER(%Friends%) == 6 INTEGER("12.5") == 12	Converts the number given into an integer with no decimal places. Does not round the number up. Also accepts a string.
NULL		Renders an object's value null.
NUMBER(value)	NUMBER(%Friends%) == 128	Converts a string into a number.
POW(number, exponent)	POW(%Friends%, 3) == 274.625	Returns exponent value.
RANDOM()	RANDOM()	Returns a random number between zero and one. This function is commonly used in A/B testing. The RANDOM() function does not accept parameters.
ROUND(number, decimalPlaces)	ROUND(%Friends%, 3) == 6.525	Rounds number to certain defined number of decimal places.

Function Name	Example	Details
AND();	IF((%reset%"Yes" AND(%reboot%"Yes")), "Closed", "Escalated")	Logical operator that performs a function if all conditions are both met.
COUNTIF(values, expression_or_value)	COUNTIF(%MyArray%, >10)	Returns a count of the number of repeated elements if a condition is met.
EQUALS(Field_Name, 'Condition')	IF(EQUALS(%Department%, '401(k)'),	The EQUALS function is used when comparing a field to a particular value or another field.

Function Name	Example	Details
	<pre>"Then Result", "Else Result")</pre>	
IF(EXPRESSION,THEN, ELSE)	IF((%reset%"Yes" %reboot%"Yes"), "Closed", "Escalated")	If EXPRESSION evaluates to True, THEN is returned, otherwise ELSE is returned.
OR()	IF((%reset%"Yes" OR (%reboot%"Yes")), "Closed", "Escalated")	
SUMIF(values, expression_or_value)	SUMIF(%MyArray%, >5)	Returns the sum of all comma-separated elements and value if a condition is met.

Function Name	Example	Details
CASE(value, \$CASE)	<pre>CASE(%Name%, UPPER) == "TONY JONES" CASE(%Name%, LOWER) == "tony jones" CASE(%Name%, SENTENCE) == "Tony jones" CASE(%Name%, TITLE) == "Tony Jones"</pre>	Changes case based on \$CASE type. \$CASE must be LOWER, UPPER, SENTENCE, or TITLE.
CONCATENATE(value1, value2, ..., valueN)	CONCATENATE(%FirstName%, " ", %LastName%) == "Tony Jones"	Concatenates the elements/strings together into a single string.
CONTAINS(input_string,value)	CONTAINS(%FirstName%, "Tony") == True	Evaluates if a value is contained within a string.
SPLIT(text, splitToken, limit);	SPLIT("Tony Jones", " ", 2) ==	

Function Name	Example	Details
	"Tony", "Jones"	
STRING(value)	STRING(%Friends%) == "6.5"	Converts any value into a String.
SUBSTRING(text, startIndex, endIndex)	SUBSTRING("Substring!", 3, 9) == "string"	Extracts the characters from a string between two specified indexes and returns the value.

Function Name	Example	Details
AGE(dateOfBirth)	AGE(%Birthdate%) == 7	Returns an age (in years) with the given date of birth on today's date. Use the Date element.
AGEON(dateOfBirth, futureDate)	AGEON(%Birthdate%,"07-09-2024") == 16	The age of someone (in years) with the given date of birth on a future date. Use the Date element.
DATE(value)	DATE(%DOB%) == Wed Jul 19 1978 16:00:00 GMT-0700 (PDT)	Use a Date element. Returns full JavaScript format.
DATEDIFF(date1, date2)	DATEDIFF(%DOB%,%TODAY%) == 13559	The difference between 2 dates in days. The value will always be a positive integer. Use the Date element.
DAYOFMONTH(date)	DAYOFMONTH(%TODAY%) == 2	Returns the day of the month. Use the Date element.
DAYOFWEEK(date)	DAYOFWEEK(%TODAY%) == 4	Returns the day of the week as an integer. Monday is 1, Sunday is 7.
HOUR(date)		Return the current hour according to local time.
MINUTE(date)		Return the current minute

Function Name	Example	Details
		according to local time.
MOMENT()	MOMENT(%policyStartDate%).add(1, 'year').subtract(1, 'day')	Returns the moment object of Moment.js This can be used to perform complex date formatting, calculations, manipulation, comparisons, etc. The MOMENT() function must contain parameters, for example, MOMENT(NOW()). Appending .calendar() to this function is a recommended best practice.
MONTH(date)	MONTH(%TODAY%) == 9	The month of the year as an integer. Use the Date element.
NOW()	NOW() == Thu Aug 27 2019 16:32:00 GMT -0700 (PDT)	Returns current date and time in full JavaScript format. Milliseconds are always set to 0. The \$Vlocity.NOW environment variable applies the user or org time zone, while the NOW() function applies UTC time.
TODAY()	TODAY() == Thu Aug 27 2019 00:00:00 GMT -0700 (PDT)	Returns today's date. The time is always set to midnight
YEAR(date)	YEAR(%TODAY%) == 2019	Returns the year of the date as an integer. Use the Date element.

Function Name	Example	Details
ARRAY(value1, value2, ..., valueN)	ARRAY(%Child1%, %Child2%, %Child3%, %Child4%) ==	Returns an array of the value of the elements.

Function Name	Example	Details
	[4,3,1,2]	
AVERAGE(array)	AVERAGE(%Age%) == 2.5	Aggregate: returns the average mean value of the numbers provided. Use a repeating element.
AVERAGE(value1, value2, ..., valueN)	AVERAGE(%Child1%,%Child2%, %Child3%,%Child4%) == 2.5	Returns the average mean value of the numbers provided.
COUNT(array)	COUNT(%Age%) == 4	Aggregate: Returns a count of the number of repeated elements.
EXISTS(array_or_value, expression_or_value)	EXISTS([%FirstName%,%LastName%], "Tony") == true	Returns true if the given value exists in one or more elements. For Aggregate, enter the name of a repeating element—for example EXISTS(%FirstName%,"Tony"), where FirstName is repeatable.
MAX(array)	MAX(%AGE%) == 4	Aggregate: Returns the largest value in the numbers provided. Use a repeating element.
MAX((value1, value2, ..., valueN))	MAX(%Child1%,%Child2%, %Child3%,%Child4%) == 4	Returns the largest value in the group of elements provided.
MIN(array)	MIN(%AGE%) == 1	Aggregate: returns the smallest value in the numbers provided. Use a repeating element.
MIN((value1, value2, ..., valueN))	MIN(%Child1%,%Child2%, %Child3%,%Child4%) == 1	Returns the smallest value in the group of elements provided.

Function Name	Example	Details
SUM(array)	SUM(%AGE%) == 10	Aggregate: returns the sum of all values in the repeatable element.
SUM(value1, value2, ..., valueN)	SUM(%Child1%, %Child2%, %Child3%, %Child4%) == 10	Returns the sum of all comma-separated elements and values. Note that when you use a Formula input in an Omniscript with the SUM formula, the Expression text box defaults the precision of decimal values to two. This occurs even if you specify the desired number of decimal values in your formula. For instance, if you use the formula <code>ROUND(SUM(4.216546546, 5.55463551), 7)</code> , the result is 9.77 instead of 9.7711821. To get precise decimal points, use the ROUND formula with a + (plus) sign instead. For instance, you can use the formula <code>ROUND(4.216546546+5.55463551, 7)</code> instead of the one in the previous example to get the desired precision.

Display Messages in Omniscripts

Display comments, requirements, and success and warning messaging depending on whether the validate expression returns True or False. Configure the Messaging element after creating an element, such as a formula or aggregate, to display validation messages in the OmniScript. Merge fields are supported in messages.

- From the OmniScript designer, drag a Messaging element into the structure panel where the message must appear.
LWC OmniScripts don't support the Messaging element in an Edit Block.
- For Validate Expression, enter the expression to validate.
You can evaluate any element or elements, including Formula and Aggregate.

3. In the Messages section, select message types for when the expression evaluates as True or False, and then enter messages.
True and False messaging can be deactivated, and messaging is optional.
4. In Omniscripts, validation runs when a user clicks out of a field by using the onBlur function. To enable the Omniscript to run validation when a user types:
 - a. In the Setup properties, click **Edit as JSON**.
 - b. Add the property `"commitOnChange": true`.
 - c. Preview the behavior.



Note In LWC Omniscripts, the onChange behavior runs after a half-second delay.

Omniscript Input Elements

When users to change, add, or delete data, add an Omniscript element for a specific type of data, such as email addresses, files, dates, passwords. You can also show information and allow users to select from a list of options, such as radio, checkboxes, multi-select, select, and disclosure elements.

To add any of these elements, drag the element to the canvas.

In the properties for an input element, set the name, label (that users see), and other properties.

Omniscripts Checkbox Element

Enable users to select a checkbox by adding the Checkbox element to your Omniscript. For example, map the user's yes or no answer to a true or false field in a record. Checkbox is a boolean input control and returns true or false to the Data JSON.

Omniscripts Currency Element

Enable users to enter a currency amount by adding the Currency element to your Omniscript. Update default format options such as the number of decimal places to display, minimum and maximum currency value, and more.

Custom LWC Element

Add a custom Lightning web component that does not extend an Omniscript element component to an Omniscript using the Custom LWC element.

Omniscripts Date Element

Enable users to select a date from a date picker by adding a Date element to your Omniscript. Set the minimum and maximum dates and the date format.

Omniscripts Time Element

Enable users to select from a list of times by adding the Time element to your Omniscript. Format the display and set the time intervals.

Set Date and Time Ranges

The Date, Date/Time, and Time elements enable users to input dates and times into an Omniscript form. Each element has different properties that enable you to present a selectable range of dates or times to the User.

Omniscripts Date/Time Element

Enable users to select a date from a date picker and a time from a list by adding the Date/Time

element to your Omniscript. Set the minimum and maximum dates, and the date and time formats.

Omniscripts Disclosure Element

To allow users to agree to a disclosure statement, add a Disclosure element to your Omniscript. Enter the copy in a rich text editor.

Enter an Email Address in Omniscripts

Enable users to enter an email address by adding the Email element to your Omniscript. Update the default allowable format with Validation Options.

Query Salesforce Data from an Omniscript with the Lookup Element

Run a query using text input to retrieve Salesforce data using the Lookup element. The retrieved data is returned in value-label pairs and becomes available for selection in a dropdown list.

Omniscripts Number Element

Enable users to enter a number by adding the Number element to your Omniscript. Limit what the user can enter by setting a mask.

Omniscripts Password Element

Enable users to enter a password by adding the Password element to your Omniscript. Set the minimum and maximum lengths of the password.

Omniscripts Multi-Select Element

Enable users to select from multiple items by adding the Multi-select element to your Omniscript. Display options vertically, horizontally, or as an image. Read-only Multi-Select images show in grayscale, but you can use a custom CSS to them in color.

Omniscripts Select Element

Enable users to select from a dropdown by adding a Select element to your Omniscript. Users can enter text to filter the options. Omnistudio pulls the options from an Apex class and method or from a Salesforce object.

Omniscripts Radio Element

Enable users to select one from multiple items by adding the Radio element to your Omniscript. Display options vertically, horizontally, or as an image.

Add Options for Selects, Multi-Selects, and Radio Buttons

To add options for Select, Multi-Select, and Radio Button elements, go to the **Options** section of the element properties and click **+ Add New Option**. If no values are defined for a Select element, an Undefined Value is returned.

Omniscripts Range Element

Enables users to select a number from a specified range by adding the Range element to your Omniscript. Specify the increment values and use static values to define the minimum and maximum values for the range.

Omniscripts Telephone Element

Enable users to enter a phone number by adding a Telephone element to your Omniscript. Limit the length and format of the number the user can enter.

Omniscripts Text Area Element

Enable users to enter multiple lines of text by adding a Text Area element to your Omniscript. Limit the number of characters the user can enter.

Omniscripts Text Element

Enable users to enter a line of text by adding the Text element to your Omniscript. Limit the length of the text using minimum and maximum values and set allowable information using the mask property.

Omniscripts URL Element

Enable users to enter a website address by adding the URL element to your Omniscript. For example, a user can enter `https://salesforce.com`, or `https://www.salesforce.com`, and so on.

Upload Files and Images in Omniscripts

To upload files in an Omniscript, add a File input element. To upload an image, add an Image input element. Details about uploads are added to the Files node of the Omniscript's Data JSON.

Omniscripts Checkbox Element

Enable users to select a checkbox by adding the Checkbox element to your Omniscript. For example, map the user's yes or no answer to a true or false field in a record. Checkbox is a boolean input control and returns true or false to the Data JSON.

To enable the checkbox by default, select **Default Value**.

When you use a checkbox in an edit block, the value of the Checkbox Label field for the checkbox is sometimes not updated in the designer or the preview. It's also not updated in the Elements Inside Edit Block section. If you face this issue, select the checkbox, and click **Edit Properties as JSON**. Update the value of the checkLabel parameter in the JSON.

Omniscripts Currency Element

Enable users to enter a currency amount by adding the Currency element to your Omniscript. Update default format options such as the number of decimal places to display, minimum and maximum currency value, and more.

To override the default allowable format, select **Custom Mask** and enter a new mask in the **Format Override: Custom Mask** field. See [JavaScript Number Formatter](#).

Currency masks require both a decimal separator and a thousands separator, with a different character for each. Separator choices are a whitespace, a comma, or a period.

Here are some examples:

- `#,###.##` – Commas separate the thousands, and a period separates the decimals, with 2 decimal places.
- `#.###,##` – Periods separate the thousands, and a comma separates the decimals, with 2 decimal places.
- `#,###.` – Commas separate the thousands, and a period separates the decimals, with no decimal places. For the mask to ignore decimal places, the period is required.
- `#,###` – Commas separate the thousands, but no decimal separator anchors where the decimal places begin. Therefore decimal places are promoted instead of ignored, and numbers are multiplied by 10 until they're at least four digits long.

In Formatting Options, configure other properties as needed.

-  **Note** Omniscripts support all currency codes, but some currency symbols may not be shown correctly. For instance, the Somalian Shilling with the currency code SOS and symbol Sh.So. The currency itself works but instead of the symbol, the currency code is shown.

- In **Format Override: Decimal Places**, enter a number to change the default number of decimal places to display.
 - In **Minimum Currency Value**, enter the minimum value that a user can enter, such as **1**.
 - In **Maximum Currency Value**, enter the maximum value that a user can enter, such as **10,000**.
 - If required, let users to enter negative values, such as **-15**.
 - If required, hide commas for values greater than three digits. For example, display **1000** instead of **1,000**.
 - Select **Show currency code** to display a currency code based on the logged-in user's locale. For example, if the user locale is en-US, the currency code is USD.
- For information on configuring multiple currencies, see [Manage Multiple Currencies](#).

Custom LWC Element

Add a custom Lightning web component that does not extend an Omniscript element component to an Omniscript using the Custom LWC element.

The Custom LWC element supports these two types of custom Lightning web components:

- Components that extend the OmniscriptBaseMixin component. See [Extend the OmniscriptBaseMixin Component](#).
- Standalone components that run independently from Omniscript. See [Create a Standalone Custom Lightning Web Component](#).

-  **Note** Use Custom LWC Elements in a Step and a non-repeatable Block element that is not nested.

1. From the elements panel, drag a **Custom LWC** element to the canvas.
2. In **Name** and **Field Label**, enter a name and display name for the Custom LWC element.
3. In the **Lightning Web Component Name** field, enter the name of your custom Lightning web component.
4. Select **Standalone LWC** if your custom component does not extend the OmniscriptBaseMixin component.
5. Pass property values into the Omniscript by entering values in these property fields:

Property	Description
Property Name	Enter a property name using the HTML attribute format to pass it into the custom component. For example, the property recordId converts to the

Property	Description
	HTML attribute record-id.
Property Source	Enter a value to pass in the property. Values may use merge field syntax to pass a JSON node. For example, to pass the JSON node ContextId, enter <code>%ContextId%</code> . To avoid Flexcard loading issues, don't use the merge field syntax to reference a custom LWC. For example, if the name of the LWC is <code>customlwc1</code> , don't use <code>%customlwc1%</code> .

Embed Flexcards in an Omniscript

Embed a Flexcard Lightning Web Component in an Omniscript by using the Custom LWC element. Flexcards can receive data from the Omniscript and perform any action available in the Flexcard.

Embed Flexcards in an Omniscript

Embed a Flexcard Lightning Web Component in an Omniscript by using the Custom LWC element. Flexcards can receive data from the Omniscript and perform any action available in the Flexcard.

Before You Begin

1. Ensure your Flexcard includes Omniscript support.
2. (Optional) Configure a Flexcard to receive data from Omniscript.
 1. From the elements panel, drag the **Custom LWC** element into a step.
 2. In the properties panel, in the element's Lightning web component name, enter the Flexcard component's name and prepend `cf` to the beginning of the name.
For example, a Flexcard named `Account Card` must be entered as `cfAccountCard`.
 3. In **Property Name**, enter a property that the Flexcard expects to receive by converting the property to an HTML attribute format. Three options enable you to pass data into the Flexcard from your Omniscript. Each data option requires you to pass data as an HTML attribute. For example, if a Flexcard receives the property `recordId`, you must enter `record-id` in the property name field to pass the property correctly.

Property Name Option	Description
<code>record-id</code>	Pass a record id into a Flexcard using the record-id property. Flexcards use record ids to perform data queries.

Property Name Option	Description
parent-attribute	Pass a parent object containing parent attributes such as Parent.id into the Flexcard. Use merge fields in Flexcard queries and fields.
parent-data AND records	Map data to Flexcard fields directly without running a query. The parent-data property is a boolean that Flexcards uses to determine whether to run a query or parse over a set of records.
listen-os-data-change	Update the Flexcard whenever the passed-in Omniscript data changes.

4. In **Property Source**, using merge field syntax, enter one of these options based on your property name:

Property Name	Property Source
record-id	Enter a JSON node that contains an object. For example, to pass a record ID stored in the ContextId node, enter <code>%ContextId%</code> . To avoid Flexcard loading issues, don't use the merge field syntax to reference a custom LWC. For example, if the name of the LWC is customlwc1, don't use <code>%customlwc1%</code> .
parent-attribute	Enter a JSON node containing an object.
parent-data AND records	Set parent-data's property source to <code>true</code> . Set records equal to an object containing a record or an array of records. Flexcards parses over these records and maps the nodes to Flexcard fields.
listen-os-data-change	Enter <code>true</code> to update the Flexcard whenever the passed-in Omniscript data changes.

5. (Optional) Rerender and refresh the Flexcards save state whenever a user navigates to the step by:

- a. In the Flexcards Designer, disable Omniscript support.
 - b. In the Omniscript Designer, open the Custom LWC element and check **Standalone Mode**. For information on Standalone Custom LWCs, see [Create a Standalone Custom Lightning Web Component](#).
6. (Optional) Beginning with Summer '21, provide Selectable Items in your Omniscript using Flexcards. See [Select Card](#).
7. To pass the Flexcard data to the parent Omniscript, see [Update an Omniscript's JSON Code from a Flexcard](#).
8. Save and Activate the Script.
9. Preview the Omniscript.
-  **Note** Omniscripts using custom Lightning web components must be active to preview the Omniscript.

[Pass Data from an Omniscript to an Embedded Flexcard](#)

To populate data fields and perform actions on a Flexcard embedded in an Omniscript, you can pass data from an Omniscript's data JSON to the Flexcard. Embed a Flexcard inside an Omniscript with the Custom Lightning Web Component element.

[Pass Data from a Flexcard to an Omniscript By Using a Custom JSON Data Source](#)

To pass data from a Flexcard's data JSON to an Omniscript, embed the Flexcard as a custom Lightning web component (LWC) inside the Omniscript.

Pass Data from an Omniscript to an Embedded Flexcard

To populate data fields and perform actions on a Flexcard embedded in an Omniscript, you can pass data from an Omniscript's data JSON to the Flexcard. Embed a Flexcard inside an Omniscript with the Custom Lightning Web Component element.

There are three ways to pass data from the Omniscript to the embedded Flexcard:

Options	Description
Pass a set of records	Use the records global context variable to map data from the LWC Omniscript to an embedded Flexcard.
Pass the recordId	Pass the recordId global context variable from the LWC Omniscript to the Flexcard.
Pass a parent object	Pass a Parent object containing parent attributes to the Flexcard from the LWC Omniscript. The Omniscript is the parent of the embedded

Options	Description
	Flexcard. The Parent global context variable, such as <code>{Parent.Id}</code> , must be used in the Flexcard where the merge field is supported.

-  **Note** To update the Flexcard whenever the passed-in Omniscript data changes, add a **Property Name** of `listen-os-data-change` and a **Property Source** of `true` to the Omniscript's Custom LWC element properties for the Flexcard.

Map Data from an LWC Omniscript to an Embedded Flexcard

After embedding a Flexcard inside an LWC Omniscript, map records from the Omniscript to data fields in the Flexcard. For example, populate a list of Flexcards from an array in an Omniscript. In this scenario, the Flexcard uses data from the Omniscript instead of its own data source.

Pass the RecordId from an Omniscript to Run a Query on a Flexcard

Pass the **recordId** context variable from an LWC Omniscript to an embedded Flexcard to run a query on the Flexcard. The query returns the data that populates the Flexcard data fields and actions.

Pass a Parent Object from an LWC Omniscript to Run a Query on a Flexcard

Pass a Parent object from the LWC Omniscript to the Flexcard. The LWC Omniscript is the parent of the embedded Flexcard. The **Parent** context variable must be used in the Flexcard where merge fields are supported, such as `{Parent.Id}`.

Map Data from an LWC Omniscript to an Embedded Flexcard

After embedding a Flexcard inside an LWC Omniscript, map records from the Omniscript to data fields in the Flexcard. For example, populate a list of Flexcards from an array in an Omniscript. In this scenario, the Flexcard uses data from the Omniscript instead of its own data source.

The data fields on the LWC Omniscript must exist in the Flexcard even if the values are empty. For example, if your Omniscript data JSON has Id, Name, Email, and Phone fields, these field names must exist even if the value of Email or any other fields are empty in the data source of the embedded Flexcard.

1. In your org, go to the Flexcards tab and click a version of a Flexcard to open the Flexcard Designer. Or, create a new Flexcard. See [Create a Flexcard](#).
2. In the Setup panel, select and configure the data source with data fields you want the Omniscript to map.

For example, if your data source is an Omnistudio Data Mapper that gets a list of Accounts, confirm that the Data Mapper Output fields selected include the fields that your Omniscript data JSON maps to. If your Omniscript data JSON has a Name field, so should the Output from the Data Mapper, and so on.

3. Click **Save & Fetch**.

4. From the Build panel, add elements such as Fields, Datatables, and so on onto the canvas. See [Add Elements to a Flexcard](#).
5. In the Setup panel, check **Omniscript Support**.
6. Click **Activate**.
7. Embed your Flexcard in an LWC Omniscript with the Custom LWC element in the LWC Omniscript Designer.
8. In the **Custom Lightning Web Component Properties** section of the Properties panel, add `parent-data` as the **Property Name**, and set the **Property Source** value to `true`.
9. Click **+ Add New Property**.
10. Enter `records` as the **Property Name**, and in the **Property Source** field, enter the object in the data JSON that lists records that map to the records in your Flexcard. Enter the object as a merge field using Omniscript merge field syntax, such as `%objectname%`.
For example, if your list of records is stored in a JSON object called `accounts`, enter `%accounts%`.
To avoid Flexcard loading issues, don't use the merge field syntax to reference a custom LWC. For example, if the name of the LWC is `customlwc1`, don't use `%customlwc1%`.
11. Click **Activate**.
12. Click **Preview** to preview the Flexcard inside the LWC Omniscript.

Pass the RecordId from an Omniscript to Run a Query on a Flexcard

Pass the **recordId** context variable from an LWC Omniscript to an embedded Flexcard to run a query on the Flexcard. The query returns the data that populates the Flexcard data fields and actions.

For example, if your Flexcard has an Omnistudio Data Mapper that gets Account Cases, the fields and actions get updated based on the value of the `recordId` set in the LWC Omniscript data JSON. If you use the `{recordId}` merge field in your Flexcard such as in an SOQL query or a Text element, that data that relies on that merge field is also updated based on the `recordId` coming from the LWC Omniscript data JSON.

1. In your org, go to the Flexcards tab and click a version of a Flexcard to open the Flexcard Designer. Or, create a new Flexcard. See [Create a Flexcard](#).
2. Configure your data source if you haven't already. See [Set Up a Data Source on a Flexcard](#).
3. Add data elements such as Fields, Actions, and so on, from the Build panel onto the canvas. See [Add Elements to a Flexcard](#).
4. In the Setup panel, check **Omniscript Support**.
5. Click **Activate**.
6. Embed your Flexcard in an LWC Omniscript with the Custom LWC element in the LWC Omniscript Designer. See [Embed Flexcards in an Omniscript](#).
7. In the **Custom Lightning Web Component Properties** section of the Properties panel, add `record-id` as the **Property Name**.
 **Note** Because the Property Name is an HTML attribute, it must be written in kebab case, with words separated by a dash. See [Property and Attribute Names](#).
8. Set the **Property Source** to a JSON node that contains a `recordId`. Use Omniscript merge field syntax, such as `%nodename%`, to enter the node as a merge field.

For example, if your recordId is stored in a JSON node called ContextId, enter `%ContextId%`.

To avoid Flexcard loading issues, don't use the merge field syntax to reference a custom LWC. For example, if the name of the LWC is customlwc1, don't use `%customlwc1%`.

9. Click **Activate**.
10. Click **Preview** to preview the Flexcard inside the Omniscript.

Pass a Parent Object from an LWC Omniscript to Run a Query on a Flexcard

Pass a Parent object from the LWC Omniscript to the Flexcard. The LWC Omniscript is the parent of the embedded Flexcard. The **Parent** context variable must be used in the Flexcard where merge fields are supported, such as `{Parent.Id}`.

The parent object in the Omniscript data JSON can be named anything as long as it is an object with fields that match the name of the parent attribute that you want the Omniscript to look for in the Flexcard. For example, the JSON object name can be `parent` or `account`, such as `"parent" : {"Id" : "1234567"}` or `"account" : {"Id" : "1234567"}`.

1. In your org, go to the Flexcards tab and click a version of a Flexcard to open the Flexcard Designer. Or, create a new Flexcard. See [Create a Flexcard](#).
2. Configure your data source if you haven't already in Step 1. See [Set Up a Data Source on a Flexcard](#).
3. Add data elements such as Fields, Datatables, and so on, from the Build panel onto the Canvas.
4. Add a **{Parent}** merge field, such as `{Parent.Id}`, in your Flexcard where merge fields are supported such as in an SOQL query, a Text element, an Input Parameter, and so forth.
5. In the Setup panel, check **Omniscript Support**.
6. Click **Activate**.
7. Embed your Flexcard in an LWC Omniscript with the Custom LWC element in the LWC Omniscript Designer. See [Embed Flexcards in an Omniscript](#).
8. In the **Custom Lightning Web Component Properties** section of the Properties panel, add `parent-attribute` as the **Property Name**, and set the **Property Source** to the parent JSON object that contains parent attributes. Use Omniscript merge field syntax, such as `%objectname%`, to enter the node as a merge field.
For example, if your parent object is stored in a JSON object called account, enter `%account%`.
To avoid Flexcard loading issues, don't use the merge field syntax to reference a custom LWC. For example, if the name of the LWC is customlwc1, don't use `%customlwc1%`.
9. Click **Activate**.
10. Click **Preview** to preview the Flexcard inside the Omniscript.

See Also

[Download a Sample Data Pack that Passes Data from an Omniscript to a Flexcard](#)

Pass Data from a Flexcard to an Omniscript By Using a Custom JSON Data Source

To pass data from a Flexcard's data JSON to an Omniscript, embed the Flexcard as a custom Lightning web component (LWC) inside the Omniscript.

1. Create a Flexcard with a custom data source. See [Create a Flexcard](#).
2. Select Custom as the data source type and add your custom data JSON.
3. Add elements to the Flexcard corresponding to the fields you've specified in your JSON.

4. In the Setup tab of the Flexcard, select the **Omniscript Support** checkbox.
 5. Activate the Flexcard.
 6. Create an Omniscript. See [Create an Omniscript](#).
 7. In the first step, add elements corresponding to the fields you've specified in the Flexcard.
 8. Create a new step and add the Custom LWC input element to it.
 9. Configure the custom LWC with these parameters.
 - a. Lightning Web Component Name: Choose the Flexcard you created.
- Note** Any validation rules applied to the Flexcard input elements doesn't prevent the Omniscript from proceeding. Set up all input validations within the Omniscript.
- b. In the Custom Lightning Web Component Properties section, click **Add Properties**.
 - c. Property Name: *parent-data*
 - d. Property Source: *true*
 - e. Add another property and for Property Name, enter *records*.
 - f. In the corresponding Property Source field, enter the name of the previous step surrounded by the % key. For example, if the previous step is called Step1, enter *%Step1%*.

Custom Lightning Web Component Properties

Property Name	Property Source	
parent-data	true	
records	%Step1%	

[+ Add Property](#)

10. Preview your Omniscript to check if the Flexcard sends the requisite data to your Omniscript.

Omniscripts Date Element

Enable users to select a date from a date picker by adding a Date element to your Omniscript. Set the minimum and maximum dates and the date format.

1. Enter the earliest date the user can select from the date picker, such as `01-01-1950`.

You can pick a fixed date, or you can enter a relative date in the format `TODAY +/- N day(s) /week(s) /month(s) /year(s)`. For example, you can enter `TODAY - 1 day` or `TODAY + 3 months`.



Note The format entered must be the same as specified in the date display format. For example, to set the minimum date to January 1, 1950, if the date format is `MM-DD-YYYY`, enter `01-01-1950`. The Minimum Date field doesn't accept merge fields.

2. Enter the latest date the user can select from the date picker, such as `12-31-2019`.

You can pick a fixed date, or you can enter a relative date in the format `TODAY +/- N day(s) /week(s) /month(s) /year(s)`. For example, you can enter `TODAY - 1 day` or `TODAY + 3 months`.



Note The format entered must be the same as specified in the date display format. For example, to set the minimum date to January 1, 1950, if the date format is `MM-DD-YYYY`, enter `01-01-1950`. The Maximum Date field doesn't accept merge fields.

3. Enter the format to display the date.

For example, if the format is `MM-DD-YYYY`, and the user selects `January 3, 2020` from the date picker, the date displays as `01-03-2020`. For more format options, see [Day.js](#).

4. In the Advanced Options section, update values for the following properties:

- **Data Type:** Specifies how to display the date in the Data JSON.

If set to String, the specified format is applied. String is the default data type.

If set to Date, the date is encoded as an ISO 8601 datetime, such as `2018-01-28T05:00:00.000Z`, in the GMT/UTC time zone.

Using this format instead of a String such as `YYYY-MM-DD` without a time requires a thorough understanding of time zone handling. Midnight of the date is converted from either the org time zone or the user's time zone to GMT/UTC.

- **Format:** Specifies how the date is formatted in the Data JSON when **Data Type** is set to **String**. For example, `YYYY-MM-DD`. In the designer for a managed package, specify the format in the **Model Date Format** field. For more format options, see [Day.js](#).

The Date element's **Model Date Format** must match the Omnistudio Data Mapper's date format.

Mismatched date formats can result in a blank Date element.

5. Enter a **Default Value**. See [Set a Default Value for an Omniscript Element](#).



Note The format entered must be the same as in the specified in the date display format. field.

For example, to set the minimum date to January 1, 1950, if the date display format is `MM-DD-YYYY`, you must enter `01-01-1950`. The date picker lets the user choose a date that is +/- 100

years from the default date.

Omniscripts Time Element

Enable users to select from a list of times by adding the Time element to your Omniscript. Format the display and set the time intervals.

1. In **Minimum Time**, enter the earliest time the user can select, such as `12:00 am`.

 **Note** The format entered must be the same as in the **Time Display Format** field. For example, to set the minimum time to midnight, if the **Time Display Format** is `hh:mm a`, you must enter `12:00 am`.

2. In **Maximum Time**, enter the latest time the user can select, such as `08:00 pm`.

 **Note** The format entered must be the same as in the **Time Display Format** field. For example, to set the maximum time to 8pm, if the **Time Display Format** is `hh:mm a`, you must enter `08:00 pm`.

3. In **Time Display Format**, enter the format to display the time, such as `hh:mm a`, which is the default.

 **Note** Enter `h` or `hh` for hour, `m` or `mm` for minute, `s` or `ss` for seconds, and `a` for am/pm. A single letter, such as `h` or `m`, has no leading zero. Remove the `a` to display 24 hour time. For example, `hh:mm a` displays `03:30 pm`, `h:mm a` displays `3:30 pm`, and `hh:mm:ss` displays `15:30 01`. For more format options, see [Day.js](#).

4. In **Time Interval**, enter the intervals within an hour the user can select from the time dropdown. For example, enter `15` for 15 minute periods within the hour such as 12:00, 12:15, 12:30, 12:45, and so on. The default interval is `30`.

5. In the **Advanced Options** section, update values for these properties:

- **Data Type**: Specifies how to display the time in the Data JSON.
 - If set to **String**, the specified format is applied. **String** is the default data type.
 - If set to **Date**, the time is encoded as an ISO 8601 time.
- **Format**: Specifies how the time is formatted in the Data JSON when **Data Type** is set to **String**. The default format is `HH:mm:ss.sss'Z'`, and '`Z`' refers to the GMT timezone. If you choose the data type as **Date**, after you save the time in your Omniscript, and when you retrieve the time later, it shows your current timezone instead of GMT. For example, if you add 3:00 PM GMT in a time field, and you're in the IST timezone, when you retrieve the time in Omniscript, it shows the time as 8:30 PM IST. Therefore, to retrieve the time with the correct timezone, we recommend that you convert the retrieved time to GMT. If you choose the data type as **String**, you can enter the date in the `HH:mm:ss.sss'Z'` format, and you don't have to convert when you retrieve the time.
In the designer for a managed package, the field is **Model Time Format**. For more format options, see [Day.js](#).

6. For additional properties, see [Common Omniscript Element Properties](#).

Set Date and Time Ranges

The Date, Date/Time, and Time elements enable users to input dates and times into an Omniscript form. Each element has different properties that enable you to present a selectable range of dates or times to the User.

Date Range

To create a selectable Date range for the Date or Date/Time element, configure these properties:

- Minimum Date: Sets the earliest selectable date in the range. In the designer for a managed package,
- Maximum Date: Sets the latest selectable date in the range.

The Minimum Date and Maximum Date fields accept both dynamic and static values.

Examples of acceptable dynamic value syntax:

- *today + 1 day*
- *today - 5 days*
- *today + 2 months*
- *today - 1 month*

Examples of acceptable static value syntax:

- *2018/10/21*
- *10-21-2018*

Time Range

To create a selectable Time range for the Time element, configure the following properties:



Note Omniscripts do not allow the DateTime format to be included in the Minimum Time or Maximum Time fields.

- Minimum Time: Sets the earliest selectable time in the range.
- Maximum Time: Sets the latest selectable time in the range.

The Minimum Time and Maximum Time fields accept static values that must include four digits formatted as HH:mm in the 12-hour clock or 24-hour clock syntax.

Examples of acceptable 12-hour clock values include:

- *01:00 AM*
- *01:00 PM*
- *T07:30:00.000Z*

Examples of acceptable 24-hour clock values include:

- `01:00`
- `13:00`
- `T19:30:00.000Z`

Omniscripts Date/Time Element

Enable users to select a date from a date picker and a time from a list by adding the Date/Time element to your Omniscript. Set the minimum and maximum dates, and the date and time formats.

1. From the elements panel, drag a **Text** element to the canvas.
2. In the Properties panel, give the element a name in the **Name** field.
3. In **Field Label**, enter a label visible to the user.
4. Enter the earliest date the user can select from the date picker, such as `01-01-1950`.

 **Note** The date format entered must be the same as specified in the date display format. For example, to set the minimum date to January 1, 1950, if the date format is `MM-DD-YYYY`, enter `01-01-1950`.

5. Enter the latest date the user can select from the date picker, such as `12-31-2019`.

 **Note** The date format entered must be the same as specified in the date display format. For example, to set the minimum date to January 1, 1950, if the date format is `MM-DD-YYYY`, enter `01-01-1950`.

6. In the date format, enter the format to display the date. For example, if the format is `MM-DD-YYYY`, and the user selects `January 3, 2020` from the date picker, the date displays as `01-03-2020`. For more format options, see [Day.js](#).

7. Enter the format to display the time, such as `hh:mm a`, which is the default.

 **Note** Enter `h` or `hh` for hour, `m` or `mm` for minutes, `s` or `ss` for seconds, and `a` for am/pm. A single letter, such as `h` or `m`, has no leading zero. Remove the `a` to display 24-hour time. For example, `hh:mm a` displays `03:30 pm`, `h:mm a` displays `3:30 pm`, and `hh:mm:ss` displays `15:30 01`. For more format options, see [Day.js](#).

8. Enter the time interval within an hour the user can select from the time dropdown. For example, enter `15` for 15-minute periods within the hour such as `12:00`, `12:15`, `12:30`, `12:45`, and so on. The default interval is `30`.

9. In the **Advanced Options** section, select a time zone:

- Select **Local Timezone** to use the time zone of the browser.
- Select **User Timezone** to use the time zone set in the logged-in user's profile.

Omniscripts Disclosure Element

To allow users to agree to a disclosure statement, add a Disclosure element to your Omniscript. Enter the copy in a rich text editor.

If needed, select **Required** to make selecting the checkbox next to the disclosure statement required.

Click the **Text** field to open the rich text editor and enter the text for your disclosure statement.

Enter an Email Address in Omniscripts

Enable users to enter an email address by adding the Email element to your Omniscript. Update the default allowable format with Validation Options.

If needed, update data validation options:

- In **Pattern**, enter a pattern to override the default allowable email format. Use simple pattern matching. Regular expressions (regex) are accepted. See [Regular Expressions](#).
- In **Pattern Error Text**, enter the message displayed if the input doesn't match the expression in **Pattern**.

For additional properties, see [Common Omniscript Element Properties](#).

Query Salesforce Data from an Omniscript with the Lookup Element

Run a query using text input to retrieve Salesforce data using the Lookup element. The retrieved data is returned in value-label pairs and becomes available for selection in a dropdown list.

The Lookup element calls the Omnistudio Data Mapper Extract service to query sObject tables based on the parameters configured in the Properties section. To avoid performance issues, keeping the number of value results under 150 is recommended. To use a custom Apex class to perform the lookup, select the Custom type. Lookup elements display values from Salesforce based on a standard SOQL query built using the parameters of the Lookup elements.

SObject Type

To query for SObject data:

1. Set the Data Source Type to **SObject**.
2. Under **Input Parameters**, configure these fields:
 - **Data Source**: Select an Element Name from the dropdown to send that element's JSON value in the lookup request. For example, suppose the Data Source is a Text input element named CompanyName that asks the user for the name of an Account.
 - **Filter Value**: Enter a name for the JSON value to use in the lookup query. You can use any value as long as it matches a Filter Value in a Lookup Query Configuration.
3. Configure the Lookup Query Configuration:
 - a. Click **Add New Lookup Query Configuration**.
 - b. Enter a number to set priority for the Lookup order if there are multiple queries.
 - c. Select a Lookup SObject.
For example, Account.

- d. Select a Lookup SObject Field field to run the query on.
For example, Name.
 - e. Click Filter Operator to select an operator.
For example, to return an exact match, use the = operator, or to select matches similar to the filter value, select the **LIKE** operator.
 - f. Enter the name of a Filter Value used in the Input Parameters section.
For example, Company Name.
 - g. Set the return path for the filtered values by adding the element name of the Lookup element in the **JSON path**.
For example, Company.
4. In the Populate Lookup Element with Query Results section, set a label and a value for the Lookup results by configuring these fields:
- Label: Set the label for a Lookup Element's dropdown list to a JSON Value returned by the query. Because the query returns the values in the Lookup Element's JSON Path, you must set the full path. For example, to display an Account Name as a label in a Lookup Element named Company, the full path is *Company:Name*.
 - Value: Set the Value to an Object field that populates the Lookup Element's JSON Node using the full path. When a user selects a label, the value populates the Lookup element's JSON Node. For example, to use an Account Id for a JSON value when a user selects an Account Name from the Lookup element, the full path is *Company:Id*.
5. Preview the Omniscript to ensure the query is working and the value is being set correctly.

The query is structured in this format:

```
SELECT Lookup Object WHERE Lookup Object.Field Name = Filter Value
```

Custom Type

Use the Custom type to call a custom Apex class.

The Apex class implements Callable or *namespace.VlocityOpenInterface*, where *namespace* is the namespace of the package. The source is *namespace.ClassName.MethodName*.

For more details on query configurations, see [Working with Lookup Query Configurations](#).

Picklist Filter by Record Type

Enable the filtering of picklist options by record type.

You can look up picklist fields and record types for an object in Setup in the Object Manager.

1. In the **Picklist Object and Field** property, enter the picklist in this format:
ObjectAPIName.*FieldAPIName*.
2. Filter the picklist by **Record Type**.

Working with Lookup Query Configurations

When using the Lookup Element in Omniscript, the filtering of results is performed by the Lookup Query Configuration properties section. There are three main types of queries that can be built using this the Lookup Query Configuration properties section: AND join, OR join, and Deep query.

AND Join

To create AND Join and filter results on two or more fields set the sequence field value to the same value as the fields you want to join together in the query.

Example AND Join

The screenshot shows the configuration interface for a Lookup Query. The top section, 'LOOKUP QUERY CONFIGURATION', contains two rows of filters. Both rows have a 'Lookup Order' of 1. The first row filters 'Lead' by 'Country' ('USA') and the second row filters 'Lead' by 'City' ('San Francisco'). The bottom section, 'POPULATE THE LOOKUP ELEMENT WITH QUERY RESULTS', maps these filters to JSON paths: 'Lead.Id' is mapped to 'Name' and 'Lead.Company' is mapped to 'Value'. A red box highlights the 'Lookup Order' column in both rows of the configuration table.

In this example, we search for Leads that have Country = "USA" AND City = "San Francisco". This is done by selecting the same Lookup order and populating the JSON Path with the same value.

Note The JSON path can be any text but a best practice is to name the JSON Path the object being referenced.

Deep Query

To create a Deep query, the Lookup Order is put in sequential order. Omniscript builds smaller query sets at each order set.

Example Deep Query

LOOKUP QUERY CONFIGURATION

Lookup Order	Lookup Object And Field	Filter Value	JSON Path
1	Lead Country	= "USA"	Lead
1	Lead City	= "San Francisco"	Lead
2	Lead Industry	= "Government"	Lead:Government

POPULATE THE LOOKUP ELEMENT WITH QUERY RESULTS

JSON Path Field Name	Name	Value
Lead:Government:Id	Name	Value
Lead:Government:Company	Name	Value

In the example Deep query we are search for Leads that have Country = "USA" AND City = "San Francisco".

The from that data set we also only display Leads that have Industry = "Government". Since we are performing a Deep Query the JSON Path Field Name needs to be updated to populate Accordingly.

- Note** The JSON path in the Field Population property section must match the text used in the JSON Path. A best practice is to separate the JSON Path with a colon to indicate the nested JSON Path.

OR Join

OR Joins are not currently supported from within the Omniscript editor so a workaround must be used to accomplish this functionality. To perform an OR Join, a custom formula field on the object can be implemented to accomplish the OR portion of the query such that the formula will return either a TRUE value if the OR query is met and FALSE if not.

Example with an OR Join

LOOKUP QUERY CONFIGURATION

Lookup Order	Lookup Object And Field	Filter Value	JSON Path
1	Lead Country	= "USA"	Lead
1	Lead City	= "San Francisco"	Lead
2	Lead Vlocity_Industry__c	= "TRUE"	Lead:Industry

POPULATE THE LOOKUP ELEMENT WITH QUERY RESULTS

JSON Path Field Name	Name	Value
Lead:Industry:Id	Name	Value
Lead:Industry:Company	Name	Value

The Vlocity_Industry__c formula field example uses this syntax:

```
IF (
OR (
ISPICKVAL(Industry, "Government"),
ISPICKVAL(Industry, "Insurance"),
ISPICKVAL(Industry, "Communications")

),
"TRUE",
"FALSE")
```

This effectively allows for an OR join even though the Omniscript element doesn't support this functionality directly.

 **Note** In all instances, Filter Values must be text and wrapped in double quotations.

Omniscripts Number Element

Enable users to enter a number by adding the Number element to your Omniscript. Limit what the user can enter by setting a mask.

In **Mask**, set the allowable format the user can enter.

The # symbol and the integers 1-9 represent a digit in the field. You can separate the integers by using commas and a single decimal. The decimal is only applied if a user enters it into the field. For example, the mask ###,###.## transforms an input of 123456.78 into 123,456.78.

Enter a + or - symbol to return a positive or negative integer. A + or - symbol in the Mask property is ignored.

See [JavaScript Number Formatter](#).

For **Validation Options** and other additional properties, see [Common Omniscript Element Properties](#).

Omniscripts Password Element

Enable users to enter a password by adding the Password element to your Omniscript. Set the minimum and maximum lengths of the password.

Enter the minimum number of characters the user must enter for a valid password. Set the maximum number of characters the user can enter for a valid password.

For **Validation Options** and other additional properties, see [Common Omniscript Element Properties](#).

Omniscripts Multi-Select Element

Enable users to select from multiple items by adding the Multi-select element to your Omniscript. Display options vertically, horizontally, or as an image. Read-only Multi-Select images show in grayscale, but you can use a custom CSS to them in color.

1. Select how to display the radio buttons.
 - **Horizontal:** (Default) Display radio buttons next to each other.
 - **Vertical:** Display radio buttons stacked.
 - **Image:** Display each radio button as an image.
Read-only Multi-Select element images render in grayscale. To restore color for these images, see [Restore Color for Read-Only Multi-Select Element Images](#).
2. If you select **Image** as the display format, the following are additional properties you can configure:
 - **Width:** Sets the image width.
 - **Height:** Sets the image height.
 - **Image Count In Row:** Sets how many images per row to display.
 - **Enable Caption:** Displays a caption below the image.
3. In **Option Source**, select where the list of options come from. Select from one of the following:
 - **Manual:** (Default) Manually enter value/label pairs. Available when the display format isn't **Image**.
 - **Custom:** Enter the Apex class and method that returns the options. Use the format `ClassName.method`. See [Populating Picklist Values in Omniscript Inputs from Apex](#).
 - **SObject:** Retrieves the picklist values from the Salesforce object and field. Use the format `ObjectName.FieldName`.
 - **Image:** When Image is selected as Display Mode, manually enter value/label pairs and upload images to display. If no image is uploaded, the label displays in the image box.
4. If you select **Manual** or **Image** as an **Option Source**, for each option follow these steps:
 - a. Click **Add Option**.
 - b. Enter the **Value** and visible **Label**.
 - c. If your option is an image, select an image available from the **Choose existing image** dropdown. Or upload an image from your computer from the **Or upload new image** section.

 **Note** If you're using the Omnistudio standard runtime and standard designer without the Omnistudio managed package installed, you may encounter an error when uploading images. To resolve this issue, install the latest version of Omnistudio managed package. For installation instructions, see [Install or Upgrade the Omnistudio Managed Package](#).
 - d. Select **Use as Default Value** to select the option by default.
5. If you select **Custom** as an **Option Source**, in the **Source**, enter the name of a method to call on a class in the format `class.method`.
6. If you select **SObject** as an **Option Source**, in the **Source**, enter the name of a field on an object in the format `sobject.field`.
7. To display options based on the selection of another value, configure **Controlling Field Type** by

following these steps:

- a. To define the source of the controlling field:
 - To retrieve picklist options from an Apex class, select **Custom**.
 - To retrieve dependent picklist values from a Salesforce object, select **SObject**.
 - b. In **Controlling Field Source**, enter an Apex class.
 - c. In **Controlling Field Element**, enter an Omniscript element name.
8. For additional properties, see [Common Omniscript Element Properties](#).

Restore Color for Read-Only Multi-Select Element Images

Read-only Multi-Select element images render in grayscale. To restore color for these images, create a .css file, save it as a static resource, then reference it in the Custom Lightning Stylesheet File Name property in Omniscript Setup.

1. Create a `ReadOnlyRestoreColor.css` file with this code:

```
.omni-read-only [data-omni-input] .slds-img-item_select-container,  
.omni-read-only [data-omni-input] input[type=checkbox],  
.omni-read-only [data-omni-input] input[type=radio],  
.omni-read-only [data-omni-input] input[type=range] {  
    filter: brightness(1.25) grayscale(0) !important;  
}
```

2. From Setup, open Static Resources.
3. Click **New**.
4. In Name, enter `ReadOnlyRestoreColor`.
5. Click **Browse**, then select the `ReadOnlyRestoreColor.css` file.
6. Save your changes.
7. From the App Launcher, find and select **Omnistudio**, then click **Omniscripts**.
8. In the list of Omniscripts, Find and expand your Omniscript, then click the version.
The Omniscript opens in the Omniscript Designer.
9. In the Omniscript Designer, click **Setup**.
10. Expand **Styling Options**.
11. In Custom Lightning Stylesheet File Name, enter `ReadOnlyRestoreColor`.

Omniscripts Select Element

Enable users to select from a dropdown by adding a Select element to your Omniscript. Users can enter text to filter the options. Omnistudio pulls the options from an Apex class and method or from a Salesforce object.

1. In **Option Source**, select the source of the options:
 - **Manual:** (Default) Manually enter value and label pairs.
 - **Custom:** Enter the Apex class and method that returns the options. Use the format `ClassName.method`. See [Populating Picklist Values in Omniscript Inputs from Apex](#).
 - **SObject:** Retrieves the picklist values from the Salesforce object and field. Use the format `ObjectAPIName.FieldAPIName`.
2. If you select a manual option source, for each option follow these steps:
 - a. Click **Add Option**.
 - b. Enter the **Value** and visible **Label**.
 - c. (Optional) Check **Use as Default Value** to make the option selected by default.
 - d. (Optional) Check **Auto Advance** to advance the user to the next step when they click this option.
3. If you select **Custom** as an **Option Source**, in the **Source**, enter the name of a method to call on a class in the format `class.method`.
4. If you select **SObject** as an **Option Source**, in the **Source**, enter the name of a field on an object in the format `sObject.field`.
5. (Optional) To display options based on the selection of another value, configure **Controlling Field Type** by following these steps:
 - a. To define the source of the controlling field by retrieving picklist options from an Apex class, select **Custom**.
 - b. To define the source of the controlling field by retrieving dependent picklist values from a Salesforce object, select **SObject**.
 - c. In **Controlling Field Source**, enter an Apex class.

The Controlling Field Property on a Select Element can display Select Field Values based on the selection of another value from a Controlling Select field. Both the controlling and dependent fields must pull their field values from a field in Salesforce using the sObject property and those fields must have a Field Dependency setup between one another via Salesforce Field Dependency feature.
 - d. In **Controlling Field Element**, enter an Omniscript element name.
6. For additional properties, see [Common Omniscript Element Properties](#).
7. Activate the Omniscript for the Select elements to function properly in the Preview mode.

Omniscritps Radio Element

Enable users to select one from multiple items by adding the Radio element to your Omniscript. Display options vertically, horizontally, or as an image.

1. Select the display format for the radio buttons. These options are available:
 - **Horizontal:** (Default) Display radio buttons next to each other.
 - **Vertical:** Display radio buttons stacked.
 - **Image:** Display each radio button as an image.
2. If you select **Image** as the display format, configure these additional properties:
 - **Width:** Sets the image width.
 - **Height:** Sets the image height.
 - **Image Count In Row:** Sets how many images per row to display.

- **Enable Caption:** Displays a caption below the image.
3. In **Option Source**, select where the list of options come from. Select from one of the following:
- **Manual:** (Default) Manually enter value/label pairs. Available when **Display Mode** is not **Image**.
 - **Custom:** Enter the Apex class and method that returns the options. Use the format `ClassName.method`. See [Populating Picklist Values in Omniscript Inputs from Apex](#).
 - **SObject:** Retrieves the picklist values from the Salesforce object and field. Use the format `ObjectName.FieldName`.
 - **Image:** When Image is selected as Display Mode, manually enter value/label pairs and upload images to display. If no image is uploaded, the label displays in the image box.
4. If you select **Manual** or **Image** as an **Option Source**, for each option follow these steps:
- a. Click **Add Option**.
 - b. Enter the **Value** and **visible Label**.
 - c. If your option is an image, select an image available from the **Choose existing image** dropdown. Or upload an image from your computer from the **Or upload new image** section.
-  **Note** If you're using the Omnistudio standard runtime and standard designer without the Omnistudio managed package installed, you may encounter an error when uploading images. To resolve this issue, install the latest version of Omnistudio managed package. For installation instructions, see [Install or Upgrade the Omnistudio Managed Package](#).
- d. If needed, make the option selected by default.
 - e. If needed, advance the user to the next step when they click this option.
5. If you select **Custom** as an **Option Source**, in the **Source**, enter the name of a method to call on a class in the format `class.method`.
6. If you select **SObject** as an **Option Source**, in the **Source**, enter the name of a field on an object in the format `SObject.field`.
7. To display options based on the selection of another value, configure **Controlling Field Type** by following these steps:
- a. To define the source of the controlling field:
 - To retrieve picklist options from an Apex class, select **Custom**.
 - To retrieve dependent picklist values from a Salesforce object, select **SObject**.
 - b. In **Controlling Field Source**, enter an Apex class.
 - c. In **Controlling Field Element**, enter an Omniscript element name.
8. For additional properties, see [Common Omniscript Element Properties](#).

Add Options for Selects, Multi-Selects, and Radio Buttons

To add options for Select, Multi-Select, and Radio Button elements, go to the **Options** section of the element properties and click **+ Add New Option**. If no values are defined for a Select element, an Undefined Value is returned.

Each option has a Value and Label pair. The Value is the language-independent data that gets passed in and out of the Omniscript remotely, through an Omnistudio Data Mapper, a Remote action, or an HTTP (REST) action. It's also referenced when setting up a Conditional View.

The Label is displayed in the Omniscript UI. The Value and Label can be identical or different.

For example, an ethernetlink element has four options:

Option	Value
No Link to an Ethernet Device	EN
No Link to the IDSL Network	NW
Integrated Services Router Crashes	IS
Hardware Crashes	HW

The value of one of these options is `EN`, and its label is `No Link to an Ethernet Device`. If an end user clicks **No Link To An Ethernet Device**, the data returned is `EN`.

For Multi-Select elements, if the user selects multiple options, a semicolon-delimited list of data is returned. For example, selecting **No Link to an Ethernet Device** and **Hardware Crashes** returns `EN;HW`.

-  **Note** If the values change frequently, enable the **Fetch Picklist Values at Script Load** to ensure that the Omniscript elements contain the most up-to-date set of values. Salesforce sources include picklists and custom classes.

Using Salesforce Picklists with Omniscript Inputs

You can dynamically populate a Select, Multi-Select, or Radio element in an Omniscript with values from a picklist in your Salesforce org or with a custom Apex class.

Populating Picklist Values in Omniscript Inputs from Apex

You can populate Select, Multi-select, and Radio elements from a Callable Apex class. The values for the element are obtained during the Omniscript's activation.

Populating Dependent Picklist Values in Omniscript Inputs with Apex

You can populate Select, Multi-select, and Radio elements from a Callable implementation. The values for the element are obtained during the Omniscript's activation.

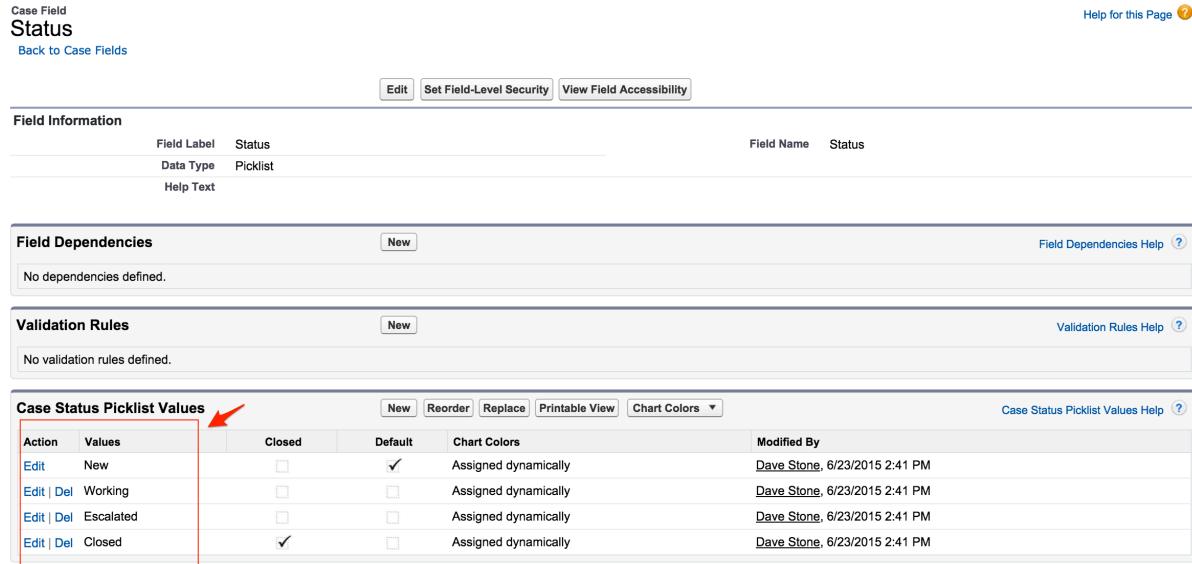
Use Images for Buttons in Omniscript

Use images as buttons for both radio and multi-select options in Omniscript.

Using Salesforce Picklists with Omniscript Inputs

You can dynamically populate a Select, Multi-Select, or Radio element in an Omniscript with values from a picklist in your Salesforce org or with a custom Apex class.

For example, an Omniscript to update cases requires the Status element to dynamically populate. These status values are from the **Status** field on the Case object.

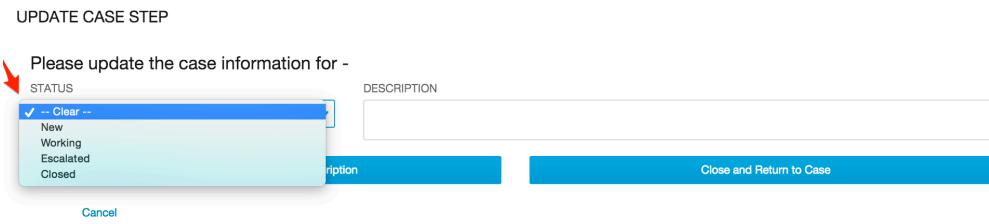


The screenshot shows the 'Status' field configuration for the Case object. It includes sections for Field Information, Field Dependencies, Validation Rules, and Case Status Picklist Values. A red arrow points to the 'Case Status Picklist Values' section, which lists four entries: New, Working, Escalated, and Closed. The 'New' entry is selected (indicated by a checked checkbox in the 'Default' column). The 'Closed' entry has a checked checkbox in the 'Default' column.

Action	Values	Closed	Default	Chart Colors	Modified By
Edit	New	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Assigned dynamically	Dave Stone, 6/23/2015 2:41 PM
Edit Del	Working	<input type="checkbox"/>	<input type="checkbox"/>	Assigned dynamically	Dave Stone, 6/23/2015 2:41 PM
Edit Del	Escalated	<input type="checkbox"/>	<input type="checkbox"/>	Assigned dynamically	Dave Stone, 6/23/2015 2:41 PM
Edit Del	Closed	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Assigned dynamically	Dave Stone, 6/23/2015 2:41 PM

To do this, select **SObject** as the **Option Source** type and type `Case.Status`:

Now the form retrieves and dynamically populates the list with values from Case.Status:



The screenshot shows an 'UPDATE CASE STEP' form. A red arrow points to the 'STATUS' picklist field, which contains the values: 'New', 'Working', 'Escalated', and 'Closed'. The 'New' option is selected. The form also includes fields for 'DESCRIPTION' and a 'Close and Return to Case' button.

-  **Note** If the object or field name are custom, they must be preceded with the NameSpace of the package—for example, `vlocity_cmt__Party__c.vlocity_cmt__Location__c` or `Case.vlocity_cmt__Amount__c`.

You can also use a custom class to populate lists in Omniscripts. See the sample code for instructions on implementation.

If the values of the list are [dependent on another field](#), add the information for that field in the Controlling Field section. Follow the instructions above to define the controlling field and also enter the element name for the field on the form.

Populating Picklist Values in Omniscript Inputs from Apex

You can populate Select, Multi-select, and Radio elements from a Callable Apex class. The values for the element are obtained during the Omniscript's activation.

To populate a picklist from an Apex class, make sure to enable **Get picklist values when the Omniscript loads** for the Omniscript from Setup. For the designer on a managed package, enable **Fetch Picklist Values at Script Load**.

To populate the options of an element, a `List<Map<String, String>>` is returned from the Apex class. The Map Options are **Name**, the Language Independent Option, and **Value**, the UI Display value.

To download a DataPack of a simple Omniscript that calls the following example Apex class, click [here](#).

Sample Apex Code for Populating a Picklist:

```
global with sharing class SelectOptionsCallable implements Callable
{
    global Object call(String action, Map<String, Object> args)
    {
        switch on action {
            when 'SelectOptions' {
                return SelectOptions((Map<String, Object>)args.get('input'), (Map<String, Object>)args.get('output'));
            }
        }
        return null;
    }

    private List<Map<String, Object>> SelectOptions(Map<String, Object> inputMap, Map<String, Object> outMap)
    {
        List<Map<String, Object>> optionList = new List<Map<String, Object>>();
        for (Account acc : [ Select Id, Name from Account limit 8])
        {
            Map<String, Object> option = new Map<String, Object>();
            option.put('name', acc.Id);
            option.put('value', acc.Name);
            optionList.add(option);
        }

        outMap.put('options', optionList);
    }
}
```

```

    }
}

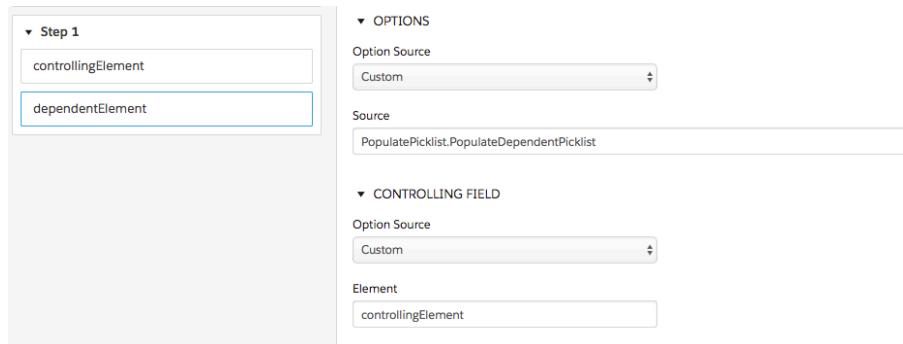
```

For information on populating dependent picklists, see [Populating Dependent Picklist Values in Omniscript Inputs with Apex](#).

Populating Dependent Picklist Values in Omniscript Inputs with Apex

You can populate Select, Multi-select, and Radio elements from a Callable implementation. The values for the element are obtained during the Omniscript's activation.

When populating a Dependent Picklist through Apex, the returned Picklist Options are a **Map<String, List<Map<String, String>>**. The keys to the initial Map are the Language Independent ('name') options from the Controlling field. The List<Map> Options are 'Name' the Language Independent Option and 'Value' the UI Display value. For information on populating a picklist with no dependencies, see [Populating Picklist Values in Omniscript Inputs from Apex](#).



The order of values from the Omniscript is the order of the output picklist.

- (i) **Note** Omniscript does not check to see if the dependent picklist value is valid when passed. Invalid values are discarded when the controlling picklist is changed.

Sample Apex Code:

```

global class PicklistPopulationExample implements Callable {
    public Object call(String action, Map<String, Object> args) {
        Map<String, Object> input = (Map<String, Object>)args.get('input');
        Map<String, Object> output = (Map<String, Object>)args.get('output');
        Map<String, Object> options = (Map<String, Object>)args.get('options');
        return invokeMethod(action, input, output, options);
    }
    public Boolean invokeMethod(String methodName, Map < String, Object > input, Map < String, Object > outMap, Map < String, Object > options) {

```

```
if (methodName.equals('PopulateDependentPicklist')) {
    PopulateDependentPicklist(input, outMap, options);
}
else if (methodName.equals('PopulatePicklist')) {
    PopulatePicklist(input, outMap, options);
}
return true;
}

// add the map under options to populate the picklist
public void PopulatePicklist(Map<String, Object> input, Map<String, Objec
t> outMap, Map<String, Object> options)
{
    // hard coded list, these options could be results from a remote call/
3rd party api
    List<Map<String, String>> picklist = new List<Map<String, String>>();

    Map<String, String> picklistOption = new Map<String, String>();
    picklistOption.put('name', 'Licensing & Permitting');
    picklistOption.put('value','Licensing & Permitting');
    picklist.add(picklistOption);

    picklistOption = new Map<String, String>();
    picklistOption.put('name', 'Contract');
    picklistOption.put('value','Contract');
    picklist.add(picklistOption);

    outMap.put('options', picklist);
}

// add the map of all possible picklists that will be chosen given the "co
ntrollingElement"
// note that the "controllingElement" is an element name that gets passed
in from omniscrypt
public void PopulateDependentPicklist(Map<String, Object> input, Map<Strin
g, Object> outMap, Map<String, Object> options)
{
    // Map of List where the Key is the Potential Values in the Other Pick
list
    Map<String, List<Map<String, String>>> dependency = new Map<String, Li
st<Map<String, String>>>();

    List<String> controlValList = new List<String>();
    // note that these are listed (above)
```

```
controlValList.add('Licensing & Permitting');
controlValList.add('Contract');

// generating arbitrary entries for each of the above keys
for(Integer i=0; i<2; i++)
{
    List<Map<String, String>> optionList = new List<Map<String, String>>();

    for(Integer j=0; j<2; j++) {
        Map<String, String> tempMap = new Map<String, String>();
        tempMap.put('name', controlValList[i] + ' Child');
        tempMap.put('value', controlValList[i] + ' ChildV');

        optionList.add(tempMap);
    }
    dependency.put(controlValList[i], optionList);
}

outMap.put('dependency', dependency);
}

}
```

Use Images for Buttons in Omniscript

Use images as buttons for both radio and multi-select options in Omniscript.

 **Note** Omniscripts and Flexcards support all image file types by default. If set, the lightning-file-upload `accept` attribute filters file types in the user's file browser. If set, the **Don't allow HTML uploads as attachments or document records** security setting disallows .svg image files.
See [Lightning Web Component File Upload](#).

1. From the elements panel, drag the element for which you want to use an image to the canvas.
2. In the Properties panel, select **Image** from the Display Mode and Option Source dropdowns.
3. If needed, enter the properties of the image such as the width and height.
4. Click **Add Option**.
5. Complete the required fields, and select an existing image or upload a new one.
6. Save your changes.

Omniscripts Range Element

Enables users to select a number from a specified range by adding the Range element to your Omniscript. Specify the increment values and use static values to define the minimum and maximum values for the range.

In **Step**, enter an increment value of the input range. For example, enter **2** so the user can select 2, 4, 6, 8, and 10 from a range of 2 to 10, or 4.5, 6.5, 8.5, and 10.5 from a range of 4.5 to 10.5. Default Step is **1**.

To create a selectable range configure the following properties:

- **Minimum Value:** Sets the lowest selectable number in the range.
- **Maximum Value:** Sets the highest selectable number in the range.

 **Note** Both fields accept static values. Acceptable static values include whole, negative, and decimal numbers, such as **1**, **-5**, and **.5**.

For additional properties, see [Common Omniscript Element Properties](#).

Omniscripts Telephone Element

Enable users to enter a phone number by adding a Telephone element to your Omniscript. Limit the length and format of the number the user can enter.

In **Autocomplete Input**, update the allowable format by entering a custom pattern. In the designer for a managed package, enter the format in the **Mask** field.

The default is **(999) 999-9999**, which matches any US-based phone number. For more information about patterns, see **Pattern Mask** at <https://imask.js.org/guide.html#masked-pattern>.

- In **Minimum Length**, update the minimum number of characters the user must enter.
- In **Maximum length**, update the maximum number of characters the user can enter.

For **Validation Options** and other additional properties, see [Common Omniscript Element Properties](#).

Omniscripts Text Area Element

Enable users to enter multiple lines of text by adding a Text Area element to your Omniscript. Limit the number of characters the user can enter.

 **Note** When you enter texts in multiple lines in the text area, the output appears in a single line.

- In **Minimum Length**, enter the minimum number of characters the user must enter.
- In **Maximum Length**, enter the maximum number of characters the user can enter.

Use additional properties described in [Common Omniscript Element Properties](#).

See Also

- [Omniscript Element Reference](#)
- [Omniscripts Text Element](#)
- [Omniscript Input Elements](#)

Omniscripts Text Element

Enable users to enter a line of text by adding the Text element to your Omniscript. Limit the length of the text using minimum and maximum values and set allowable information using the mask property.

To set the type of information and its format, enter a **Mask**. Use **A** for any letter, **9** for any number, ***** for any character, and **[]** to enclose optional characters. See [Pattern Mask](#).

When you set the Mask property on text fields, Salesforce shows the default error message instead of custom error messages for those fields.

- In **Minimum Length**, set the minimum number of characters the user must enter, such as **1**.
- In **Maximum Length**, set the maximum number of characters the user must enter, such as **300**.

For **Validation Options** and other additional properties, see [Common Omniscript Element Properties](#).

Omniscripts URL Element

Enable users to enter a website address by adding the URL element to your Omniscript. For example, a user can enter `https://salesforce.com`, or `https://www.salesforce.com`, and so on.

In the **Error Text** field of the **Validation Options** section, enter text to override the default message displayed when the user does not enter the URL correctly.

For additional properties, see [Common Omniscript Element Properties](#).

Upload Files and Images in Omniscripts

To upload files in an Omniscript, add a File input element. To upload an image, add an Image input element. Details about uploads are added to the Files node of the Omniscript's Data JSON.

By default, files are uploaded to Salesforce Content Documents directly. To associate one or more Salesforce objects with the uploaded file, specify a comma-separated list of object Ids in the **Content Parent Id** field. In Salesforce, the uploaded files are listed in the sObject detail page's Content section.

To specify a Vlocity Open Interface to be run after the file or image is uploaded, set the **Remote Action** section's **Remote Class** and **Remote Method** fields. As input, the specified method receives a `Map<String, Object>` list containing the File data from the Data JSON. The options contain the `remoteOptions` and the `filesMap`, which contains the Ids of any Content documents that were created.

By default, File and Image elements accept a file size of up to 2GB.

Omniscripts and Flexcards support all image file types by default. If set, the lightning-file-upload `accept` attribute filters file types in the user's file browser. If set, the **Don't allow HTML uploads as attachments or document records** security setting disallows .svg image files. See [Lightning Web Component File Upload](#).

Add a File or Image to Omniscript

Add and configure a File or Image element to enable the upload of files and images. By default, the File and Image elements accept a file size of up to 2GB.

 **Note** The Image and File elements don't work in preview because they use SFDC components. When Image and File elements are required in a Step, the Preview cannot advance past the Step. Marking File and Image elements as Required only before activating the Omniscript is recommended. Omniscripts and Flexcards support all image file types by default. If set, the lightning-file-upload `accept` attribute filters file types in the user's file browser. If set, the **Don't allow HTML uploads as attachments or document records** security setting disallows .svg image files. See [Lightning Web Component File Upload](#).

1. From the elements panel, drag a **File** or **Image** element to the canvas.
2. Enter an a content parent object ID, or comma-separated list of object IDs, to attach the Content Document to a parent record. When left blank, the Content Document does not attach to a parent record.
3. When configuring the Image element, allow uploads of multiple images, if required.
4. Use one of the JSON node names from Image and File to apply a condition to another element:

```
"yourFileElementName": [  
    {  
        "data": "0693g000000T1ZFAA0",  
        "filename": "isolated-on-png.png",  
        "vId": "0683g000000T11iAAC",  
        "size": 8234  
    }  
]
```

Omnistudio

Build industry-specific UI components and applications on the Salesforce platform by creating Omnistudio Flexcards in a visual editor. Show Salesforce object information with clickable actions that adapt to the context and type of data, helping your team handle tasks more efficiently. Flexcards are built with the Lightning Component framework.

[Working with Flexcards](#)

Create and manage Flexcards in the designer in the Omnistudio standard runtime. The designer provides the standard components for building user interfaces within Salesforce. Create Flexcards in fewer steps. With the designer, Flexcards activation is instant.

[Open a Flexcard in a Canvas Instead of the Record Page Detail](#)

Omnistudio provides an intuitive canvas to easily set up Flexcards. If canvas view isn't available when you open a Flexcard, update the default Lightning record page for the Omni UI Card object. For example, the default record page is sometimes different after you migrate from the custom object data model to the standard object data model.

[Create a Flexcard](#)

Create a Flexcard by specifying a unique name and author. Optionally, save the Flexcard as a child card to embed it in other Flexcards. In the designer for a managed package, you can also configure its data sources and test parameters.

[Configure Flexcard Settings](#)

Specify your Flexcard's behavior with the setup options. For example, add or update a data source, create an event listener, or add session variables.

[Add Elements to a Flexcard](#)

Build your card by dragging elements into a state on the canvas. Add data fields, display, and input elements to your Flexcard. Show data generated dynamically from the Flexcard's data source, or add static information. Rearrange, clone, delete, and adjust the widths of your elements as needed. Style an element and configure its properties in the Flexcard designer.

[Set Up Actions on a Flexcard](#)

Configure actions that users can complete on Flexcards to improve the self-service experience, and save time on manual tasks. Add actions to supported elements or add an action element to a Flexcard, and then configure the action settings. For example, you want to have a button that opens a URL for additional information. To do so, add an action to an existing image or add an action element for which you select an image. You can also set up sequential actions and configure each action's trigger.

[Display Data on Flexcards Based on Conditions](#)

Determine whether to apply conditions to individual Flexcard elements, to the entire Flexcard, or to both. Use states to control what's available to users at the Flexcard level. For a more granular configuration, apply conditions to specific elements, fields, events, and more. For example, to show an alert icon for escalated cases, apply conditions to that icon. To show different Flexcards based on the payment status of an insurance account, create states for each status.

Style a Flexcard

Configure the look and feel of a Flexcard by styling individual elements, fields inside elements, and entire states. Configure backgrounds, text, and borders; adjust dimensions and element spacing; apply classes and inline styles; make your element responsive; and more.

Preview and Debug a Flexcard

Preview a Flexcard's appearance and test its functionality before you publish it to a Lightning page or to a page in an Experience Builder Aura site. Review how the Flexcard looks on multiple devices and test the functionality of interactive elements such as actions. Debug the Flexcard's data output, events, actions, and more.

Activate and Publish a Flexcard

After you confirm that the Flexcard is ready, activate it. After activation, add the Flexcard to a Lightning page or to a page in an Experience Builder site.

Export and Import Flexcards

Import or export Flexcards to use them in another org, such as when you create them in a sandbox or to use sample data packs that Salesforce provides. When exporting or importing data packs for Flexcards, if you encounter Apex limit errors such as heap or CPU limits, reduce the size of the data pack by unselecting dependencies during the export process. Additionally, break the data pack into multiple smaller data packs. As a best practice, we recommend including no more than 10 elements in each data pack.

Flexcards Context Variables

To provide context to data sources, actions, and other components, add global and local context variables to a Flexcard. All variables are case-sensitive.

Flexcards Omni Interaction Access Configuration

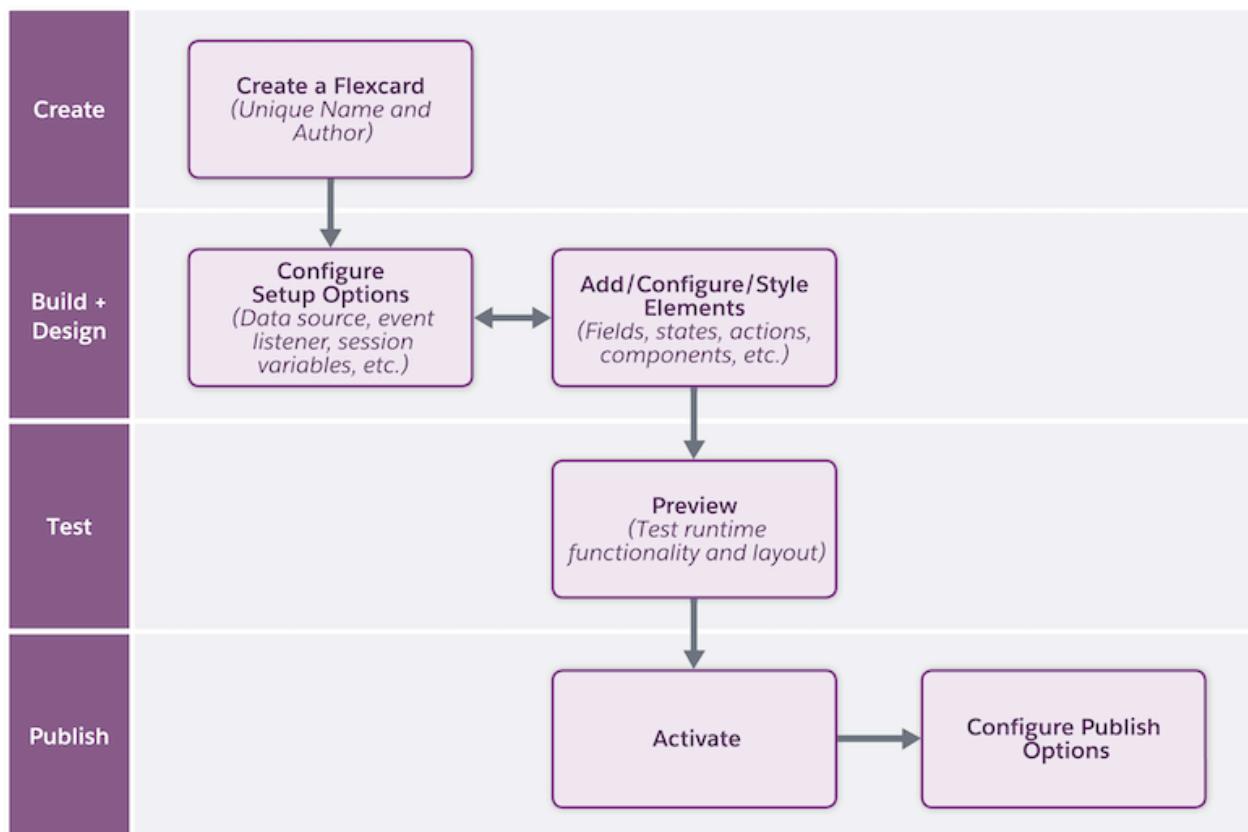
In Omnistudio, limit access to data sources and update cache settings for the Omni UI Card object (Flexcard) across your org. Enable and disable data source types for an org, profile, or user level. Disable caching on the Platform Cache for all Flexcards. Enable Flexcards to cache asynchronously without waiting for the full caching process to finish.

Flexcard Designer

Starting with Winter '25, create and manage Flexcards in the new designer included in Omnistudio standard runtime. The new designer provides the standard components for building user interfaces within Salesforce. It doesn't require any package installation and is available with the Salesforce platform after the permissions are enabled. These designers are available to all new customers of Omnistudio. The existing designer on the managed package remains unchanged.

Flexcard Workflow

Starting with Winter '25, use this workflow to create, configure, and publish Flexcards in the new designer included in Omnistudio standard runtime.

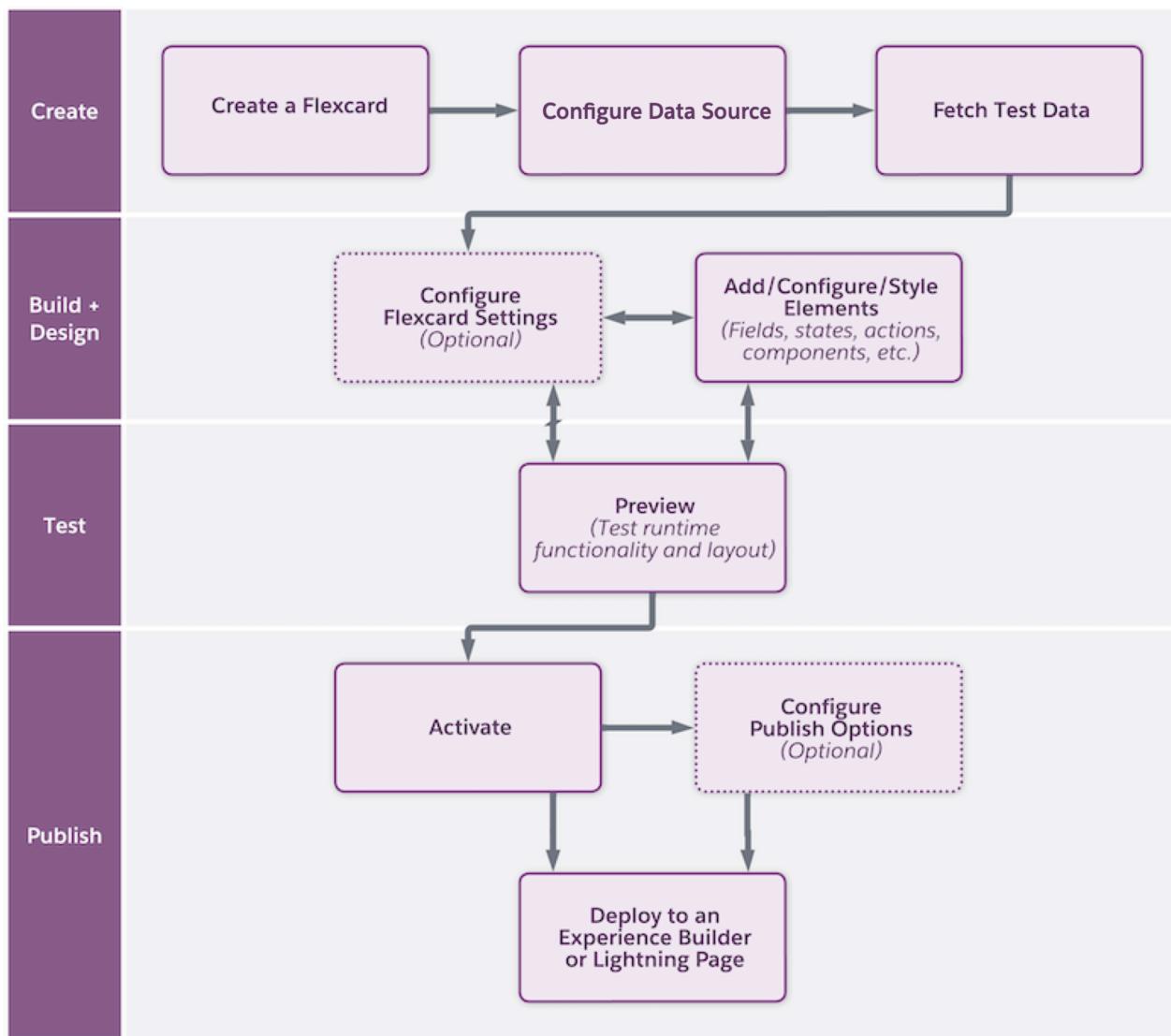


1. Create a Flexcard with a unique combination of name and author.
See [Create a Flexcard](#).
2. Configure the behavior, visual elements, and interactions that the Flexcard offers to your users. For example, configure settings, add fields for input data, navigation elements to open links, specify the elements' responsiveness, and more. If needed, modify the style elements such as background and size to provide a customized experience to your users.
See [Configure Flexcard Settings](#), [Add Elements to a Flexcard](#), [Set Up Actions on a Flexcard](#), [Set Up Conditions to a Flexcard Element](#), and [Style a Flexcard](#).
3. Preview the Flexcard, test its look and feel, and resolve issues as needed.
See [Preview and Debug a Flexcard](#).
4. After you confirm that the Flexcard is ready, activate it. Then, add the Flexcard as a standard component to a target page, such as an app page or community page.
See [Activate and Publish a Flexcard](#).

Flexcard Workflow in the Designer on a Managed Package

Use this workflow to create, configure, and publish Flexcards in the designer on a managed package.

To enable this designer, contact your account executive.

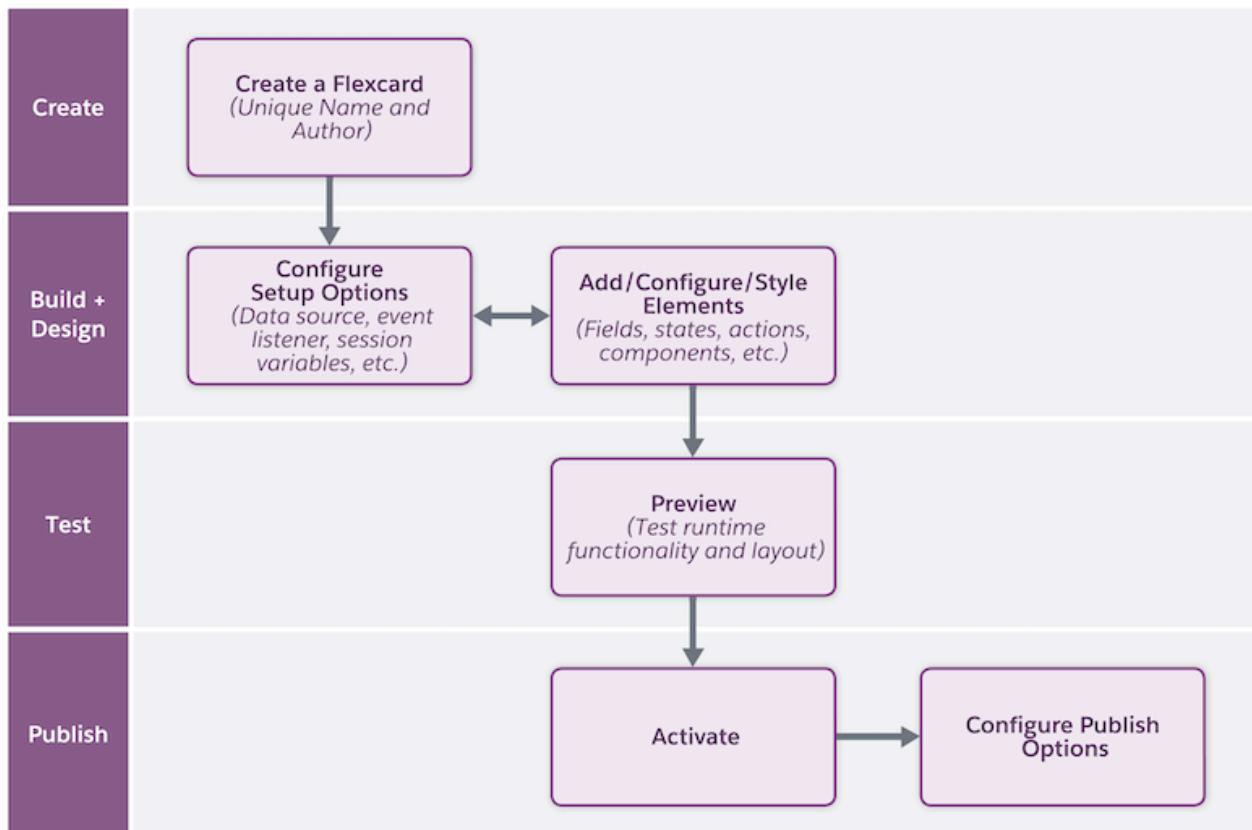


1. Create a Flexcard with a unique combination of name and author and, if needed, configuring the data source type and testing data retrieval.
See [Create a Flexcard](#) and [Configure Flexcard Settings](#).
2. Configure the visual elements and interactions that the Flexcard offers to your users. For example, add fields for input data, navigation elements to open links, specify the elements' responsiveness, and more. If needed, style elements such as for the background or the size to provide a customized experience for your users.
See [Configure Flexcard Settings](#), [Add Elements to a Flexcard](#), [Set Up Actions on a Flexcard](#), [Set Up Conditions to a Flexcard Element](#), and [Style a Flexcard](#).
3. Preview the Flexcard, test its look and feel, and resolve issues as needed.
See [Preview and Debug a Flexcard](#).
4. Publish and deploy the Flexcard. After you confirm that the Flexcard is ready, activate it. Then, publish that component to a Lightning page, an Experience Cloud page, or an external site, depending on your Omnistudio version.

See [Activate and Publish a Flexcard](#).

Working with Flexcards

Create and manage Flexcards in the designer in the Omnistudio standard runtime. The designer provides the standard components for building user interfaces within Salesforce. Create Flexcards in fewer steps. With the designer, Flexcards activation is instant.



1. Create a Flexcard with a unique combination of name and author.
See [Create a Flexcard](#).
2. Configure the behavior, visual elements, and interactions that the Flexcard offers to your users. For example, configure settings, add fields for input data, navigation elements to open links, specify the elements' responsiveness, and more. If needed, modify the style elements such as background and size to provide a customized experience to your users.
See [Configure Flexcard Settings](#), [Add Elements to a Flexcard](#), [Set Up Actions on a Flexcard](#), [Set Up Conditions to a Flexcard Element](#), and [Style a Flexcard](#).
3. Preview the Flexcard, test its look and feel, and resolve issues as needed.
See [Preview and Debug a Flexcard](#).
4. After you confirm that the Flexcard is ready, activate it. Then, add the Flexcard as a standard component to a target page, such as an app page or community page.
See [Activate and Publish a Flexcard](#).

Omnistudio provides an intuitive canvas to easily set up Flexcards. If canvas view isn't available when you open a Flexcard, update the default Lightning record page for the Omni UI Card object. For example, the default record page is sometimes different after you migrate from the custom object data model to the standard object data model.

1. From the object management settings for Omni UI Card, go to [Lightning Record Pages](#).
2. Make sure that Vlocity Card Designer is activated and set as default for the org and the Omnistudio app.
See [Activate Lightning Experience Record Pages](#).
3. If Vlocity Card Designer isn't listed, create a Lightning Record page with these values:
 - Label: Vlocity Card Designer
 - Object: Omni UI Card
 - Page Template: Header and Collapsible Right Sidebar
4. Drag these components to the canvas:
 - `cardDesignerHeader` : Upper section
 - `cardPreview` : Left section
 - `rightSidebar` : Right section
 - `cardDesignerCommon` : Right section below the `rightSidebar`
5. Save your changes.
6. Activate the record page and set it up as the org and Omnistudio app defaults.

When you open a Flexcard, Omnistudio opens it on the record page that you created, in canvas view.

Create a Flexcard

Create a Flexcard by specifying a unique name and author. Optionally, save the Flexcard as a child card to embed it in other Flexcards. In the designer for a managed package, you can also configure its data sources and test parameters.

Before You Begin

Make sure you don't use reserved words when you create a Flexcard. For more information, see [Omnistudio Naming Conventions](#).

To create a Flexcard, complete these tasks.

1. From the App Launcher, find and select **Flexcards**.
2. Click **New**.
3. Enter a name and author for the Flexcard.

Keep in mind that:

The combination of the name and author must be unique. For example, you can't import a Flexcard whose name and author match an existing Flexcard without overwriting it.

You can't change the name, the theme, or the author of a Flexcard unless you clone it, and change the name of the cloned Flexcard.

4. If needed, update the default title.

Salesforce shows the title in the Lightning App Builder or the Experience Builder when you add the Flexcard to a page.

5. If needed, save the card as a child card to embed the Flexcard in another Flexcard.
6. If needed, in the designer on a managed package, select the data source type and then configure the data source.



Note You can configure the data source later from the Setup panel.

7. If needed, in the designer on a managed package, configure **Test Parameters** to view test data when your data query has merge field variables such as {recordId}.

Clone a Flexcard or Create a New Version

Consider the available options for copying a Flexcard.

Clone a Flexcard or Create a New Version

Consider the available options for copying a Flexcard.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select the version to copy.
3. Clone or create a version of the Flexcard:
 - **New Version:** Update a Flexcard that you already published such as for testing purposes. The name and author remain the same, and only one version can be active at a time. When you activate a version, Omnistudio deactivates all other versions of the Flexcard.
 - **Clone:** Create a Flexcard with the same layout, settings, or both, as an existing Flexcard. The required information is the same as for a new Flexcard.
4. Set up the cloned Flexcard or the new version as needed.

Configure Flexcard Settings

Specify your Flexcard's behavior with the setup options. For example, add or update a data source, create an event listener, or add session variables.

Note After updating or configuring Flexcards, allow 15–20 minutes for updates to appear, even after clearing your cache. If you see any errors or if old versions load, contact Salesforce support to enable the **Template API** setting. This helps updates show up faster and more reliably.

1. From the App Launcher, find and select **Flexcards**.

2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard.
4. Go to Setup and update the relevant information.
 - To show or hide the Setup panel, click .
 - In the designer on a managed package, click the **Setup** tab.
5. If the Flexcard interacts with a Lightning Web Component Omniscript, turn on Omniscript support. For example, render a Flexcard inside an Omniscript with the Omniscript's Custom Lightning Web Component element. Or update an Omniscript with the Update OmniScript action element. See [Embed Flexcard in a Flexcard](#).
6. To translate custom labels into any supported language, turn on multi-language support. You can then add custom labels to these Flexcard elements by using the {Label} merge field.
 - Field: Label, output, placeholder
 - Text: In the rich text editor
 - Action: Label



Note Omnistudio updates custom labels and doesn't store them in the cache.

See [Create and Edit Custom Labels](#).



Note Omnistudio provides autotranslated system labels, such as standard error messages, in the standard runtime. These autotranslated labels appear in your Flexcards based on the user's locale.

7. To add a dynamic class automatically in the CSS for resizing the element and rendering the card again when the browser width changes, turn on the width listener.
8. To restrict access to the Flexcard, enter a comma-separated list of custom permissions.



Note If this field is left blank, all users can see the Flexcard. Assigning user permission only affects the Flexcard's visibility in the user interface at runtime and doesn't restrict data access. Data access is managed through field-level security. To ensure data security and maintain compliance with Salesforce encryption access controls, always check that a user has the **View Encrypted Data** permission before displaying or processing decrypted values of encrypted fields.

For example, if you have custom permissions named `Can_Edit_Policy` and `Can_View_Policy`, enter them as `Can_Edit_Policy,Can_View_Policy`.

9. To improve performance when you load the card in preview, disable the child Flexcard preview.
10. If needed, generate a card container for each data source's record instead of a single card that shows all the records.
 - a. Select **Repeat Records**.
 - b. Enter a render key that identifies the repeated card record to update in the Document Object Model (DOM).

A render key helps improve performance by regenerating only the updated card record.

If your data source is a data table, deselect Repeat Records to avoid having multiple cards showing the same information. Similarly, we recommend that you deselect the checkbox for charts or lists that retrieve data from multiple records. Also, deselect the checkbox for container elements such as Lightning web components or child Flexcards that use their own data sources.

If your Flexcard has multiple states, Omnistudio applies the state's condition. For example, if you select Repeat Records, but the available records don't meet the state's condition, Omnistudio hides those records.

11. If needed, apply a specific style sheet for the Flexcard.
12. If needed, add an event listener that tracks an event triggered from a component, and performs an action in response:
 - a. Add an event listener.
 - b. Set up the listener properties.
 - Pubsub: Event from another component, such as another Flexcard on the same Lightning page.
 - Custom event: Event from a child Flexcard or from an element on the card.
 - Record change: Change on a record that's on the Flexcard. Enter the record's ID as {recordId} or enter an ID value, and select the record type. If needed, also select which field values to copy onto the card when the record changes.

By default, in the Record Change event type you can select fields based on a Salesforce object, by using the Fields dropdown menu. However, if you turn on Advanced Mode, you can specify fields within any SObject and related objects in an array. For example, you can specify fields for the Account object in your array this way: `Contact.Id, Contact.Account.Id, Contact.Name`.

 **Note** If you change the record through an Experience Cloud page, the change only takes effect when it's edited within the same browser tab and not via backend processes.

For information about the pubsub channel name or custom event name, see [Pass Data to Flexcards by Using Events](#). Also see [Set Up Actions on a Flexcard](#) and [Set Up Conditions on a Flexcard Element](#).

- c. If you set up multiple actions and want to arrange them in a different sequence, expand the action properties sections and drag them in the desired order.
See [Set Up Actions on a Flexcard](#).
13. To create session variables that are globally accessible, make attributes available from a component, or store values from data sources, external systems, or hardcoded values.
 - a. Add a session variable.
 - b. Enter a key and a value.

You can then apply the session variable as a merge field to an element field. For example, enter {Session.var} to a REST data source endpoint URL.

Session variables are available for these elements and their fields:

- Action: Label
- Custom event action: Input Parameter > Value
- Field: Label, Placeholder, and Output
- Text: Inside the rich text editor

You can't use data source values as session variables.

14. To create a public property whose value you set in a Lightning or Experience Builder page, enter the

relevant information.

- Enter the attribute name in pascal case. The generated API name is `cfPropName`. From a custom Lightning web component or an Omniscript, reference the property name in the kebab case, such as `cf-record-limit`.
- To make the property visible for Experience Builder pages, select the **lightningCommunity__Default** target.

To call the attribute as a session variable within the Flexcard, use the {Session} merge field where supported such as text, a data source's test and input parameters, or a field element's output property. For example, if you call the attribute RecordLimit, enter {Session.RecordLimit}.

To configure the public property on a Lightning or Experience Builder page, update the value of the property in the builder's properties.

Omnistudio publishes public properties to the property subtag of the targetConfig tag in the configuration file that defines the metadata values for your Flexcard Lightning web component. You can configure additional metadata values when you publish the Flexcard.

See [XML Configuration File Elements](#) and [Activate and Publish a Flexcard](#).

15. Add or update the data source information.

See [Set Up a Data Source on a Flexcard](#).

16. If needed, test your data source configuration by adding test variables.

See [Test Data Source Settings on a Flexcard](#)

Examples for Flexcard Settings

View examples for Flexcard settings, such as adding an event listener and repeating records of a Flexcard.

Enable and Disable Data Sources

Enable and disable data source types for an org, profile, or user level from the Card Framework Configuration custom setting. For example, disable Apex REST, Apex Remote, and other complex data sources for your entire org, or prevent system admins with limited permissions from using these data sources.

Set Up a Data Source on a Flexcard

Configure how to retrieve the data that a Flexcard shows. Pull it directly from Salesforce objects with Omnistudio Data Mappers, Salesforce Object Query Language (SOQL) queries, and Salesforce Object Search Language (SOSL) searches, or an external source with REST methods. For complex business processes, use the Apex remote method or Integration Procedures. If needed, set different data sources for the child and parent Flexcards. Then, choose where to publish the Flexcard.

Flexcards Data Source Common Properties

Review the common properties available for any Flexcard data source.

Test Data Source Settings on a Flexcard

Preview a Flexcard with real data by adding test values to runtime variables. The variables populate the dynamic fields in a data source, such as a contextId from an Omnistudio Data Mapper that gets data from an Account.

Examples for Flexcard Settings

View examples for Flexcard settings, such as adding an event listener and repeating records of a Flexcard.

Add a List of Filterable Products to the Lightning Page

1. Create a filter Flexcard that triggers an event.
2. Add an Apply button that triggers an event when a customer clicks it.
3. Create a product list Flexcard that shows product information, listens for the event, and executes an action.

When a customer uses the filters on the page and clicks **Apply**, they trigger an action on the filter Flexcard that updates the product list Flexcard. The product list Flexcard then shows products that meet the filtering criteria.

Repeat Case Information

Show the list of cases for an account in these ways:

- To show a card per case, with each card listing the related fields that you added such as the status or the priority, select **Repeat Records**.
- To show all records on a single card, deselect **Repeat Records**.

Access the Context ID of a Flexcard on an Experience Site

To get data from an Apex remote, an Omnistudio Data Mapper, or Integration Procedure data source, most Flexcards use a context ID represented by the `{recordId}` context variable in an input map. For a Flexcard to use the `{recordId}` context variable on an Experience Builder page, add a record ID property to the Flexcard, then configure the property on the Experience Builder page.

1. On the Flexcard, add an exposed attribute with these properties:

Attribute

`RecordId`

Type

`String`

Targets

`lightningCommunity__Default`

Label

`Record ID`

Value

{ !recordId}

2. Save and activate your Flexcard.
3. After you add the Flexcard component to your Experience Builder page, enter {!recordId} in the Record Id field.

Enable and Disable Data Sources

Enable and disable data source types for an org, profile, or user level from the Card Framework Configuration custom setting. For example, disable Apex REST, Apex Remote, and other complex data sources for your entire org, or prevent system admins with limited permissions from using these data sources.

The screenshot shows the Salesforce Setup interface with the 'Custom Settings' page selected. The page title is 'Custom Settings' under 'SETUP'. The section title is 'Card Framework Configuration'. A note says: 'If the custom setting is a list, click New to add a new set of data. For example, if your application had a setting for country codes, each set might include the country's name and dialing code.' Another note says: 'If the custom setting is a hierarchy, you can add data for the user, profile, or organization level. For example, you may want different values to display depending on whether a specific user is running the app, a specific profile, or just the organization.' Below this, there is a table titled 'Default Organization Level Value' with columns 'Location' and 'Visibility'. It lists several items with checkboxes: 'Disable Cache' (checked), 'Disable Datasource ApexRemote', 'Disable Datasource DataRaptor', 'Disable Datasource SOQL Query', 'Disable Datasource SOSL Search', 'Enable Future Method', 'Disable Datasource IntegrationProcedures', 'Disable Datasource REST', and 'Disable Streaming API'. At the bottom, there are buttons for 'View: All' and 'Create New View', and a navigation bar with letters A-Q.

Set Up a Data Source on a Flexcard

Configure how to retrieve the data that a Flexcard shows. Pull it directly from Salesforce objects with Omnistudio Data Mappers, Salesforce Object Query Language (SOQL) queries, and Salesforce Object Search Language (SOSL) searches, or an external source with REST methods. For complex business processes, use the Apex remote method or Integration Procedures. If needed, set different data sources for the child and parent Flexcards. Then, choose where to publish the Flexcard.

For optimal flexibility and easier implementation, use an Integration Procedure as a data source to execute multiple actions in a single server call. See [Set Up an Integration Procedure Data Source](#).

If you don't see a data source, confirm that it's turned on. See [Enable and Disable Data Sources](#).

Set up the data source when you create a Flexcard or update the existing data source on an existing,

inactive Flexcard from the Setup tab. See [Configure Flexcard Settings](#).

This article describes the information required after you select the data source type for the Flexcard.

[Set Up a SOQL Query Data Source](#)

Configure a data source to use a Salesforce Object Query Language (SOQL) query to search an org's data for specific information. SOQL queries are encrypted so that Flexcards don't display query information on the client-side.

[Set Up a SOSL Search Data Source](#)

Construct text-based search queries against the search index with Salesforce Object Search Language (SOSL). By using a single query, you can search text, email, and phone fields for multiple objects to which you have access, including custom objects. SOSL queries are encrypted so the query information isn't visible on the client side. To enforce field-level security for an SOSL query, navigate to Setup and add the EnableQueryWithFLS Omni Interaction Configuration and set it to true: `EnableQueryWithFLS=true`

[Set Up an Apex Remote Data Source](#)

Configure a data source to invoke an Apex class and method that fetch data to populate a Flexcard. Apex remote data sources can run asynchronously, which increases the CPU time but ensures that long-running transactions don't time out. An Apex remote class is a standard Apex class that implements the `VlocityOpenInterface`.

[Set Up an Apex REST Data Source](#)

Configure a data source to use an Apex REST call to retrieve data to populate fields on a Flexcard with the GET method. Use the POST method to send back data to the server.

[Set Up an Omnistudio Data Mapper Data Source](#)

Configure a Data Mapper data source to fetch data from a Data Mapper Extract. Field-level security is fully supported.

[Set Up a REST Data Source](#)

Configure a REST data source by retrieving or sending data through a public API, such as weather data from a weather API based on a policyholder's ZIP code. You can also use named credentials to authenticate Apex callouts that specify the URL of a callout endpoint and its required authentication parameters in one definition.

[Set Up an Integration Procedure Data Source](#)

Configure a data source to use an Integration Procedure that executes multiple actions in a single server call. For example, you want your customer service agents to view the weather forecast of policyholders when they're on a call. To do so, use an Integration Procedure with a Data Mapper Extract that returns an account's ZIP code. Then, use a REST API call that gets the current forecast for the account's region based on the ZIP code.

[Set Up a Streaming API Data Source](#)

The streaming API method uses push technology, which is a publish and subscribe model that transfers information initiated from a server to the client based on criteria that you define. It has a persistent connection that continuously delivers new data as it becomes available, rather than using client polling. By using streaming API, you reduce the number of API calls and thus improve performance.

[Set Up a Custom Data Source](#)

Directly embed custom JSON data into Flexcard without depending on an external data source. Configure this custom data to test with temporary static data, such as when waiting for API access or for rapid concept testing.

Set Up a SOQL Query Data Source

Configure a data source to use a Salesforce Object Query Language (SOQL) query to search an org's data for specific information. SOQL queries are encrypted so that Flexcards don't display query information on the client-side.

-  **Note** When you map fields, only fields in the field picker that have non-null values for the specific record in the test query are visible. Use a test record that you know has non-null values for all the fields to map. Or use custom fields if the test record doesn't have such fields.

1. To enforce field-level security for an SOQL query:
 - a. From Setup, select and open **Omni Interaction Configuration**.
 - b. Click **New Omni Interaction Configuration**.
 - c. For Name and Label, enter *EnableQueryWithFLS*. For Value, enter *true*.
 - d. Save your changes.

When the *EnableQueryWithFLS* setting is enabled, Flexcards display only the fields that the user has permission to view. Any fields the user doesn't have access to are hidden.

 **Important** During the week of February 2, 2026, Salesforce enables the *EnableQueryWithFLS* setting by default to enhance org security. Review and prepare your configuration for a seamless transition and to prevent potential service interruptions. See [Security Checks for Omnistudio](#).

2. Enter an SOQL query and other options as needed.

For example, get up to 10 fields from the Contact object when there's an existing email address by using this query:

```
SELECT Id, Name, Email, Phone, Title FROM Contact WHERE Email != Null LIMIT 10
```

Set Up a SOSL Search Data Source

Construct text-based search queries against the search index with Salesforce Object Search Language (SOSL). By using a single query, you can search text, email, and phone fields for multiple objects to which you have access, including custom objects. SOSL queries are encrypted so the query information isn't visible on the client side. To enforce field-level security for an SOSL query, navigate to Setup and add the *EnableQueryWithFLS* Omni Interaction Configuration and set it to true: *EnableQueryWithFLS=true*

 **Important** During the week of February 2, 2026, Salesforce enables the *EnableQueryWithFLS* setting by default to enhance org security. Review and prepare your configuration for a seamless transition and to prevent potential service interruptions. See [Security Checks for Omnistudio](#).

1. Enter the search term.
2. Select the fields to search from.
3. Select at least one sObject and one field to search.
4. If needed, enter a limit of the maximum number of rows to return.
5. Enter other search information as needed.

Set Up an Apex Remote Data Source

Configure a data source to invoke an Apex class and method that fetch data to populate a Flexcard. Apex remote data sources can run asynchronously, which increases the CPU time but ensures that long-running transactions don't time out. An Apex remote class is a standard Apex class that implements the VlocityOpenInterface.

Before you begin, add an Apex class permissions checker to define who can access VlocityOpenInterface classes (APIs) from a remote action on a Flexcard. See [Set Up Access to Remote Action APIs](#).

1. Select an Apex class and method.

See [View Apex Classes](#).

This example shows an Apex class `RemoteActionClass` with a `getAccounts` method:

```
global class RemoteActionClass implements [Namespace].VlocityOpenInterface2
{
    public Boolean invokeMethod(String methodName, Map<String, Object> input,
        Map<String, Object> outMap, Map<String, Object> options) {
        if (methodName.equals('getAccounts')) {
            getAccounts(input, outMap, options);
        }
        return true;
    }
    public void getAccounts(Map<String, Object> input, Map<String, Object> out
        Map, Map<String, Object> options) {
        String accountName = String.valueOf(input.get('account'));
        List<Account> accounts = new List<Account>();

        if (String.isBlank(accountName)) {
            accounts = [SELECT Id, Name, Phone FROM Account];
        } else {
            accounts = [SELECT Id, Name, Phone FROM Account WHERE Name Like
                :('%' + accountName + '%')];
        }
        outMap.put('accounts', accounts);
    }
}
```

2. Choose whether to run the call asynchronously through Apex.

3. If needed, enter an interval in milliseconds to check the response status.
4. If needed, pass values from the Flexcard to the remote class method:
 - a. Add an input map.
 - b. In **Key**, enter a context ID variable.
 - c. In **Value**, enter the variable value.

For example, enter the AccountId key and a record's account ID as the value.

5. If needed, send additional key value pairs to the input map passed to the remote method:
 - a. Add an options map.
 - b. Enter the values.

 **Note** The Value fields support static values and merge fields such as `{recordId}` or `{name}`.

6. Enter other Apex remote data source information as needed.

Set Up an Apex REST Data Source

Configure a data source to use an Apex REST call to retrieve data to populate fields on a Flexcard with the GET method. Use the POST method to send back data to the server.

 **Note** For optimal flexibility and easier implementation, use an Integration Procedure as a data source to execute multiple actions in a single-server call. See [Set Up an Integration Procedure Data Source](#).

1. Enter the endpoint URL such as `/services/apexrest/{namespace}/CardTestApexRestResource/{recordId}`.

This example shows a REST class:

```
@RestResource(urlMapping='/v1/typeahead/*')

global with sharing class TypeaheadApexRest {
    @HttpGet
    global static void doGet()
    {
        RestResponse res = RestContext.response;
        res.responseBody = blob.valueOf('["amazon","airbnb","amazon prime","american airlines","american express","apple","aol mail","at\u0026t","apple store","adele"]');
    }

    @HttpPost
    global static void doPost()
    {
        RestRequest req = RestContext.request;
```

```
RestResponse res = RestContext.response;

res.responseBody = blob.valueOf(JSON.serialize(new Map<String, Object>{ 'someResponse' => 'Post is done'}));
}
```

2. Select a method type:
 - **GET**: Request data based on the parameters of the URL.
 - **POST**: Send JSON data.
3. For the POST method, enter the JSON data to send in the JSON Payload field.
You can use context variables to pass inherited values with either method.
4. Enter other Apex REST data source information as needed.

Set Up an Omnistudio Data Mapper Data Source

Configure a Data Mapper data source to fetch data from a Data Mapper Extract. Field-level security is fully supported.

Before you begin, create a Data Mapper Extract that has an input parameter that corresponds to the Flexcard's context ID, such as AccountId. See [Working with Omnistudio Data Mappers](#).

1. Find and select the Data Mapper to use.
2. If needed, pass values from the Flexcard to the Data Mapper's input parameters:
 - a. Add an input map.
 - b. In **Key**, enter a context ID variable.
 - c. In **Value**, enter the variable value.

For example, enter the AccountId key and a record's account ID as the value.



Note The Value field supports static values and merge fields such as `{recordId}` or `{name}`.

3. Enter other Data Mapper data source information as needed.

Set Up a REST Data Source

Configure a REST data source by retrieving or sending data through a public API, such as weather data from a weather API based on a policyholder's ZIP code. You can also use named credentials to authenticate Apex callouts that specify the URL of a callout endpoint and its required authentication parameters in one definition.

Before you begin:

- Register the REST endpoint URL in Remote Sites. See [Configure Remote Site Settings](#).
- Add the REST endpoint URL as a trusted URL. See [Manage Trusted URLs](#).

1. Select the REST type and then the method type:
 - **GET**: Request data based on the parameters of the URL.
 - **POST**: Send JSON data.
2. For the named credential REST type:
 - a. Select a named credential.
 - b. Enter a relative path for data retrieval such as `My_Payroll_System/paystubs?format=json`.
3. For the Web REST type, enter the REST endpoint URL such as `http://api.weatherstack.com/current?access_key={Session.weatherAPIkey}&query={BillingCity}`.
4. For the POST method, enter the JSON data to send in the JSON Payload field.

You can use context variables to pass inherited values with either method.

5. If needed, add request headers such as tokens, public keys, or content-type:
 - a. Add a key-value pair.
 - b. In **Key**, enter a context ID variable.
 - c. In **Value**, enter the variable value.
6. Enter other REST data source information as needed.

Set Up an Integration Procedure Data Source

Configure a data source to use an Integration Procedure that executes multiple actions in a single server call. For example, you want your customer service agents to view the weather forecast of policyholders when they're on a call. To do so, use an Integration Procedure with a Data Mapper Extract that returns an account's ZIP code. Then, use a REST API call that gets the current forecast for the account's region based on the ZIP code.

Before you begin, create an Integration Procedure. See [Omnistudio Integration Procedures](#).

1. Find and select the Integration Procedure to use.
2. If needed, pass values from the Flexcard to the Integration Procedure:
 - a. Add a key-value pair.
 - b. In **Key**, enter a context ID variable.
 - c. In **Value**, enter the variable value in `{}, {}, ...` format. Array format isn't supported.

 **Note** The Value fields support static values and merge fields such as `{recordId}` or `{name}`.

array if objects in the form of `[{}, {}]` will break interpolation logic, as it only accepts data of format `{}`,
Unable to pass array of objects as input parameters to IP Data Action from a Flexcard to [Doc] Unable to pass array of objects as input parameters to IP Data Action from a Flexcard

For example, enter the AccountId key and a record's account ID as the value.

3. If you use Omnistudio Winter '23 or later, configure your Integration Procedure to use the Continuation class in Apex to make a long-running request to an external Web service:
 - a. Add an options map.
 - b. In **Key**, enter useContinuation.
 - c. In **Value**, enter true.

See [Continuation Class](#).

4. If needed, add sample data to the Integration Procedure.

See [Response Action for Integration Procedures](#).

5. Enter other Integration Procedure data source information as needed.

Set Up a Streaming API Data Source

The streaming API method uses push technology, which is a publish and subscribe model that transfers information initiated from a server to the client based on criteria that you define. It has a persistent connection that continuously delivers new data as it becomes available, rather than using client polling. By using streaming API, you reduce the number of API calls and thus improve performance.

Before you begin, create a streaming API data source. See [PushTopic Events \(Legacy\)](#), [Generic Events \(Legacy\)](#), and [Push Technology](#).

The Streaming API data source is only supported with the designer on a managed package.

1. Select the streaming API type:
 - **PushTopic:** SOQL query that returns a result that notifies listeners of record changes in Salesforce by using a streaming API channel. For example, use this method to get notified when someone creates a case for a type, a status, or both.
 - **Streaming Channel:** Streaming API that sends notifications of general events that don't relate to Salesforce data changes. You can use this method for any application that sends custom notifications, such as a chat application.
 - **Platform Event:** Event-driven messaging architecture that enables apps to communicate inside and outside of Salesforce. A platform event combines the PushTopic and streaming channel types. Instead of working with an sObject, platform events work with custom objects.
2. In Channel, enter the URL of the streaming API.
3. Select the type of operation, whether the retrieved data replaces or adds to the existing data.
4. From the Get All Messages picklist, select one of these options:
 - **True:** Get all data from the last 24 hours.
 - **False:** Get the latest data update.
5. Enter other streaming API data source information as needed.

Set Up a Custom Data Source

Directly embed custom JSON data into Flexcard without depending on an external data source. Configure this custom data to test with temporary static data, such as when waiting for API access or for rapid concept testing.

1. Enter your custom JSON code, such as:

```
[
```

```
{  
    "attributes": {  
        "type": "Contact",  
        "url": "/services/data/v49.0/sobjects/Contact/003370  
000TOF0eAAH"  
    },  
    "Name": "Perry",  
    "Id": "0033700000TOF0eAAH",  
    "Phone": "(734) 489-5851",  
    "AccountId": "001370000NVVLvAAP",  
    "Email": "perry@vlocity.com",  
    "MailingAddress": {  
        "city": "Concord",  
        "country": "US",  
        "geocodeAccuracy": null,  
        "latitude": null,  
        "longitude": null,  
        "postalCode": "53813",  
        "state": "WI",  
        "street": "856 4th Avenue #76"  
    },  
    "CreatedDate": "2017-06-14T18:53:11.000+0000"  
},  
{  
    "attributes": {  
        "type": "Contact",  
        "url": "/services/data/v49.0/sobjects/Contact/003370  
000TOB6mAAB"  
    },  
    "Name": "Robert0 Taylor",  
    "Id": "0033700000TOB6mAAB",  
    "Phone": "(888) 888-8888",  
    "AccountId": "0013700007D63JAAS",  
    "MailingAddress": {  
        "city": "SF",  
        "country": "US",  
        "geocodeAccuracy": null,  
        "latitude": null,  
        "longitude": null,  
        "postalCode": "99999",  
        "state": "CA",  
        "street": "50 Santa Rita St"  
    },  
    "CreatedDate": "2017-06-13T21:08:25.000+0000"  
}
```

```
        },
        {
            "attributes": {
                "type": "Contact",
                "url": "/services/data/v49.0/sobjects/Contact/003370000TNvK0AAL"
            },
            "Name": "c1",
            "Id": "0033700000TNvK0AAL",
            "AccountId": "001370000NVE4tAAH",
            "MailingAddress": null,
            "CreatedDate": "2017-06-08T20:13:41.000+0000"
        },
        {
            "attributes": {
                "type": "Contact",
                "url": "/services/data/v49.0/sobjects/Contact/003370000TNvKKAA1"
            },
            "Name": "c2",
            "Id": "0033700000TNvKKAA1",
            "AccountId": "001370000NVE4tAAH",
            "MailingAddress": null,
            "CreatedDate": "2017-06-08T20:13:50.000+0000"
        },
        {
            "attributes": {
                "type": "Contact",
                "url": "/services/data/v49.0/sobjects/Contact/003370000Vs6RzAAJ"
            },
            "Name": "Smith",
            "Id": "0033700000Vs6RzAAJ",
            "Phone": "(300) 333-3763",
            "AccountId": "00137000098i8mAAA",
            "MailingAddress": {
                "city": "SF",
                "country": "US",
                "geocodeAccuracy": null,
                "latitude": null,
                "longitude": null,
                "postalCode": "99999",
                "state": "CA",
                "street": "50 Fremont St"
            }
        }
    ]
}
```

```

    },
    "CreatedDate": "2017-08-16T20:26:07.000+0000"
}
]

```

2. Enter other custom data source information as needed.

Flexcards Data Source Common Properties

Review the common properties available for any Flexcard data source.

Property	Description
Order By	Order the records by a specified field.
Reverse Order	Reverse the order of the returned records.
Fetch Timeout(ms)	Set a time that Omnistudio waits for the data source to return a response.
Refresh Interval(ms)	Set the frequency for checking data. At each refresh, if the data source records changed, the Flexcard and its child components reload.
Delay(ms)	To prevent bundling this data source call with other server requests and to specify its priority, enter a delay time in milliseconds.
Key	Enter the variable to pass. For example, pass the <i>AccountId</i> context ID.
Value	Enter the value for the variable. For example, enter a merge field such as <code>{recordId}</code> , or a static value such as the alphanumeric account ID from the URL of an Account record page.
Test Parameters	Add test variables that the query uses to get data to show when you preview the Flexcard.
Result JSON Path	Specify a path to a specific node in the JSON-formatted response, such as <code>['Cases']</code> . If your

Property	Description
	entire JSON is an array, such as <code>[{ "Cases": [{ . . . }, { . . . }] }]</code> , enter the index with the path. For example, enter <code>[0]['Cases']</code> .

Test Data Source Settings on a Flexcard

Preview a Flexcard with real data by adding test values to runtime variables. The variables populate the dynamic fields in a data source, such as a contextId from an Omnistudio Data Mapper that gets data from an Account.

1. From the Flexcards home tab, click a version of a Flexcard to open the Flexcard Designer.
2. In the Setup panel, click **+ Add New** in the **Test Parameters** section.
3. In the **Key** field, enter a test variable name. For example, if your data source is a Data Mapper, enter the variable from the Input Map representing a contextId such as `recordId`. See [Flexcards Context Variables](#).
4. In the **Value** field, enter a test variable value. For example, enter a record Id, such as the one found in the URL of an Account page.
5. (Optional) If your returned data has multiple arrays, to return a specific dataset enter its JSON path in **Result JSON Path**. For example, to return just the Cases dataset, enter `[0]['Cases']` if your JSON looks like this:

```
{
[
  "Cases": [
    {
      "Priority": "High",
      "IsEscalated": false,
      "CaseId": "5006g00000FpEz5AAF",
      "Created": "2020-07-22T00:22:42.000Z",
      "Type": "Mechanical",
      "Subject": "Generator won't start",
      "Status": "New",
      "Origin": "Web",
      "Description": "Despite turn the power on and off several times, the generator wouldn't start",
      "CaseNumber": "00001028"
    },
    {
      "Priority": "Medium",
      "IsEscalated": false,
      "CaseId": "5006g00000BsfKkAAJ",
      "Created": "2020-02-27T22:28:20.000Z",
    }
  ]
}
```

```

        "Type": "Mechanical",
        "Subject": "Shutting down of generator",
        "Status": "Closed",
        "Origin": "Web",
        "CaseNumber": "00001017"
    }
],
"Account": {
    "Type": "Customer - Direct",
    "Name": "Edge Communications",
    "Street": "312 Constitution Place\nAustin, TX 78767\nUSA",
    "State/Province": "TX",
    "City": "Austin",
    "Revenue": 139000000
}
]
}

```

6. Click Save & Fetch to view the Test Results.

The screenshot shows a modal window with the following details:

- Header:**
 - Current Version: [redacted]
 - Status: Inactive
 - Parent: KemiDataTable/cardsdailyins_lwc/1/1594158921599
 - Is Child Card: [redacted]
 - Last Modified: 8/6/2020, 4:32 PM
 - Theme: Lightning
- Content:**
 - Section: Test Results
 - Records Found: 6
 - Records Size: 1.39KB
 - Performance Metrics-Browser: 860 ms
 - Buttons: Refresh (blue), Ok (blue)
 - Table:

Total	IsEscalated	Origin	Type	Subject	Status	CaseId	Created	CaseNumber	Prio
6		Web	Mechanical	Generator won't start	New	5006g00000FpEz5AAF	2020-07-22T00:22:42.000Z	00001028	High
6	true	Phone	Electrical	Component missing	Escalated	5006g00000FpEygAAF	2020-07-22T00:21:41.000Z	00001027	High
6	true	Phone	Electrical	Starting generator after electrical failure	Escalated	5006g00000BsfKTAAZ	2020-02-27T22:28:20.000Z	00001000	High
6		Web	Other	Easy installation process	Closed	5006g00000BsfKWAAZ	2020-02-27T22:28:20.000Z	00001003	Low
6		Web	Mechanical	Shutting down of generator	Closed	5006g00000BsfKkAAJ	2020-02-27T22:28:20.000Z	00001017	Med
6		Phone	Electrical	Cannot start generator after electrical failure	Closed	5006g00000BsfKIAAJ	2020-02-27T22:28:20.000Z	00001018	Med

7. (Optional) Click **JSON** to view test results as JSON.

8. (Optional) Click **Refresh** to see the latest data.

Add Elements to a Flexcard

Build your card by dragging elements into a state on the canvas. Add data fields, display, and input elements to your Flexcard. Show data generated dynamically from the Flexcard's data source, or add static information. Rearrange, clone, delete, and adjust the widths of your elements as needed. Style an element and configure its properties in the Flexcard designer.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. To add an element to a Flexcard:
 - From the elements panel, drag an element from the Elements panel to the canvas.
 - In the designer for a managed package, drag an element from the **Build** tab.
 - Fields: Display data fields generated from the Flexcard's data source. For example, display policy information for an account.
 - Display: Add elements that show information, execute actions, or send data. For example, for a customer service portal, include the priority of a case, a button to close a case, or a banner to navigate to your company website.
 - Inputs: Add elements where users enter or select information from the defined values that you provide. For example, your portal users can enter their phone number, or select a date for an appointment.
5. Select an element on the canvas, and then add or update its properties:
 - To open the property panel, click .
 - In the designer for a managed package, click the **Properties** tab.In the properties panel, specify the element's characteristics such as the name, the size, or [show conditions](#)
6. On the Style tab, specify the element's look and feel, such as the alignment, the color, or [apply a custom style](#).

Flexcards Display Elements

Show relevant information to your users by adding display elements to Flexcards such as data fields, actions, states, data tables, and more. While we provide info bubbles for most element properties directly in the app, some elements are more complex to set up, such as custom Lightning web components. Review the considerations and requirements for those elements.

Flexcards Input Elements

Add input elements to Flexcards so that your users can provide data, which then updates the JSON data. You can also configure an element to run an action when a user interacts with the element.

Flexcards Display Elements

Show relevant information to your users by adding display elements to Flexcards such as data fields, actions, states, data tables, and more. While we provide info bubbles for most element properties directly in the app, some elements are more complex to set up, such as custom Lightning web components. Review the considerations and requirements for those elements.

Adding components, such as flyout, a custom Lightning web component, or a Flexcard, that call themselves or that are embedded in another component can lead to cyclic redundancy, causing events to run continuously and exceed the browser's maximum call size stack. For example, a Flexcard has a child Flexcard that calls an Omniscript that calls the same parent Flexcard from a flyout.

Group Elements in a Block on a Flexcard

Combine logical groups of elements in a collapsible container by using the block element. For

example, group an account's basic information in one block, and group the account's contact information in another.

Add a Chart to a Flexcard

Display data on a Flexcard as a chart by using available types such as bar, pie, donut, and more. Configure settings such as the aspect ratio or dimensions to fit your company or institution's style.

Embed a Custom Lightning Web Component

Embed components to which you add custom styling and functionality. For example, embed an existing carousel component in a Flexcard and then configure its attributes.

Show Data in a Table on a Flexcard

At times, a sortable table is most suited to represent your data. You can show data retrieved from a data source in a table on Flexcards. For example, show account cases as a sortable table.

Add a Display Field to a Flexcard

Display data fields returned from a data source on a Flexcard. For example, display Policy information for an Account.

Embed Flexcard in a Flexcard

Share data between Flexcards by embedding a Flexcard as a child in another Flexcard. The child Flexcard can have its data source, or the parent can override the child's data source with its own data source. The parent can also pass specific data to the child, such as its record ID to use as the child's context ID.

Add an Icon or an Image to a Flexcard

Add icons or images to a Flexcard from an existing library or use your own. Then, add actions, CSS classes, and configure their dimensions to fit your company or institution.

Group Actions in a Menu on a Flexcard

Create a menu from a list of actions on a Flexcard. Style the menu button, and each action in the menu's dropdown.

Display Text on a Flexcard

Keep your user informed by displaying text that combines plain text and merge fields. Format the final rendering by using a rich text editor.

Group Elements in a Block on a Flexcard

Combine logical groups of elements in a collapsible container by using the block element. For example, group an account's basic information in one block, and group the account's contact information in another.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. From the elements panel, drag the **Block** element to the canvas.
5. Enter information as needed.

The **Label** field supports concatenated strings, including a combination of plain text and supported merge fields, such as `Update {AccountName}`.

6. Drag existing elements from the canvas or elements panel into the block element.

When you add an element, Omnistudio prefixes the element name with Block, which you can change as needed.

For the block element, the **Preload Conditional Component** is enabled by default to ensure that all elements within the block load correctly.

Add a Chart to a Flexcard

Display data on a Flexcard as a chart by using available types such as bar, pie, donut, and more. Configure settings such as the aspect ratio or dimensions to fit your company or institution's style.

 **Note** Beginning Summer '22, if a chart element's height is double its expected height when displayed in a [standard Flexcard component](#), enable Lightning Web Security (LWS). For more about LWS, see [Lightning Web Security FAQ](#). To enable LWS, see [Enable Lightning Web Security in an Org](#).

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Drag the **Chart** element from the element panel to the canvas.
5. If needed, configure the chart dimensions:
 - To apply a fixed height and ignore proportions, turn off Maintain Aspect Ratio and enter a value in pixels in the Chart Height field, such as 300 or 300 px.
 - To adjust the chart's height proportionally to its original height, enter a number in the Aspect Ratio field.
 - To adjust the chart's height proportionally to its width, turn on Maintain Aspect Ratio and enter a number in the Aspect Ratio field.
6. From the label picklist, select the field whose values are used to categorize the data, such as Name to display Account names.
7. From the value picklist, select the field whose values are used to populate the chart, such as AnnualRevenue.
8. If needed, select a color palette for the chart.
If the chart has more than six values, Omnistudio repeats the colors from the color palette.
9. Enter the remaining information as needed.

Embed a Custom Lightning Web Component

Embed components to which you add custom styling and functionality. For example, embed an existing carousel component in a Flexcard and then configure its attributes.

 **Important** Don't embed the same component in multiple Flexcards.

1. From the App Launcher, find and select **Flexcards**.

2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Drag a **Custom LWC** element from the elements panel to the canvas.

Custom Lightning web components render only when you preview the Flexcard.

5. Set the component in the Custom LWC Name field depending on the component type:
 - Custom Lightning web component: Select a component that you created or that comes with Omnistudio. You can also enter the name of a component from the Lightning Component Library. For example, you can enter `lightning-map`. The names of some custom Lightning web components include the prefix `cf`.
 - Omniscript created in Omnistudio: Enter `omnistudioStandardRuntimeWrapper`, a Lightning web component wrapper that you configure in the next step.

 **Note** You can't add Flexcards created in Omnistudio or Omnistudio for Managed Packages as a custom LWC in the standard runtime. You also can't add Omniscripts created in Omnistudio for Managed Packages as a custom LWC in the standard runtime.

6. Add the component's attributes and values:
 - Use the HTML attribute format for the attribute name. For example, to use the `recordId` property, enter `record-id` for the attribute name.
 - For Omniscripts created by using Omnistudio, add the type, subtype, and language attributes to apply to the Lightning web component wrapper. At runtime, Omnistudio generates the Omniscript, and the wrapper loads it.
 - The attribute value can be a merge field such as `{Number}` or a supported context variable such as `{Session.number}`. The attribute value can also be plain text or a concatenated string that consists of plain text and a merge field such as `{Parent.parentPath}>{Parent.type}>{Name}` for a breadcrumb path.
7. Enter the remaining information as needed.

Show Data in a Table on a Flexcard

At times, a sortable table is most suited to represent your data. You can show data retrieved from a data source in a table on Flexcards. For example, show account cases as a sortable table.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Drag the data table element from the element panel to the canvas.
5. Configure the newly added table element:
 - a. Add columns in the data table and enter information as needed in the window that appears.
 - b. To let users navigate between pages using numbered links in the page, set up pagination properties by using the Page Size and Page Limit fields.
If the page size is not specified, all records are displayed in a single page. If the total number of pages in the data table is less than the page limit, all pages are shown in the navigation links.
For example, if you set the page limit to 3, each page displays three numbered links at the bottom

of the page. The first and last pages are always shown in the navigation links. Any remaining pages are shown as ellipsis.

- c. To delete a record when a user sets a field value to false, enter a field name in the **Field Determining Row Deletion** field. If you're using the designer on a managed package, enter a field name in the **Row Delete Dependent Column** field.

Make sure that delete row is enabled for the data table.

For example, a table lists account cases and has a boolean record field `IsEscalated`. When `IsEscalated` is false, a user can delete the record. When the field is true, hide the trash icon to prevent users from deleting the record.

- d. Enter the remaining information as needed.

 **Note** If you hide a field by using the `Is Visible` attribute, a user can't search the column associated with the field, even if the `Is Searchable` attribute is set to true.

You can't add a clickable field in a data table. As a workaround, select the **User Selectable Row** attribute, which triggers a `rowclick` event. Then, in the Flexcard, add an event listener that uses a merge field action that contains the row data. You do this by clicking **Event Listener** in the Setup tab of the Flexcard. The merge field must be in the `{action.result.X}` format, where X is the `fieldName` from the data table. This action fetches data into the data table.

Review the events that users trigger when they interact with data table records. Each attribute that you set on the data table element when you create the Flexcard corresponds to an event.

Event	Description	Checkbox in the Data Table Properties
delete	The user deletes a row.	<input type="checkbox"/> Delete row <input checked="" type="checkbox"/> Checkbox in designer for a managed package: Row Delete
rowclick	The user clicks a row. The <code>rowclick</code> event contains the record of the row on which the user clicks.	–
selectrow	The user updates a row. The <code>selectrow</code> event contains data for the currently selected row. It doesn't consider previously selected rows. To collect data for all <code>selectrow</code> events, set up event handling.	<input type="checkbox"/> Let users select rows <input checked="" type="checkbox"/> Checkbox in designer for a managed package: User Selectable Row
update	The user updates table data.	<input type="checkbox"/> Edit cells and Edit rows

Event	Description	Checkbox in the Data Table Properties
		Checkbox in designer for a managed package: Cell Level Edit and Row Level Edit

Add a Display Field to a Flexcard

Display data fields returned from a data source on a Flexcard. For example, display Policy information for an Account.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Drag the **Field** element from the element panel to the canvas.
5. Enter information as needed:
 - The Placeholder, Label, and Output fields support concatenated strings, including a combination of plain text and supported merge fields with custom labels, such as `{Label.AccountName} > {Name}` or `{Name} + {Id}`.
 - For date fields, ensure that the date value matches the user locale format. For example, your date value comes from the custom data source `[{"DOB": "28-03-2015"}]`, and your locale is en_US, which has a format of M/D/YYYY. Because months only go up to 12, the number 28 doesn't work which results in an incorrect date. Instead, unless you localize your application, update the format to match the date to `DD/MM/YYYY`, or select a text field type.
 - If you don't select the user locale for formatting, the Flexcard author's user locale determines the format for the date, time, and currency.

Embed Flexcard in a Flexcard

Share data between Flexcards by embedding a Flexcard as a child in another Flexcard. The child Flexcard can have its data source, or the parent can override the child's data source with its own data source. The parent can also pass specific data to the child, such as its record ID to use as the child's context ID.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Drag the **Flexcard** element from the element panel to the canvas.

If you turned on Omniscript support on your Flexcard in Setup, only select child Flexcards that also have that setting turned on.

5. If your Flexcard is a child Flexcard with hierarchical data, and to add it to itself repeatedly, select **Is Recursive**.

When you apply conditions to a recursive child Flexcard, Omnistudio applies them to all recursively repeating records. If one record condition returns false, nested records don't show.

In preview or at run time, the child Flexcard passes the same child data field (also called a data node) available in each level of the JSON code, until that data field no longer exists or is empty. As a result, the Flexcard displays the children and grandchildren of the parent objects.

6. To pass an object from the parent Flexcard's data source instead of using the child Flexcard's data source, select a data node:
 - **{records}**: Pass all data.
 - **{records[#]}**: Pass the data of a specified record. For example, `{records[0]}` is the first record, `{records[1]}` is the second record, and so on.
 - **{record}**: Pass the current record's data, such as when you deselect **Repeat Records** in the setup options.
 - **{record.FieldName}**: Pass a record object, for example use `{record.Product}` for an object or array with additional data.
 - **{record.attributes}**: Send all attributes for the current record.
 - For a recursive Flexcard, pass objects within a specific dataset from the child Flexcard to itself by entering the repeating child node.

If the child card depends on the parent data, any change to the data in the parent Flexcard re-renders the child Flexcard.



Note Make sure that the fields to populate on the child Flexcard are available on the parent Flexcard's data source. For example, if your child Flexcard shows data from the Status and Priority fields from the Case object, your parent data source must have the same fields.

7. To pass specific attributes, enter the attribute name and value, for example `AccountId` and `{Id}`.
8. **Note** The Value field supports concatenated strings, including a combination of plain text and supported merge fields. For example, to pass a breadcrumbs path, enter `{Parent.parentPath}>{Parent.type}>{Name}`.
9. On the child Flexcard, under Data Source in Setup, reference the attribute by using a merge field or plain text, for example `{Parent.AccountId}` in a SOQL query.

Pass Data From a Child Flexcard to a Parent Flexcard

After you embed a Flexcard in another Flexcard, pass data from the child Flexcard to its parent by using input parameters from a custom event action on the child. Then, set up an event listener to update the parent Flexcard's data field in response to the event on the child Flexcard.

1. Open the child Flexcard.
2. Drag the **Action** element from the element panel to the canvas.
3. From the properties panel, expand the **Action** section.
4. Select the **Event** action type and the **Custom** event type.
5. Enter an event name such as `updatecity`.

6. Pass contextual data:
 - a. Add an input parameter.
 - b. In **Key**, enter a name to call from the parent such as `cityname`.
 - c. In **Value**, enter the value to pass as plain text such as `New York`, or as a merge field from the child's data source, such as `{City}`.
7. Activate the child Flexcard.
8. Open the parent Flexcard.
9. From Setup, add an event listener.
 - a. Enter the same event name as for the custom event on the child Flexcard, such as `updatecity`.
 - b. Expand the `Action` section.
 - c. Select the `Card` action type and the **Set Values** type.
 - d. In **Key**, enter the data field to update.
 - e. In **Value**, enter the name of the input parameter passed from the child Flexcard, by using the `action.param` context variable. For example, if the key from the input parameter is `cityname`, enter `{action.cityname}`.
 - f. Save your changes.

In preview, click the action on the child Flexcard to update a data field on the parent.

Display Account and Case Information

You want to display account information on a parent Flexcard and case information on the child Flexcard using a data table.

1. In your child Flexcard, deselect Repeat Records in the Setup options so that the Flexcard shows one table.
2. Add a data table to the child Flexcard and configure the data source to get the case information.
3. Activate the child Flexcard.
4. On the parent Flexcard, configure the data source to get the account information and the information from all cases related to the account:

```
[  
 {  
     "PrimaryContactEmail": "sean@edge.com",  
     "Revenue": 139000000,  
     "Website": "http://edgecomm.com",  
     "Phone": "(512) 757-6000",  
     "Name": "Edge Communications",  
     "Id": "0013g00000AeI9fAAF",  
     "Cases": [  
         {  
             "Type": "Mechanical",  
             "CaseId": "5003g000007WQHRAA4",  
             "Created": "2020-11-23T20:51:20.000Z",  
         }  
     ]  
 }
```

```
        "Priority": "High",
        "Status": "Escalated",
        "Subject": "The watchamacallit is too high"
    },
{
    "Type": "Structural",
    "CaseId": "5003g000007WQHQAA4",
    "Created": "2020-11-23T20:50:32.000Z",
    "Priority": "Medium",
    "Status": "New",
    "Subject": "The doodat wasn't properly set up",
    "CaseId": "5003g000007WQHQAA4"
}
]
}
```

5. Add the child Flexcard to the parent Flexcard.
6. In Data Node, enter `{records[0].Cases}`.

Omnistudio ignores the child data source and populates the child Flexcard data table with the parent Flexcard's data.

Pass Contact ID to Child Flexcard as a Context ID

A parent Flexcard displays account information and its child Flexcard displays the account's primary contact information.

1. To get the data source on the child Flexcard, enter this SQL query:

```
Select Id,Name,Email,Phone,MailingStreet,MailingCity,MailingState,MailingPostalCode,MailingCountry
from Contact where Id = '{Parent.contactId}'
```

2. Activate the child Flexcard.
3. On the parent Flexcard, configure the data source to get account information including the primary contact ID.
4. Pass the primary contact ID to the child Flexcard by entering `contactId` as the attribute and `{namespace__PrimaryContactId__c}` as the value.

Data Node for Recursive Flexcard

A recursive Flexcard has hierarchical JSON data with a `Products` data field that lists products within a product:

```
[  
  {  
    "Id": "5123",  
    "Name": "Exit Communications",  
    "Account Number": "QHT123456789",  
    "Phone": "415-987-6543",  
    "Website": "http://www.exitcomm.org",  
    "Contact": "Melrose Oduke",  
    "Contact Phone": "415-333-1234",  
    "Products": [  
      {  
        "Id": "1",  
        "Name": "Product1",  
        "Products": [  
          {  
            "Id": "1_1",  
            "Name": "Product1_1"  
          },  
          {  
            "Id": "1_2",  
            "Name": "Product1_2",  
            "Products": [  
              {  
                "Id": "2",  
                "Name": "Product2",  
                "Products": [  
                  {  
                    "Id": "2_1",  
                    "Name": "Product2_1"  
                  },  
                  {  
                    "Id": "2_1_2",  
                    "Name": "Product2_1_2"  
                  }  
                ]  
              }  
            ]  
          }  
        ]  
      }  
    ]  
  }  
]
```

To display the products, their subproducts, and their sub-subproducts:

1. Embed the child Flexcard once inside itself so that it passes the products data field to itself.
2. In Data Node, enter `{record.Products}`.

Add an Icon or an Image to a Flexcard

Add icons or images to a Flexcard from an existing library or use your own. Then, add actions, CSS classes, and configure their dimensions to fit your company or institution.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Drag the **Icon or Image** element from the elements panel to the canvas.
5. Enter the icon name or image source information:
 - a. To use a custom icon or image from an external source, enter the absolute URL of the image. Ensure that you add that URL to the trusted URLs list. See [Manage Trusted URLs](#).
 - b. To use an existing file, search for it by clicking the magnifier icon. When you upload a local image, Salesforce copies it to the Content Document library for reuse.

 **Note** Salesforce recommends using an image or icon as a static resource instead of uploading it.

- c. For images and icons, to use a file uploaded as a static resource (Image: Image source; Icon: Icon Source URL), enter `/resource/resourceName`, where resourceName is the name of the static resource.

To create a static resource, see [Define Static Resources](#).

 **Note** Images used in Flexcards on Experience Cloud pages must be static resources.

6. Enter the remaining information as needed.
-  **Note** For images, the Title field supports concatenated strings, including a combination of plain text and supported merge fields, such as `{Label.AccountName} > {Name}`.

Group Actions in a Menu on a Flexcard

Create a menu from a list of actions on a Flexcard. Style the menu button, and each action in the menu's dropdown.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Drag the **Menu** element from the element panel to the canvas.
5. Enter information as needed.

The Label field supports concatenated strings, including a combination of plain text and supported

merge fields, such as `Update {AccountName}`.

Display Text on a Flexcard

Keep your user informed by displaying text that combines plain text and merge fields. Format the final rendering by using a rich text editor.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Drag the **Text** element from the element panel to the canvas.

Omnistudio uses the TinyMCE editor by default. If needed, you can enable Lightning Rich Text Editor. See [Differences Between Lightning and TinyMCE Rich Text Editors](#) and [Enable Lightning Rich Text Editor in Omniscripts and Flexcards](#).

5. Enter content and div tags as needed:
 - a. Enter plain text.
 - b. Select fields or enter the field name in the {field name} format, such as `{Type}`.
 - c. Enter custom labels in the {Label.customLabelName} format. For example, to add a custom label for AccountName, enter `{Label.AccountName}`.
 - d. Enter a context variable as a merge field.
For example, to add the logged-in user's profile name, enter `{User.userProfileName}`. The rendered User global context variable is visible at run time only, such as when you preview or publish the Flexcard.
 - e. To add a CSS class at the paragraph level, select a class from the Div picklist.
 - f. Apply more formatting as needed.

Flexcards Input Elements

Add input elements to Flexcards so that your users can provide data, which then updates the JSON data. You can also configure an element to run an action when a user interacts with the element.

For all input elements, you can configure these settings:

- Common parameters such as field binding to update data fields based on the information that users enter.
- Lightning web component custom attributes such as conditional checkbox color or specific labels when users hover on a toggle.
- Actions, conditions, or style formats, similar to other Flexcard element types. See [Set Up Actions on a Flexcard](#), [Set Up Conditions to a Flexcard Element](#), and [Style a Flexcard](#).

[Update Data Field Based on Value Entered by User](#)

Ensure that your data is always up to date by binding an input element value to a data field in your Flexcard's data source. As a result, when a user updates the input value, Omnistudio updates the corresponding JSON code. You can use the new data like any other piece of data from a data source. For example, update the value of an attribute sent from a parent to a child Flexcard based on user

input.

Element-Specific Considerations for Flexcard Input Elements

Review some element-specific considerations and learn which Lightning web component the element maps to, and the attribute that you can configure. Then, access the corresponding readme file for information about the custom attribute setup.

Example: Suspend a Device from Self-Service

Learn how to set up a Flexcard that your customers use to suspend a mobile device from a self-service portal.

Update Data Field Based on Value Entered by User

Ensure that your data is always up to date by binding an input element value to a data field in your Flexcard's data source. As a result, when a user updates the input value, Omnistudio updates the corresponding JSON code. You can use the new data like any other piece of data from a data source. For example, update the value of an attribute sent from a parent to a child Flexcard based on user input.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Drag an Input element from the elements panel to the canvas.
5. Open Properties, and click **Field Binding**.
6. To confirm the data JSON updates when the input value changes an existing data field, perform the following tasks:
 7. Select an existing data field, or to create a field on the Flexcard's data source, enter a unique name. Use only alphanumeric characters without spaces.

When you create a data field, the default value is empty. After you bind the field to an existing field, the default value is the value in the data source.

8. Confirm the data JSON update:
 - a. Click **Preview**.
 - b. Enter data in the bound field on the canvas.
 - c. In the Data JSON panel, find the updated data field and check the value. The panel shows new fields only after you enter data on the canvas.

Element-Specific Considerations for Flexcard Input Elements

Review some element-specific considerations and learn which Lightning web component the element maps to, and the attribute that you can configure. Then, access the corresponding readme file for information about the custom attribute setup.

See [Base Omnistudio LWC ReadMe Reference](#).

Input Element	Description or Consideration	Lightning Web Component in ReadMe
Checkbox	Select a checkbox by default.	CheckboxGroup
Currency	The default value is based on the logged-in user's locale settings, such as USD. If the user locale doesn't have the currency specified, currency code from company information is used.	Input when type = currency
Date	<p>Supports custom labels.</p> <p>If needed, enter a default value in the selected date and time formats.</p>	DatePicker
Date/Time	<p>Supports custom labels.</p> <p>If needed, enter a default value in the selected date and time formats.</p>	DateTimePicker
Email	–	Input when type = email
Multi-Select	Display options as checkboxes, buttons, images, or as a dropdown.	<ul style="list-style-type: none"> • CheckboxGroup for checkboxes and buttons • CheckboxImageGroup for images • Combobox for dropdowns
Number	<p>If needed, define a minimum and maximum mask value by setting the min and max custom attributes. These fields don't support merge fields.</p> <p>Users can enter numbers of any length regardless of the mask length.</p>	Input when type = number
Radio	If you choose the color display mode, enter the color in the Value field under Options as a CSS color name such as <code>Yellow</code> , or a HEX code such as <code>#ffff00</code> . See Color	<ul style="list-style-type: none"> • RadioGroup for radios and buttons • RadioImageGroup for images • RadioColorPickGroup for colors

Input Element	Description or Consideration	Lightning Web Component in ReadMe
	Names Supported by All Browsers and Colors HEX.	
Range	The Step field determines the incremental increase in the range. For example, for a range from 1 through 3, a step of .5 lets users select values between 1, 1.5, 2, 2.5, and 3.	Input when type = range
Select	–	Combobox when multiple = false
Telephone	–	Input when type = tel
Text	<p>When you set the mask property on text fields, Salesforce shows the default error message instead of custom error messages for those fields.</p> <p>Flexcards do not support curly braces {} in text fields as those are reserved for merge field syntax and cannot be used in a string or text value.</p>	Input when type = text
Text Area	<p>By default, Omnistudio uses the <code>\n</code> element instead of the <code>
</code> element to create line breaks.</p> <p>Flexcards do not support curly braces {} in text area fields as those are reserved for merge field syntax and cannot be used in a string or text value.</p>	Textarea
Time	If you specify a default value, ensure that it's in the selected time format.	TimePicker
Toggle	Let users trigger actions and send values as true or false parameters, or a comma-separated list of options that they select with the	Toggle

Input Element	Description or Consideration	Lightning Web Component in ReadMe
	<p><code>{element.value}</code> merge field.</p> <p>Field binding isn't supported with the Checkbox Group and Checkbox Group Button toggle types. For the other types, provide data field information in the Checked field instead of the Field Binding field.</p> <p>To disable the toggle element based on the boolean value of a merge field, select that merge field from the Disabled picklist.</p> <p>For example, a Flexcard shows open cases and has a toggle to indicate that it's closed, and an Is Escalated field. You can disable the toggle for escalated cases so that support agents can only close cases after they solve the de-escalate the case.</p>	
Typeahead	–	Typeahead

Example: Suspend a Device from Self-Service

Learn how to set up a Flexcard that your customers use to suspend a mobile device from a self-service portal.

1. Create a Flexcard whose data source:
 - Gets a list of a customer's active mobile devices.
 - Returns an IsSuspended data field with a true or false value.
2. Add an image element that displays a picture of the device.
3. Add a toggle element that meets these criteria:
 - Customers use the toggle to select the device to suspend.
 - You bind the value of the toggle to the IsSuspended data field.
 - You add an Update Omniscript action to the toggle.
4. Include the Flexcard in an Omniscript that uses a Data Mapper Post Action to update the status of the mobile device in the customer's account.

Set Up Actions on a Flexcard

Configure actions that users can complete on Flexcards to improve the self-service experience, and save time on manual tasks. Add actions to supported elements or add an action element to a Flexcard, and then configure the action settings. For example, you want to have a button that opens a URL for additional information. To do so, add an action to an existing image or add an action element for which you select an image. You can also set up sequential actions and configure each action's trigger.

These steps walk you through setting up actions from the properties panel. It includes more fields than when you add actions to existing elements from Setup such as an event listener. In those cases, ignore the steps or fields that don't apply, such as the option for blocking interactions until actions are complete.

1. From the App Launcher, find and select **Flexcards**.
2. From the list page, select a Flexcard version.
3. If needed, deactivate the Flexcard.
4. In the Flexcard Designer, drag the **Action** element into a state on the canvas.
5. To add an action to:
 - An existing element such as an image, a block, an icon, or a toggle: Select the element on the canvas. Then, add an action from the properties panel.
 - The Flexcard: Drag the **Action** element from the elements panel to the canvas. Then, open **Properties**.
6. Enter information as needed.
 **Note** The Label field supports concatenated strings, including a combination of plain text and supported merge fields with custom labels, such as `{Label.AccountName} > {Name}` or `{Name} + {Id}`.
7. Expand the **Action** section.
8. To prevent users from interacting with the Flexcard when an action is in progress, enable the option for blocking interactions until actions are complete, or enter a merge field whose value is `true` such as `{requireResponse}`.

For example, lock a form while the user waits for the Flexcard to refresh after they click an Update button to which you added a Data action.

9. Select an action type.
10. If needed, turn on tracking to record the card load, card unload, and state load events at the Flexcard level, and the UI action event at the action level.

By default, Omnistudio tracks parent Flexcards but not child Flexcards. To track child Flexcards, turn on the setting on the child Flexcard and all their parents. When you embed Flexcards inside other Flexcards, we recommend that you check and confirm the setting on all Flexcards.

11. To set up sequential actions, add an action.
12. To rearrange the actions sequence, drag an action to the desired position.



Tip If you have any action that navigates away from the Flexcard, add them as the last action in the queue, or set them up to stop any further actions. For a data action type, save the response in a response node to use it in a subsequent action. In preview or at run time, Omnistudio executes the actions from top to bottom, waiting for each action to complete before triggering the next action. You can use the response from the previous action to perform subsequent actions. If an action fails, Omnistudio doesn't run the subsequent actions.

When an action triggers sequential custom or pubsub events, and each of their event listeners trigger their own sequential actions, a failing action doesn't prevent the next sequence of actions in the next event listener from executing.

For example, set up a sequence on an actionable element or an event listener that:

- Updates a data value with the set values action.
- Passes the updated value to an Omniscript with an update Omniscript action.
- Navigates to a record page with the navigate action.

13. If needed, save your changes.

Set Up Card Actions to Update a Flexcard

Configure actions that users can trigger and that affect a record on a Flexcard or the Flexcard itself. For example, let users arrange a records view to suit their personal preference.

Get or Send Data from or to a Flexcard

Transfer data from or to a server by calling a data source. When Omnistudio receives the data, it adds it to the JSON data in the card's record object or to a data node that you specify. For example, populate a child Flexcard with data from its parent, or conditionally show messages from an API response.

Pass Data to Flexcards by Using Events

When an event occurs, a notification is sent to a Flexcard through an action on an element. Then, execute an action based on the event. Set up events to communicate between a child Flexcard and its parent, between an element and the Flexcard that it's on, or between distinct components. For example, add an action on an embedded child Flexcard that, when clicked, updates a field on the parent Flexcard. Or, add an action on an image that when clicked, updates a merge field value in a text element.

Set Up Flyouts on a Flexcard

Display additional information from a child Flexcard, an Omniscript, or a custom Lightning web component when a user clicks an action on a Flexcard. For example, show account information on a Flexcard and the primary contact's name and email address in a flyout.

Set Up Navigation on a Flexcard

Enable navigation to an external URL, a Lightning page, a page in an Experience Builder Aura site, or any Salesforce page. For example, when a policyholder views their account policies on a Flexcard, they can click a button that opens a selected policy record page.

Launch an Omniscript from a Flexcard

Set up a Flexcard to call an Omniscript that updates the Flexcard's data source. For example, users enter their account ID on a Flexcard. Then, when they click the Omniscript action, Omnistudio passes the entered value and launches a form that updates their account information.

[Update an Omniscript's JSON Code from a Flexcard](#)

Specify the JSON node that Omnistudio updates in response to a user action on the Flexcard by embedding the Flexcard in the Omniscript as a custom Lightning web component.

Set Up Card Actions to Update a Flexcard

Configure actions that users can trigger and that affect a record on a Flexcard or the Flexcard itself. For example, let users arrange a records view to suit their personal preference.

After you select the card action type, select the specific type of operation for your users to launch on the Flexcard.

[Reload](#)

Refresh the data shown on the Flexcard such as in response to an event. For example, create a Flexcard that listens for an event from another Flexcard and reloads itself when the event is fired.

[Remove a Record](#)

Remove a Flexcard from the page that shows it, or a specific record from the Flexcard. Omnistudio removes the Flexcard or record from the page that the user views, but not from the data source.

[Select Card](#)

Select records from a list on a Flexcard, such as to add or remove shopping cart items, or to select support cases. Then, create a custom event that passes the selected records to an Omniscript.

[Set Values](#)

Update field values on a Flexcard, such as for customer service reps to update the status of a case when they resolve a case.

[Update Datasource](#)

Change the data source or update parameters of an existing data source, such as updating the pagination limits on a list of returned records.

Reload

Refresh the data shown on the Flexcard such as in response to an event. For example, create a Flexcard that listens for an event from another Flexcard and reloads itself when the event is fired.

See [Configure Flexcard Settings](#).

Remove a Record

Remove a Flexcard from the page that shows it, or a specific record from the Flexcard. Omnistudio removes the Flexcard or record from the page that the user views, but not from the data source.

For information about permanently removing a record from the data source if the data source is from an Omniscript, see [Update an Omniscript's JSON Code from a Flexcard](#).

1. From the Flexcard Designer, drag an **Action** element into a state.

2. Select **Card** from the **Action Type** dropdown.
3. From the **Type** dropdown, select **Remove**.
4. To remove a specific record, create an Event Listener, and enter a **Record Index**.

Select Card

Select records from a list on a Flexcard, such as to add or remove shopping cart items, or to select support cases. Then, create a custom event that passes the selected records to an Omniscript.

 **Note** All actions with a type of Select Card on the same Flexcard use the same list, even when the actions have different names. When you enter values in Selected Cards List Name, Selectable Mode, or Selectable Field, Omnistudio populates those values across all Select Card actions on the Flexcard

1. In **Selected Cards List Name**, enter a value that uses only lowercase and alphanumeric characters and no spaces, such as `selectedcaseslist`.

To use the default name, `selectedcards`, leave the field empty.

2. If needed, to show a preselected record to the user, enter a boolean data field.

For example, you have case records with an `IsEscalated` field that indicates the escalation status. To send a list of preselected escalated case records, enter `IsEscalated`. In the JSON code in preview, all cards where `IsEscalated` is `true` are selected.

3. Enter the remaining information as needed.
4. In Setup, turn on Omniscript support.
5. Add an event listener.
6. Select the custom event type.
7. Enter `selectcards_` and the name of the selected card list as the event name, such as `selectcards_selectedcaseslist`. If you use the default list name, enter `selectcards_selectedcards`.

8. Expand the **Action** section.

9. Select the **Update Omniscript** action type.

For example, to send a list of preselected escalated case records to an Omniscript, enter `IsEscalated` in the Selectable Field field. In Preview, all cards where `IsEscalated` is true are selected.

10. Add these input parameters:

- a. In Key, enter the name for the data node to pass to the Omniscript, such as `selected`.
- b. In Value, enter the name of the selected cards list as a variable, such as `{selectedcaseslist}`.

11. Save your changes.

When you preview the Flexcard, in the JSON code under `_flex`, all cards where `isSelected` is `true` are selected.

12. Embed your Flexcard in an Omniscript as a custom Lightning web component.

See [Embed Flexcards in an Omniscript](#).

When the user launches the action, Omnistudio executes the `selectcards_{listname}` event. This event adds the selected records to the `{listname}` list, and updates the Omniscript's JSON code with the selected records.

For information about clearing the selection so that users can pick other records, see [Example Use Cases for Flexcard Events](#).

Set Values

Update field values on a Flexcard, such as for customer service reps to update the status of a case when they resolve a case.

1. Add a key-value pair for Set Values.
2. In Key, enter one of these values:
 - The field name returned from a data source
 - A session variable to update such as `{Session.time}`.
See [Flexcards Context Variables](#).
 - A private variable name for configuring conditions in the Flex.varname format, where varname is an alphanumeric string, excluding index, which gets the Flexcard index. When record data changes, private variables remain the same.
For example, you can set a condition on an element where the element displays for the first three records only. If your condition is `Flex.index < 3`, the element displays on card[0], card[1], and card[2]. For more examples of conditional display, including by using a private variable, see [Display Data on Flexcards Based on Conditions](#).
3. In the Value field, enter a new field value as plain text, a merge field, or a concatenated string of both, such as `New Value`, `{Name}`, or `{Parent.type} > {Name}`.

Update Datasource

Change the data source or update parameters of an existing data source, such as updating the pagination limits on a list of returned records.

Add a data source, or update the existing data source settings.

See [Set Up a Data Source on a Flexcard](#).

Get or Send Data from or to a Flexcard

Transfer data from or to a server by calling a data source. When Omnistudio receives the data, it adds it to the JSON data in the card's record object or to a data node that you specify. For example, populate a child Flexcard with data from its parent, or conditionally show messages from an API response.

1. After you select the data action type, set up data source information.

See [Set Up a Data Source on a Flexcard](#).

2. To skip updating the JSON data, select **Ignore Response**.
3. In Response Node, specify where to save the returned data:
 - If the request returns one object, enter record. Omnistudio inserts the data at the root of the card object.

Returned Data	JSON Data
<pre>//One record returned { "Type": "Mechanical", "Subject": "The watchamacallit is too high", "Status": "Escalated", "Priority": "High", "CaseId": "5003g000007WQHRAA4", }</pre>	<p>Data JSON Action Debugger</p> <pre> 1 ▼ Object 2 ► User: Object 3 ► Session: Object 4 ► Params: Object 5 ▼ TestParams: Object 6 ▼ Label: Object 7 ► dataSource: Object 8 title: "KemiTestCallDatasourceAction" 9 theme: "lightning" 10 recordId: "a073g000007GgjxAAC" 11 ▼ cards: Array[1] 12 ▼ 0: Object 13 StateName: "Active" 14 ► ChildCards: Array[2] 15 Id: "5003g000007WQHRAA4" 16 ► attributes: Object 17 Name: "Adeola Widgets" 18 Phone: "925-7373-37377" 19 uniqueKey: "RECO" 20 Type: "Mechanical" 21 Subject: "The watchamacallit is too high" 22 Status: "Escalated" 23 Priority: "High" 24 CaseId: "5003g000007WQHRAA4" 25 ► Flex: Object </pre>
<pre>//One array object returned { "contacts": [{ "AccountId": "0013g00000AeI9fAAF", "Name": "Rose Gonzalez", "Birthdate": "1968-09-26", "CreatedDate": "2020-10-15T21:28:48.000+0000", "Phone": "(510) 757-6000", "Id": "0033g0000EXVKbAAP" }, { "AccountId": "0013g00000AeI9fAAF", }] }</pre>	<p>Data JSON Action Debugger</p> <pre> 1 ▼ Object 2 ► User: Object 3 ► Session: Object 4 ► Params: Object 5 ▼ TestParams: Object 6 ▼ Label: Object 7 ► dataSource: Object 8 title: "data_action_1" 9 theme: "lightning" 10 recordId: "a073g000007GLDsAAO" 11 ▼ cards: Array[10] 12 ▼ 0: Object 13 StateName: "Active" 14 ► ChildCards: Array[0] 15 Id: "0013g00000KmQX4AAN" 16 ► attributes: Object 17 Name: "Adeola Widgets" 18 BillingCity: "Walnut Creek" 19 uniqueKey: "RECO" 20 ► contacts: Array[2] 21 ► 0: Object 22 ► 1: Object 23 ► Flex: Object 24 ► 1: Object 25 ► 2: Object 26 ► 3: Object 27 ► 4: Object 28 ► 5: Object 29 ► 6: Object 30 ► 7: Object 31 ► 8: Object </pre>

Returned Data	JSON Data
<pre> "Name": "Sean Forbes", "Birthdate": "1947-06-25", "CreatedDat e": "2020-10-15T21:28:48.000+000 0", "Phone": "(51 2) 757-6000", "Id": "0033g0 000EXVKcAAP", }] } </pre>	

- If the request returns multiple records or to add the data to a new data node, enter an alphanumeric name with no spaces or special characters such as `caseList` or `caseList1`.

Returned Data	JSON Data
<pre> //multiple records returned [{ "Type": "Mech anical", "Subject": "T he watchamacallit is too high", "Status": "Es calated", "Priority": "High", "CaseId": "50 03g000007WQHRAA4" }, { "Type": "Stru ctural", "Subject": "T he doodat wasn't properly set up", "Status": "Ne w", </pre>	<p>JSON Data</p> <p>Data JSON Action Debugger</p> <pre> 1 ▼ Object 2 ► User: Object 3 ► Session: Object 4 ► Params: Object 5 ▼ TestParams: Object 6 ▼ Label: Object 7 ► dataSource: Object 8 title: "data_action_1" 9 theme: "lightning" 10 recordId: "a073g000007GLDsAAO" 11 ▼ cards: Array[10] 12 ▼ 0: Object 13 StateName: "Active" 14 ► ChildCards: Array[0] 15 Id: "0013g00000KmQX4AAN" 16 ► attributes: Object 17 Name: "Adeola Widgets" 18 BillingCity: "Walnut Creek" 19 uniqueKey: "RECO" 20 ▼ caseList: Array[5] 21 ► 0: Object 22 ► 1: Object 23 ► 2: Object 24 ► 3: Object 25 ► 4: Object 26 ► Flex: Object 27 ► 1: Object 28 ► 2: Object 29 ► 3: Object 30 ► 4: Object 31 ► 5: Object </pre>

Returned Data	JSON Data
<pre> "Priority": "Medium", "CaseId": "50 03g000007WQHQAA4" }]</pre>	<pre> "Priority": "Medium", "CaseId": "50 03g000007WQHQAA4" }</pre>

 **Note** For sequential actions, Omnistudio uses the data stored in Response Node.

To access the data, use the `{dataResponseNodeName.dataNode}` syntax.

For example, if the response node name is `contactList`, then to return an error message from an `error` data node, enter `{contactList.error}`.

4. To confirm your request response, access the action debugger in preview, and then inspect these sections:
 - **response:** The requested data from the data action.
 - **records:** The records from the Flexcard's data source and all data actions that appended records to the JSON data.
 - **node:** The name of the response node that holds the data action records.
 - **ignoreResponse:** Indicator that you choose not to update the JSON data. If you select Ignore Response, the JSON data shows this parameter and sets it to `true`.
5. To confirm that Omnistudio added the records to the JSON data, in preview, expand the cards node, and then search for the response node name.

Use Cases to Transfer Data by Using Flexcard Actions

- Send Account Cases Data to Child Flexcard
 1. On a parent Flexcard, create a data action that retrieves the case information with a response data node called `caseList`.
 2. On the child Flexcard, in Data Node, enter `{record.caseList}`.
- Display Messages from API Response
 1. Include the error message returned from a response in a text element, such as `{caseList.error}`.
 2. Add a condition on the element so that the Flexcard shows the text element only when an error message exists.

Pass Data to Flexcards by Using Events

When an event occurs, a notification is sent to a Flexcard through an action on an element. Then, execute an action based on the event. Set up events to communicate between a child Flexcard and its parent, between an element and the Flexcard that it's on, or between distinct components. For example, add an action on an embedded child Flexcard that, when clicked, updates a field on the parent Flexcard. Or, add an action on an image that when clicked, updates a merge field value in a text element.

1. After you select the event action type, select the event type:

- **Custom:** Notify events between a child and its parent, such as a child Flexcard and its parent Flexcard, or an element and the Flexcard that it's on.
- **PubSub:** Notify events between distinct components, such as between Flexcards.

2. Enter an event name. Don't use a reserved name.

To register dynamic events based on a parent attribute or a session variable, use these context variables:

- `{Parent.attr}` : Use an attribute passed to a child from its parent Flexcard such as `event_{Parent.closed}`.
- `{recordId}` : Use a stored value defined as a session variable of the Flexcard, or a public property defined as an exposed attribute of the Flexcard such as `chnnl_{Session.tab}`.
- `{Session.var}` : Use the ID of the record page that the Flexcard is on such as `accountEvent_{recordId}`.

3. For pubsub events, enter a channel name with the same criteria as for the event name.

By default, the channel name is the Flexcard name.

4. For custom events, consider how the event passes through elements and select the checkboxes accordingly:

- a. When you trigger a custom event from a nested element such as a block, select **Bubbles**.
- b. When components communicate with each other, such as a child Flexcard and its parent, or a Flexcard and the parent Omniscript, select **Bubbles** and **Compose**.

5. To pass contextual data, add an input parameter.

- a. In Key, enter a variable name.
- b. In Value, enter the variable value.

Avoid using [] or <> brackets in value or user input fields. These characters can cause issues when sending pubsub events between Flexcards, as they are interpreted as merge fields, leading to data corruption.

The Value field supports concatenated strings including a combination of plain text and supported merge fields, such as `{Parent.type} > {Name}`, and the `{Session.var}` context variable.

Flexcards Reserved Event and Channel Names

Review the event names and channel names that are reserved for specific purposes and use cases.

Example Use Cases for Flexcard Events

Review examples of Flexcard event setup for common use cases such as updating a product list, registering dynamic custom events, or resetting a user selection.

Flexcards Reserved Event and Channel Names

Review the event names and channel names that are reserved for specific purposes and use cases.

Only use the reserved names in this table for their intended purpose.

Reserved Name	Event Type	Property Name Used In	Description
close_modal	Pubsub Event	Channel Name	Close a flyout window automatically when its channel name is <code>close_modal</code> .
closemodal	Custom Event	Event Name	Close a flyout window from an action.
close	Pubsub Event	Event Name	Close a flyout window automatically in combination when its channel name is <code>close_modal</code> .
resetselectedcards	Custom Event	Event Name	Reset all selected child cards across multiple parent cards.
selectcards_	Custom Event	Event Name	In a custom event listener, update an Omniscript with data from cards that the user selects.

Because Omnistudio uses these reserved names for specific events, don't use them when setting up events:

- executeaction
- fireactionevent
- reload
- remove
- setvalue
- showtoast
- updatedatasource
- updatefieldbinding

- updateos
- updateparent
- updatestyle

Example Use Cases for Flexcard Events

Review examples of Flexcard event setup for common use cases such as updating a product list, registering dynamic custom events, or resetting a user selection.

Update Product List

Create a Lightning page that shows a filterable list of products.

Register Dynamic Custom Event

Create an event on a Flexcard by using a merge field called `channel_{Session.name}`, `case{recordId}`, or `evnt_{Parent.status}`. Then register the specific event from another component. For example, register `channel_acme`, `case143437h8f9f00`, or `evnt_New`.

Select One Child Flexcard Across Multiple Parents

Let users select new values by resetting previously selected records. Configure the `resetselectedcards` event to reset the selection on the child card, or on the child card and its parent Flexcards.

Pass Data from a Flexcard to an Omniscript in a Flyout

Passing data from a Flexcard to an Omniscript in a flyout involves setting up a custom data source, triggering the Omniscript with an action element, and configuring the event listener to pass the data.

Reload a Flexcard After Updating an Omniscript in a Flyout

To update information on a Flexcard after the user interacts with an Omniscript embedded in a flyout:

Update Product List

Create a Lightning page that shows a filterable list of products.

1. Create a Flexcard that shows filters, and has an Apply button that, when clicked, triggers an event.
2. Create another Flexcard that shows product information, listens for the first Flexcard's event, and executes an action.

When a user selects a filter such as <\$2000, and clicks **Apply**, the first Flexcard sends an event that triggers an action on the second Flexcard. Then, the second Flexcard updates the list of displayed products.

Register Dynamic Custom Event

Create an event on a Flexcard by using a merge field called `channel_{Session.name}`, `case{recordId}`, or `evnt_{Parent.status}`. Then register the specific event from another component. For example, register `channel_acme`, `case143437h8f9f00`, or `evnt_New`.

Select One Child Flexcard Across Multiple Parents

Let users select new values by resetting previously selected records. Configure the `resetselectedcards` event to reset the selection on the child card, or on the child card and its parent Flexcards.

Let's illustrate this with an example where users schedule a healthcare appointment with a provider of their choice. The healthcare website uses an Omniscript that contains a Flexcard, which in turn contains a child Flexcard. When users select a time slot for their appointment, any previous selection must be reset. Only one selection must be active at any given time. To achieve this:

- Create a parent Flexcard that lists all available providers. Embed a child Flexcard that lists the provider's available appointments.
- In the child Flexcard, add an actionable element that executes sequential actions that triggers a pubsub event followed by a custom event. From the pubsub event listener, execute the custom event that resets all selected appointments. From the custom event listener, execute sequential actions that first selects a new appointment time, then updates an Omniscript with the selected data.

When a user selects an appointment time using a child card, all other selected times are reset across all providers.

1. Create a child Flexcard.
2. To give users a way to select the child card, add an actionable element such as Block or Action.
3. Create an action to trigger the event that deselects all previously selected child cards. In the appointment example, let's clear all selected appointments across all healthcare providers. Perform these tasks:
 - a. In the Actions section of the Properties panel, select **Event** as the Action Type and **Pubsub** as the Event Type.
 - b. Enter a Channel Name, such as *makeappointment*.
 - c. Enter an Event Name, such as *clearprevappt*.
4. In the Setup panel, create the event listener for this event:
 - a. Enter the same values for Event Type, Channel Name, and Event Name from the previous steps.
 - b. Under Actions, create a custom event action whose Event Name is `resetselectedcards`.
 - c. To reset child cards across all parents, select **Bubbles**, and confirm **Composed** isn't selected. However, if your parent Flexcard has selectable elements, and you want users to reset selected items on both parent and child cards, select **Composed** as well as **Bubbles**. For example, if a user must select the doctor before selecting the appointment, both the doctor and the appointment are reset when the user selects a new appointment.
 - d. Click **Save**.
5. On the same actionable element from step 2, create an action to select a new child card.
 - a. In the Actions section of the Properties panel, create a sequential action.
 - b. Select **Event** as the Action Type and **Custom** as the Event Type.
 - c. Enter the Event Name such as *selectappt*.
 - d. Select **Bubbles**. If you selected **Composed** in step 4 c, select it now. Else, make sure it remains deselected.
6. In the Setup panel, create an event listener for the custom event.

- a. Enter *Custom* as the Event Type, and enter the Event Name from your custom event action.
 - b. Under Actions, select **Card** as the Action Type and **Select Cards** as the Type.
 - c. In Selected Cards List Name, enter a name for the array that holds the selected card data, such as *appointment*.
 - d. Configure other properties for the Select Cards action.
 - e. To update an Omniscript with the selected card's data, add a sequential action.
 - f. Select **Update Omniscript** as the Action Type.
 - g. In the Input Parameters section, click + **Add New**.
 - h. In the Key field, enter a name of the data node to pass to the Omniscript. For example: `.selectedappointment`
 - i. In Value, enter the name entered in Selected Cards List Name as a merge field. For example: `{appointment}`
 - j. Click **Save**.
7. Activate the Flexcard.
 8. In the parent Flexcard, and drag a Flexcard element.
 9. For Flexcard Name, select the child Flexcard created in step 1.
 10. To preview and test your Flexcard before publishing, click the **Preview** tab. Select a child card, such as an appointment time. Then select another one. When you select a new child card, any previously selected child card on any parent card is unselected. If Composed is selected with Bubbles, if there are any selectable items on the parent Flexcard, those are also reset when a child is selected.
 11. In the Setup panel, select **Omniscript Support**.
 12. Before embedding your Flexcard as a custom LWC into an Omniscript, activate your Flexcard.

When a user selects an appointment time on the child Flexcard, Omnistudio resets all other selected times across all providers, which are on its parent Flexcards.

Pass Data from a Flexcard to an Omniscript in a Flyout

Passing data from a Flexcard to an Omniscript in a flyout involves setting up a custom data source, triggering the Omniscript with an action element, and configuring the event listener to pass the data.

1. Create an Omniscript.
See [Create an Omniscript](#).
2. Create a Flexcard with a custom data source and add the custom JSON `{test}`.
Pass this `{test}` data to an Omniscript in subsequent steps.
3. Drag an action element to the Flexcard canvas that triggers the action to launch the Omniscript.
 - a. Display the action element as a button.
When the user clicks this action button, it launches the Omniscript in a flyout.
 - b. Add a pubsub action, `dataactionevent`, to the action element.
 - c. Pass an input parameter in the event action by setting the key as `obj` and value as `{test}`.
4. From the Flexcard setup, configure the event listener of the Flexcard with these values.
 - **Event Type:** `Pubsub`
 - **Event Name:** `dataactionevent`
 - **Channel Name:** `dataactionevent`
 - **Action Type:** `Flyout`

- **Flyout Type:** *Omniscripts*
 - **Flyout:** Select the Omniscript that you created in Step 1, which you want to launch in a flyout from the Flexcard.
 - **Event Type:** *Pubsub*
 - Pass an input parameter in the event action with key as `test` and value as `{action.obj}`.
5. In the Omniscript, add an element that accepts the `{test}` data from the Flexcard.
For example, add a text block with the merge field `%test%`. When the user clicks the action button in the Flexcard, the Omniscript appears in a flyout and the text block displays the `{test}`.
 6. Preview the Flexcard to verify that clicking the action button launches the Omniscript in a flyout and that the test data is correctly displayed in the Omniscript.

Reload a Flexcard After Updating an Omniscript in a Flyout

To update information on a Flexcard after the user interacts with an Omniscript embedded in a flyout:

1. Create the Omniscript to embed in the Flexcard flyout.
See [Create an Omniscript](#).
2. Set up the communication between the Omniscript and a Lightning web component by adding a navigate action to the Omniscript, and selecting the pub/sub messaging framework.
See [Navigate Action](#).
3. Add a flyout action on the Flexcard.
See [Set Up Flyouts on a Flexcard](#).
4. On the Setup tab of the Flexcard, add an event listener to the flyout to reload the Flexcard in response to an event with these values:

Event Type: *Pubsub*

Channel Name: *omniscript_action*

Event Name: *data*

Action Type: *Card*

Type: *Reload*

Set Up Flyouts on a Flexcard

Display additional information from a child Flexcard, an Omniscript, or a custom Lightning web component when a user clicks an action on a Flexcard. For example, show account information on a Flexcard and the primary contact's name and email address in a flyout.

1. After you select the flyout action type, select the type and name of the component to embed in the flyout. Only active components are available.

 **Important** Don't embed the same component in more than one Flexcard.

 **Note** We recommend that you avoid using the navigate action directly on a flyout or modal. However, if required, add an action of type `Event` to the Flexcard and set the event name to `closemodal` to ensure smooth functionality.

2. To pass an object from the parent Flexcard's data source instead of using the child Flexcard's data source, select a data node:
 - `{records}`: Pass all data.
 - `{records[#]}`: Pass the data of a specified record. For example, `{records[0]}` and `{records[1]}` are the first two records.
 - `{record}`: Pass the current record's data, such as when you deselect Repeat Records on the Setup tab.
 - `{record.FieldName}`: Pass a record object, for example `{record.Product}` for an object or array with additional data.
 - `{record.attributes}`: Pass all attributes for the current record.
3. To pass attributes from the flyout's component to the child component, add an attribute.
 - a. In Key, enter an attribute name such as `ContextId` for passing the context ID of an Omniscript to the Flexcard.
 - b. In Value, enter an attribute value.

To access the attribute in the child component, use the `{Parent}` context variable. For example, enter `{Parent.Id}` in the child component's data source input map value field to use the parent's account ID as the child's context ID.

The Value field supports concatenated strings, including a combination of plain text and supported merge fields, such as `{Parent.type} > {Name}`.

4. Enter a channel name.

The Channel Name field supports data merge fields such as `{Name}`, and the context variables `{recordId}` and `{Session.var}`.

5. Enter the remaining information as needed.

Close a Flyout Window Automatically

Set up a modal or popover flyout to close after a task is complete by adding a pubsub event in that window. For example, close a modal after a user updates an Omniscript from a flyout.

1. Open or create the Flexcard that is to display in the flyout.
2. Set up a pubsub event that closes the flyout window in one of these ways:
 - Set up an event action by using the action element or by adding it to a supported element such as a block.
 - Set up an event from Setup.
3. Enter the channel name that you used in the flyout's parent Flexcard.

4. Enter `close` as the event name.

Close a Flyout Window from a User Action

Enable users to close a modal flyout after they perform an action by adding a `closemodal` event in that window.

1. Open or create the Flexcard that is to display in the flyout.
2. Set up a custom event that closes the flyout window in one of these ways:
 - Set up an event action by using the `action` element or by adding it to a supported element such as a block.
 - Set up an event from Setup.
3. Enter `closemodal` as the event name.
4. Select **Composed** and **Bubbles**.

Set Up Navigation on a Flexcard

Enable navigation to an external URL, a Lightning page, a page in an Experience Builder Aura site, or any Salesforce page. For example, when a policyholder views their account policies on a Flexcard, they can click a button that opens a selected policy record page.

 **Note** Looking to add an Omniscript as your target URL? Use the [Omniscript action type](#) instead. The Navigate action type is intended for external pages.

After you select the navigate action type, enter information as needed. The target format depends on the target type:

- For standard apps, enter the app target as `standard_[appName]`.
- For custom apps, enter the app target as `[namespace]_[appName]`.
For example, if your app target name is `accountList`, enter `standard_accountList` or `vloc_accountList`, where `vloc` is the namespace.
- For components, enter the component name as `[namespace]_[componentName]`. For example, enter `vloc_accountList`, where `vloc` is the namespace and `accountList` is the name of the component.

 **Tip** To create a link that opens your users' default mail client, select the web page target type and in the URL, enter `mailto`. For example, enter `mailto:janedoe@acme.org`.

Launch an Omniscript from a Flexcard

Set up a Flexcard to call an Omniscript that updates the Flexcard's data source. For example, users enter their account ID on a Flexcard. Then, when they click the Omniscript action, Omnistudio passes the entered value and launches a form that updates their account information.

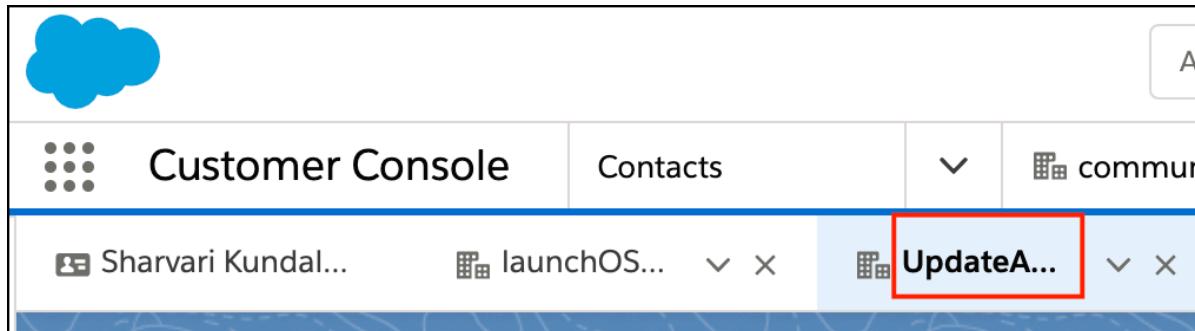
1. After you select the Omniscript action type, select the Omniscript to launch.

Tip To include a scrollbar in an Omniscript, create a model flyout in the flexcard that calls the Omniscript. No scrollbar is included when you call the Omniscript in a new page.

2. If needed, you can select the VisualForce page where you want to show the Omniscript. For example, if you have a custom template for your Omniscript, select the custom page that you created.
3. Enter the sObject's context ID associated with your Omniscript.

For example, if your Omniscript updates account information, your context ID is an account record ID variable from the data source, such as `{AccountId}`.

4. If you selected a Lightning web component Omniscript:
 - a. Enter the label for the console tab after the user launches the Omniscript.
 - b. Search for and select the icon to show next to the console tab label.



5. If needed, add these input parameters:
 - a. In Key, enter a variable name with the prefix `omniscript_`.
 - b. In Value, enter the value to pass as a string, a merge field, or a context variable, such as `New Value` or `{Name}`.
6. Enter the remaining information as needed.
7. Add the Flexcard to a Lightning page or a page in an Experience Builder Aura site. See [Activate and Publish a Flexcard](#).

[Launch an Omniscript from an Omniscript Action on a Flexcard in Experience Cloud](#)

Omnistudio supports Omniscripts launched from Flexcards in Experience Builder. With the Omniscript for Experience Builder component, you can load a dynamic Omniscript launched from a Flexcard Action element into an Experience Cloud page.

Launch an Omniscript from an Omniscript Action on a Flexcard in Experience Cloud

Omnistudio supports Omniscripts launched from Flexcards in Experience Builder. With the Omniscript for Experience Builder component, you can load a dynamic Omniscript launched from a Flexcard Action element into an Experience Cloud page.

Before you begin, add and configure the Omniscript Action element on a Flexcard. See [Launch an Omniscript from a Flexcard](#).

The Experience Cloud page created in this task serves as a dynamic placeholder and builds its URL from the type and subtype of the Omniscript in the linked Flexcard's actions. If you're working with an Aura site, the page must contain the Omniscript for Experience Cloud component, which doesn't require pre-defined values for its type and subtype fields. This page is a central repository for all such components used within the Experience Cloud site. Regardless of the number of actions utilized on the site, an Open Omniscript action directs to this page, where the Omniscript associated with the clicked action is active. If you're working with a Lightning Web Runtime (LWR) site, you don't need to add this component. Instead, you can select the **Load Omniscript from URL** checkbox.

1. Create an Experience Cloud page with the URL /lwcos and the page name lwcos.

See [Creating Custom Pages with Experience Builder](#).

2. Depending on whether you're working with an Aura or LWR site, select one of these options.

For an Aura site:	Drag an Omniscript for Experience Builder component into the /lwcos page.
For an LWR site:	Select the Load Omniscript from URL checkbox from the Omniscript Options section.

3. Add the Flexcard to the Experience Cloud page that launches the action, such as the Home page or Record Detail page.

See [Add a Flexcard to an Experience Cloud Page](#).

Update an Omniscript's JSON Code from a Flexcard

Specify the JSON node that Omnistudio updates in response to a user action on the Flexcard by embedding the Flexcard in the Omniscript as a custom Lightning web component.

1. After you select the update Omniscript action type, enter the information as needed. If you don't select a parent node, Omnistudio stores the result in a new node in the JSON code
2. To pass the data to update, add an Input Parameter.
 - a. In Key, enter the name of the field to update. For the preceding example, enter `records`.
 - b. In Value, enter the new value from the Flexcard's data source. For the preceding example, enter the `{records}` context variable to pass all asset records.
3. From Setup, turn on Omniscript support.
4. [Embed Flexcards in an Omniscript](#)

 **Suspend a Mobile Device Plan** Create a Flexcard where customers start the process to suspend their mobile device plan.

- 1. Create a Flexcard that displays the customer's available devices.
2. Add a toggle element so that they select a device.
3. To pass the value of the action (`true` or `false`) to the Omniscript, add an update Omniscript action to the toggle.
4. In Parent Node, enter `data`.

5. Add a new key/value pair for Input Parameters.
 - a. In Key, enter `records`.
 - b. In Value, enter the `{records}` context variable.
6. Create an Omniscript that walks a customer through the process of selecting the device and entering a suspension date range, before submitting the request.

Display Data on Flexcards Based on Conditions

Determine whether to apply conditions to individual Flexcard elements, to the entire Flexcard, or to both. Use states to control what's available to users at the Flexcard level. For a more granular configuration, apply conditions to specific elements, fields, events, and more. For example, to show an alert icon for escalated cases, apply conditions to that icon. To show different Flexcards based on the payment status of an insurance account, create states for each status.

Set Up Conditions to a Flexcard Element

Apply conditions to Flexcards elements to show only relevant information to your users based on data field values. For example, on a Flexcard that summarizes the terms of a contract, only show reminder text when the contract renewal date is within a defined range.

Set Up Conditions Based on Flexcard States

Flexcard states determine the fields and actions available to the user. When you create a Flexcard, Omnistudio assigns a default state. You can create more as needed to offer different information and interactions based on conditions that you define. For example, include an empty state when no data is available, or visual cues when a user action is required such as a payment or an acknowledgment. Or add different buttons based on a support case status such as Reopen for closed cases and View for escalated cases.

Example Flexcard Conditions Setup

Review examples to help you determine how to use states or conditions for your use case.

Set Up Conditions to a Flexcard Element

Apply conditions to Flexcards elements to show only relevant information to your users based on data field values. For example, on a Flexcard that summarizes the terms of a contract, only show reminder text when the contract renewal date is within a defined range.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. Select the element to update.
5. Access the conditions settings depending on the element:
 - For non-action elements, open the element's properties.
 - For actions on a menu element, add an action from the element's properties.

- For actions on an event listener, in the Setup options, add an event listener, and then select the action.
6. Under Conditions, add a condition.
7. For best performance, deselect **Preload Conditional Component**.

By default, Omnistudio preloads all conditional elements on a Flexcard and adds them to the Document Object Model (DOM) or the rendered HTML. If the condition isn't met, Omnistudio hides them with the CSS but still renders the element's HTML. Preloading all elements and hiding those elements can add unnecessary wait time for the user to complete their task.

For the block element, the **Preload Conditional Component** is enabled by default to ensure that all elements within the block load correctly.

8. Select a data field to evaluate, an operator, and a value to check against.
 - a. To check against an empty value, enter `undefined` as the value or leave it empty.
 - b. You can't use a variable as the value, except for the `Parent.attr` context variable for a state element in a child Flexcard.
9. If needed, add more conditions and operators between the conditions.
10. If needed, build nested conditions by adding groups of conditions.

A group is similar to enclosing a series of tests in parentheses in a programming language. For example, you want to add a condition for high priority cases that are either escalated or have an account whose revenue is above 500000. To do so, create two condition groups connected by an `OR` operator as `[Priority = High AND IsEscalated = true] OR [Priority = High AND AnnualRevenue > 500000]`.

11. Save your changes.

The Flexcard applies the condition in preview or at run time.



Turn Off Preloading Conditional Components You have a list of cases that your customer service reps can show or hide as needed. The Case object has a `ShowInfo` boolean field, set to false by default. Your rep manages about 100 cases. If you hide the cases by using conditions, Omnistudio loads all the cases even if your rep doesn't see them on the Flexcard, leading to potential performance issues. Instead, display the case ID, status, and creation date. Wrap all other case information in a block element and hide it with a condition that displays the block only when `ShowInfo` is true. `ShowInfo` is a boolean value that comes from your data source and is set to false by default. Disable the **Preload Conditional Component** property.

- 1. Group all other case information in a block element.
2. On the Flexcard, add the case ID, the status, and the creation date.
3. To hide the block by default, add this condition:
 - Data Field: `ShowInfo`
 - Operator: `=`
 - Value: `true`
4. Deselect **Preload Conditional Component**.

5. Add two action buttons that your rep uses to control the display of the case information:
 - Show Case Info: Add a condition so that the button is visible when ShowInfo is false. When the rep clicks the button, the Flexcard updates ShowInfo to true.
 - Hide Case Info: Add a condition so that the button is visible when ShowInfo is true. When the rep clicks the button, the Flexcard updates ShowInfo to false.
6. Deselect **Preload Conditional Component** for both actions.
7. Preview the Flexcard.

For each case, the Show Case Info button is visible, and the Case Info block is hidden. When you inspect the code in your browser, confirm that the HTML code for the Case Info block and the Hide Case Info button aren't rendered.

Set Up Conditions Based on Flexcard States

Flexcard states determine the fields and actions available to the user. When you create a Flexcard, Omnistudio assigns a default state. You can create more as needed to offer different information and interactions based on conditions that you define. For example, include an empty state when no data is available, or visual cues when a user action is required such as a payment or an acknowledgment. Or add different buttons based on a support case status such as Reopen for closed cases and View for escalated cases.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. From the elements panel, drag the **State** element to the canvas.
5. To create a state to display when there are no records:
 - Select **Use this state when the FlexCard returns a null value**.
 - If you're using the designer on a managed package, select **Blank Card State**.For example, when a data source returns no records from the Policy object, create a state with an Omniscript action to add a policy.
6. Enter an internal name for the state.
7. Add conditions and elements as needed.

When you add multiple states, Omnistudio runs through them from top to bottom, and shows the data of the first state that meets the conditions. We recommend that you organize the states from the most complex to the simplest ones. Then, at the end, add a state with no conditions, and then add a blank card state.

Example Flexcard Conditions Setup

Review examples to help you determine how to use states or conditions for your use case.

Filter Policy Information by Using States

Users access their account policy information on a Flexcard, and you want to personalize the available

actions and data depending on their account.

Let Users Hide Information in a Block

You have a Flexcard that shows the name, the ID, and the age for an account. The Flexcard also includes a block element that contains additional information such as the gender or the contact details. You want to enable users to expand or collapse the block element to show the information as needed.

Filter Policy Information by Using States

Users access their account policy information on a Flexcard, and you want to personalize the available actions and data depending on their account.

Action Button Based on Expiration Date

You want to emphasize to the policyholder when their policy is close to the expiration date. If the policy is up to date, the default state shows data fields and an action button to update account information. If the policy is about to expire, there's a red border, an alert notification, and an action button to renew the policy.

- Create states with these conditions and settings, and arrange them in this order:

1. State 1:

```
DaysToExpiration < 31  
AND  
DaysToExpiration != Null (empty)
```

2. State 2: No condition

3. State 3: Blank card state

View Based on Expiration Date and Policy Type

You want to create a view for your reps that shows universal life or universal variable life policies that are close to the expiration date.

- Create states with these conditions and nested conditions, and arrange them in this order:

1. State 1:

```
Status = Purchased  
AND  
DaysToExpiration < 31  
AND  
(PolicyType = Universal Life  
OR  
PolicyType = Universal Variable Life)
```

2. State 2: No condition

3. State 3: Blank card state

Let Users Hide Information in a Block

You have a Flexcard that shows the name, the ID, and the age for an account. The Flexcard also includes a block element that contains additional information such as the gender or the contact details. You want to enable users to expand or collapse the block element to show the information as needed.

Set a private variable in the Flexcard to define and evaluate its value at run time. First, create an action to set the value of the `Flex.varname` private variable. Then, add a condition to display the contact information based on the variable's value.

1. Configure a clickable element to show the Block element with extra case info:
 - a. Access the block element on the Flexcard.
 - b. Add an icon before the block.
 - c. Add a card action to the icon with the Set Values type.
 - d. Add a new Set Value.
 - e. In Key, enter `Flex.showinfo`.
 - f. In Value, enter `true`.
2. Configure another clickable element to hide the Block element with the extra case info:
 - a. Add an icon in the block with a different image from the icon that you added before the block.
 - b. Add a card action to the icon with the Set Values type.
 - c. Add a new Set Value.
 - d. In Key, enter `Flex.showinfo`.
 - e. In Value, enter `false`.
3. To hide the Block element by default, add a condition that checks the value of `Flex.showinfo`. Perform the following tasks:
 - a. Select the block, and in its Properties, add a new condition.
 - b. In **Data Field**, enter `Flex.showinfo`.
 - c. From the Operator picklist, select `=`.
 - d. In Value, enter `true`.

In preview, the Flexcard's JSON code includes the `showinfo` variable.

Style a Flexcard

Configure the look and feel of a Flexcard by styling individual elements, fields inside elements, and entire states. Configure backgrounds, text, and borders; adjust dimensions and element spacing; apply classes and inline styles; make your element responsive; and more.

Save the style of an element and reuse it on multiple elements. Create custom classes by using a code editor available when you build a Flexcard, and use those classes right away. Add classes to fields inside supported elements, such as the label and value of a text element. Further personalize the user experience by conditionally applying styles in response to a data field's value.

1. From the App Launcher, find and select **Flexcards**.
2. Expand a Flexcard and select a version.
3. If needed, deactivate the Flexcard for editing the Flexcard.
4. On the Style tab, select a custom style that you saved, or configure style settings as needed.
5. To save your settings for reuse, under Custom Style, click  and then select **Save As**.

Add Conditional Styles to a Flexcard Element

Show the most relevant information to your users by conditionally applying styles to a Flexcard element. Change style features such as text color, background image, and responsiveness when a condition based on the value of a data field is met. For example, add a red border around a billing Flexcard when the bill status is past due.

Configure Flexcard Element Dimensions

Set the dimensions of the Flexcard according to your company or institution's style. Set the height properties of Flexcard elements by using CSS values. Turn on responsiveness so that the width adjusts dynamically to the page's visible area. For example, expand the width of the fields for a contact's first and last names to the full page width on smaller devices. And for better user experience, limit the width to 50% on larger devices such as laptops and desktops.

Create Custom CSS Classes for a Flexcard

Expand preconfigured style options by creating custom CSS classes in a code editor when you set up your Flexcard. Omnistudio saves the CSS classes in a Salesforce attachment each time you save the Flexcard. After you activate the Flexcard, Omnistudio adds a CSS file to the generated Lightning web component package. The CSS file is available to other Flexcard elements or to supported fields inside elements.

Apply Custom CSS to a Flexcard

Apply CSS classes from style sheets such as the global Salesforce Lightning Design System (SLDS) and Newport. Or, apply classes that you create when building your Flexcard.

Overwrite Global SLDS or Newport Styles

Apply your own branding by overwriting a Flexcard's global SLDS styles. For example, if your Flexcard requires a specific appearance across multiple components, create custom SLDS styles and load them to your Flexcard as a static resource. For minor style changes on elements, we recommend that you use the settings from the Style tab.

Add Conditional Styles to a Flexcard Element

Show the most relevant information to your users by conditionally applying styles to a Flexcard element. Change style features such as text color, background image, and responsiveness when a condition based on the value of a data field is met. For example, add a red border around a billing Flexcard when the bill status is past due.

1. Select an element on the Flexcard.
2. On the Style tab, click **Add Conditional Style**.
3. Enter a conditional style name different from any other conditional style name for the same element.
4. Enter the information about the conditions to apply.
See [Set Up Conditions to a Flexcard Element](#) on how to configure conditions.

5. Save your changes.
6. Your new condition appears below the **Default**.

Omnistudio applies the conditional style and adds a Default section to the Style tab, which includes the styles applied when the defined conditions aren't met.

7. To compare the element's appearance before and after the conditions are met, turn on the toggle next to the conditional style name.

Configure Flexcard Element Dimensions

Set the dimensions of the Flexcard according to your company or institution's style. Set the height properties of Flexcard elements by using CSS values. Turn on responsiveness so that the width adjusts dynamically to the page's visible area. For example, expand the width of the fields for a contact's first and last names to the full page width on smaller devices. And for better user experience, limit the width to 50% on larger devices such as laptops and desktops.

1. Select an element on the Flexcard.
2. On the Style tab, to set a default fixed width relative to the viewport, adjust the slider or drag the border of the element on the canvas.

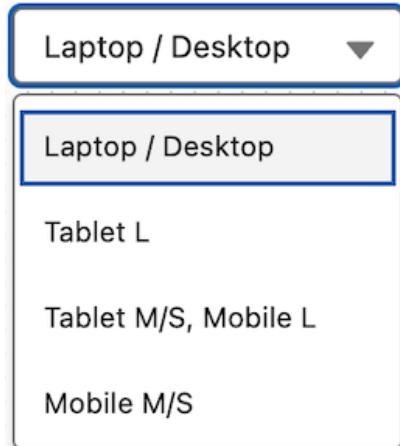
An element's width is one to 12 parts of a 12-column grid. For example, if you select 6, the element takes up 50% of the available horizontal space.

 **Note** You can adjust a state element's dimensions only when it's on a child card, and by using the slider.

3. If needed, turn on **Responsive**, and then enter the width for each viewport.

When you make the element responsive, Omnistudio adjusts the element's width to the smallest viewport breakpoint for use on mobile devices as the new default.

4. If needed, test the element's responsiveness by selecting different devices from the viewport picklist on the canvas.



 **Tip** Easily identify responsive elements by hovering over or selecting an element. Responsive

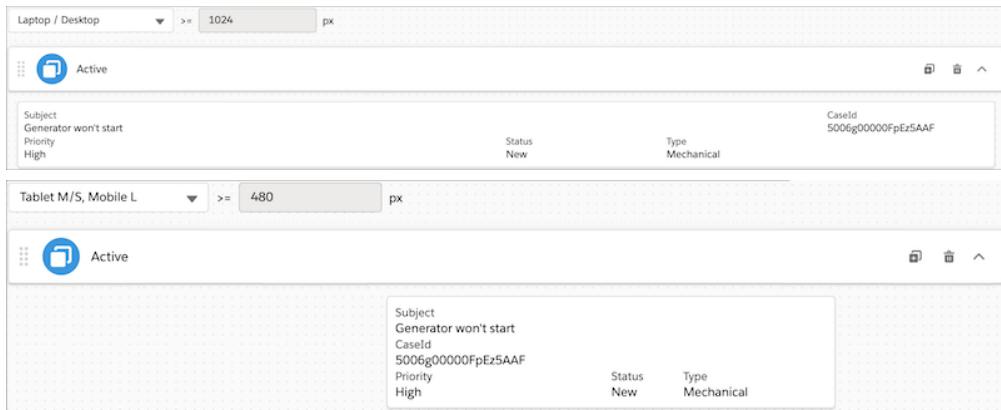
elements show  next to their names on the canvas.

-  **Responsive Subject and CaseId Field Elements** For a given account, you want to show the Subject and CaseId field elements next to each other in the medium and large viewports, and stacked in the small and smaller viewports. Set the Subject element's dimensions to these settings:

- Responsive: On
- Large and medium sliders: 10
- Small and smaller sliders: 12

Set the CaseId element's dimensions to these settings:

- Responsive: On
- Large and medium sliders: 2
- Small and smaller sliders: 12



Create Custom CSS Classes for a Flexcard

Expand preconfigured style options by creating custom CSS classes in a code editor when you set up your Flexcard. Omnistudio saves the CSS classes in a Salesforce attachment each time you save the Flexcard. After you activate the Flexcard, Omnistudio adds a CSS file to the generated Lightning web component package. The CSS file is available to other Flexcard elements or to supported fields inside elements.

-  **Warning** Use custom CSS cautiously and avoid targeting DOM elements in components you don't own. Changes to a component's internal structure might break your CSS. Salesforce might update component implementations at any time, and Salesforce Support can't assist with custom CSS issues.

1. On the Flexcard header, click , and then select **Add Custom CSS**.
2. Enter your custom classes.
3. To load these styles as a style sheet in the header of the page, turn on **Load as Global CSS**.

If you don't, Omnistudio loads the style only in the header of the Flexcard.

4. Save your changes.

Apply Custom CSS to a Flexcard

Apply CSS classes from style sheets such as the global Salesforce Lightning Design System (SLDS) and Newport. Or, apply classes that you create when building your Flexcard.

1. Select an element on the Flexcard.
2. On the **Style** tab, under Custom CSS, enter CSS class names as needed.



Note Inline CSS styles override container and element classes.

3. To apply a custom class to an element, in the Properties panel, enter a CSS class as an additional or extra class.

Omnistudio supports custom classes for icons, datatables, images, and toggles.

Grouped Data Table Element With Alternating Background Colors You have a data table that displays account cases grouped by case type.

- To alternate background colors for body rows and set a different background color for the group heading:

1. Create these custom CSS classes:

- `.alt-table-rows` for the datatable element
- `.alt-bg-group-wrapper` for the group heading

```
.alt-table-rows .tableRow:nth-child(even) {  
    background:#cccccc;  
}  
  
.alt-table-rows .tableRow:nth-child(odd) {  
    background:#ebebeb;  
}  
  
.alt-table-rows .alt-bg-group-wrapper.tableRow {  
    background-color: #000000;  
    color: #ffffff;  
}  
  
.branded-header {  
    font-variant: small-caps;  
    border-top: 1px dotted #8d1818;  
    border-bottom: 1 px dotted #8d1818;  
    padding: 1rem;
```

```
color: #999999;  
}
```

2. On the Style tab, in Custom Class, enter `alt-table-rows`.
3. In Properties, enter `alt-bg-group-wrapper` as the additional or extra class.

Overwrite Global SLDS or Newport Styles

Apply your own branding by overwriting a Flexcard's global SLDS styles. For example, if your Flexcard requires a specific appearance across multiple components, create custom SLDS styles and load them to your Flexcard as a static resource. For minor style changes on elements, we recommend that you use the settings from the Style tab.

 **Important** Loading custom Newport styles directly on the Flexcard overwrites custom global Newport styles.

1. Get global styles for your Flexcard theme:
 - SLDS: See [Lightning Design System Downloads](#).
 - Newport: See [Apply Global Branding to Omniscripts](#).
2. Customize styles as needed.
See [Lightning Apps and Components](#).
3. Upload the custom styles from a zipped file as a static resource.
See [Creating a Static Resource](#).
4. From Setup, expand the **Styling Options** section, and enter the static resource name.
See [Configure Flexcard Settings](#).

Preview and Debug a Flexcard

Preview a Flexcard's appearance and test its functionality before you publish it to a Lightning page or to a page in an Experience Builder Aura site. Review how the Flexcard looks on multiple devices and test the functionality of interactive elements such as actions. Debug the Flexcard's data output, events, actions, and more.

1. From the App Launcher, find and select **Flexcards**.
2. From the list page, select a Flexcard version.
3. Click **Preview**.



Note The Navigate and Omniscript actions don't work in preview. To view them, add them to a

Lightning or Experience Builder page.

4. If needed, add test parameters and enter this information:
 - a. In **Key**, enter a context variable that controls how the data is displayed on the Flexcard. For example, enter *pagelimit* to update how many pages the Flexcard displays for a data table element.
 - b. In **Value**, enter a test value. For example, if the key is *pagelimit*, enter *2* to show up to two pages on the Flexcard.
5. If needed, select another device from the devices picklist to test the Flexcard's responsiveness.
6. If needed, identify potential issues at run time by reviewing how the data source populates the elements on the rendered cards by inspecting the JSON code.

The values in the JSON code refresh when you interact with the Flexcard on the canvas, such as when you enter data or click an action element.

7. If needed, access the action debugger and drill down into a data source to review its logs and related events.

The action debugger helps you understand the Flexcard's client- and server-side actions and event requests and responses by providing information such as:

- The configuration, status, and responses of parent and child data sources.
- The data source type and relevant settings, such as delay time from the `config` property.
- Whether the data source successfully returns data in the `status` field, and the returned data in the `response` field.
- The response of each action or event executed on the Flexcard. For example, your Flexcard has a pubsub event that passes data. An event listener on the same Flexcard updates a data field with the passed value. View the pubsub event action and the set values action logs by clicking the Pubsub Event action link on the action debugger. Action logs include data that's relevant to the action type. For example, a pubsub event action log includes `channel`, `event`, and `inputMap`. A set values action log includes `fields` and `response`.

8. If needed, click  to copy a log.
9. When the Flexcard is ready, click  to reset the data JSON. Omnistudio doesn't reset the changes that you make on the action debugger.
10. Activate the Flexcard.

Data JSON

Use the Data JSON feature in the Flexcard Designer's Preview tab to see how your data source populates the elements on the rendered cards of your Flexcard component. Detect potential issues at run time. The Data JSON panel updates when you interact with the Flexcard, such as when entering data in an input field or clicking an actionable item.

Data JSON

Use the Data JSON feature in the Flexcard Designer's Preview tab to see how your data source populates the elements on the rendered cards of your Flexcard component. Detect potential issues at run time. The Data JSON panel updates when you interact with the Flexcard, such as when entering data in an input

field or clicking an actionable item.

The following table lists the Data JSON's available objects and the information each object provides:

Object	Description
cards	<p>Information about rendered cards based on the data displayed and the state conditions that you configure. The StateName object under each card indicates its state.</p> <p>If the Flexcard has a child Flexcard, the JSON code only shows the child Flexcard's name.</p> <p>For example, if a Flexcard displays information about an account and the data source returns three accounts with no conditions applied, then the cards object shows three card objects. Each card includes the card's state name, data fields, attributes, and child Flexcards. The uniqueKey distinguishes the card objects from each other.</p>
dataSource	<p>The array of records returned from the data source and the Flexcard setup, such as the data source type and delay time. The available settings depend on your data source type. The config object contains information about an updated data source triggered by an action. The records object contains information about the returned records for each data source.</p>
Label	<p>Custom labels fetched for the Flexcard.</p>
Params	<p>Salesforce page parameters such as the namespace.</p>
recordId	<p>ID of the Flexcard page as shown in the URL of the page when you set up the Flexcard.</p>
Session	<p>Session variables that you set on the Flexcard.</p>
TestParams	<p>Test parameters that you set on the Flexcard.</p>

Object	Description
theme	Flexcard theme.
title	Flexcard name.
User	Information about the logged-in user.

Activate and Publish a Flexcard

After you confirm that the Flexcard is ready, activate it. After activation, add the Flexcard to a Lightning page or to a page in an Experience Builder site.

If you use Omnistudio for Vlocity, or Omnistudio Standard in Summer '22 or earlier, when you activate a Flexcard, Omnistudio generates a custom Lightning web component. Add the generated LWC to a Lightning page or Experience Builder page. See [Activate, Configure, and Publish Flexcards for Omnistudio Vlocity](#).

If you use Omnistudio Standard in Summer '22 or later, add the standard Flexcard component to a Lightning page or Experience Builder page after activation.

You can't edit or delete an active Flexcard. To make changes, deactivate it first.

 **Note** It is recommended that you use a standard Flexcard component on a Lightning or Experience Cloud page. If you migrate from the Omnistudio managed package runtime to the standard runtime, you can create new versions of existing Flexcards for standard runtime. Then, you can add those Flexcards via the Process Automation section on the Lightning App Builder or Experience Builder page.

To reference your Flexcard on an Experience Cloud site, or on a Lightning page, use the Flexcard wrapper component available from the Salesforce Lightning Component Library. See [Component Library](#) for more information.

If you're referencing a Flexcard on a Lightning Web Runtime (LWR) Experience Cloud site, you must use a wrapper component that takes in the name of the Flexcard you want to load at runtime, as shown. The `flexcard-name` attribute is mandatory.

 **Example**

```
<namespace-flex-card-standard-runtime-wrapper  
    flexcard-name="name"
```

```
record-id="recordId"
object-api-name="objectApiName"
records="records"
exposed-attributes="exposedAttributes">
</namespace-flex-card-standard-runtime-wrapper>
```

Publish a Flexcard

Define metadata values, such as where your Flexcard is visible, and update the component SVG icon for your Flexcard before publishing your component to a Lightning or Experience page. For example, to view your Flexcard on an Experience site you can enable **Community Page** under **Targets** in **Publish Options**.

Add a Flexcard to a Lightning Page

Add a Flexcard that you created or an existing Flexcard to Lightning pages and set up how you want it to appear to users.

Add a Flexcard to an Experience Builder Page

Add a Flexcard that you created or an existing Flexcard to the pages of Experience Builder sites.

Considerations for Using Flexcards on Lightning Web Runtime Sites

Learn about some considerations for using Flexcards on Lightning Web Runtime (LWR) Experience Cloud sites.

Publish a Flexcard

Define metadata values, such as where your Flexcard is visible, and update the component SVG icon for your Flexcard before publishing your component to a Lightning or Experience page. For example, to view your Flexcard on an Experience site you can enable **Community Page** under **Targets** in **Publish Options**.

1. From the App Launcher, find and select **Flexcards**.
2. From the list page, select a Flexcard version.
3. If needed, activate the Flexcard.
4. Click , and then select **Publish Options**.
5. Enter information as needed. Publish the Flexcard to one of these target pages:
 - a. App Page - Enables the Flexcard to be used on an App page in Lightning App Builder.
 - b. Home Page - Enables the Flexcard to be used on a Home page in Lightning App Builder.
 - c. Record Page - Enables the Flexcard to be used on a record page in Lightning App Builder.
 - d. Community Page - Enables the Flexcard to be used as a drag-and-drop component on a page in Experience Builder.
 - e. Community Default - Enables the Flexcard in Experience Builder to expose editable properties when the component is selected.

See [XML Configuration File Elements](#).

When you select publishing options for a parent Flexcard, at least one target must be selected. If you uncheck all targets and click Save, Omnistudio automatically enables the App Page, Home Page, and

Record Page options as default targets.

6. Save your changes.

Add a Flexcard to a Lightning Page

Add a Flexcard that you created or an existing Flexcard to Lightning pages and set up how you want it to appear to users.

1. From Setup, in the Quick Find box, enter *Lightning App Builder*, and then select **Lightning App Builder**.
2. Find the page that you want to add the Flexcard to and then click **Edit**.
3. Drag the Flexcard component from the Standard section of the Components pane onto the canvas.
4. If needed, click **Flexcard Name** and then select a Flexcard.
By default, the builder uses the most recent active Flexcard.
5. To track Flexcard user interactions, select **Enable Omnistudio Analytics**.
6. If needed, configure visibility settings, for example, by using display conditions.

See [Visibility Rules on Lightning Pages](#).

7. Activate your page and configure activation properties.

See [Activate Your Lightning App Page](#).

8. Save your changes.

Add a Flexcard to an Experience Builder Page

Add a Flexcard that you created or an existing Flexcard to the pages of Experience Builder sites.

You can add a Flexcard to an Aura or Lightning Web Runtime (LWR) Experience Cloud site. To know how to create LWR-compatible Flexcards, see [Considerations for Using Flexcards on Lightning Web Runtime Sites](#).

1. From Setup, in the Quick Find box, enter Digital Experience, and then select All Sites.
2. Find the site that you add the Flexcard to and then click Builder.
3. Click . If you're using an Aura-based Experience Cloud site, find the Flexcard component under Process Automation and drag it to the canvas. If you're using an LWR site, find the Flexcard component in the Customer Interactions section.
4. From the Flexcard Name picklist, select a Flexcard.
5. If needed, in Exposed Attributes, enter a configurable public property as JSON code.

For example, to show up to five accounts on the Flexcard, if the variable is AccountRecordLimit, enter `{"AccountRecordLimit": 5}`. For multiple properties, separate key and value pairs with a comma, such as `{"AccountRecordLimit": 5, "Greeting": "Hello"}`.

6. If the Flexcard uses the record ID context variable, then to view the related record data, enter a record ID.

For information about exposed attributes and a Flexcard's context ID, see [Configure Flexcard Settings](#).

7. If needed, enter the object API name as a parameter in the `{!objectApiName}` format.

For example, for an Account record page, enter `{ !Account }`.

8. To track Flexcard user interactions, select **Enable Omnistudio Analytics**.
9. Preview your Experience Builder site, and make changes as needed.
10. Publish your Experience Builder site.

Set Up User Access to Experience Builder Pages

Make sure that your users see Flexcards on Experience Builder sites by granting them View All Records access to the Omni UI Cards object. Also, configure the necessary permission sets and groups to control Apex class access.

See [Setup Omnistudio Standard Permission Sets for Experience Site Users](#).

Considerations for Using Flexcards on Lightning Web Runtime Sites

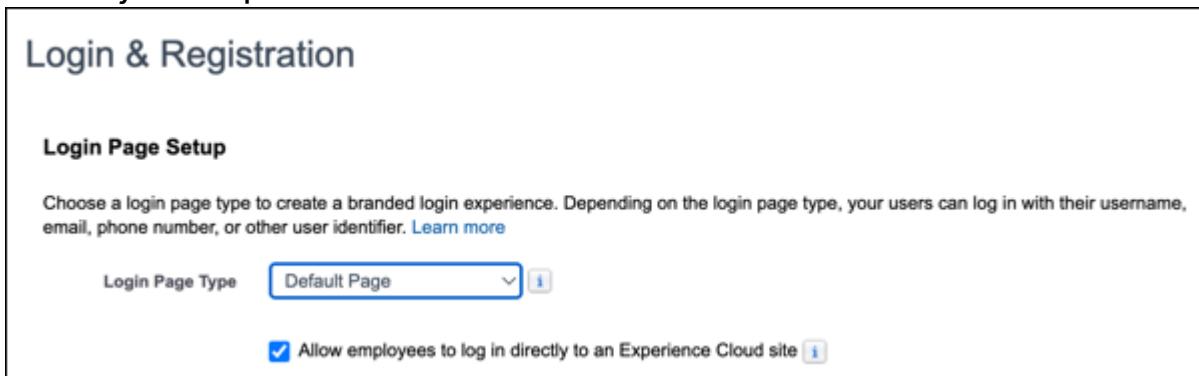
Learn about some considerations for using Flexcards on Lightning Web Runtime (LWR) Experience Cloud sites.

Both existing and newly created Flexcards can be used on LWR sites. Learn about some key considerations before you choose a Flexcard to embed in your LWR sites.

You can use Flexcards through the LWR site builder under the Customer Interactions category. To make sure that your Flexcards display correctly and function well, note these pointers and choose only compliant Flexcards for your LWR sites.

- If you make changes to your Flexcard, whether it is a child or parent Flexcard, make sure that you republish the sites that use it.
 **Note** Before republishing your sites via the site builder, consider other elements from other products that you've embedded in your site. Only republish if everything is production-ready.
- To use the Open Omniscript option on an LWR site, make sure that you select the **Load Omniscript from URL** checkbox in the Properties section of the page. This checkbox is added to the Omniscript component and loads the Omniscript present in the URL configured via the Action Type. You must also create an `lwcos` page for this action to work. For more information, see [Launch an Omniscript from an Omniscript Action on a Flexcard in Experience Cloud](#).
- The Style tab on LWR sites don't work with Flexcard elements. You can continue to control the style of the Flexcard through the options available within the Flexcard designer, but they aren't compatible with the settings native to the LWR site builder.
- Custom LWCs are supported. However, due to a technical limitation, nested custom LWCs require a workaround to work as expected. Because LWR sites work on cached data, they look for Flexcards and Omniscripts nested within your custom LWCs on the site. If present, they load the appropriate component at build time. To do this, you can pick one of the two methods outlined here.

- Create a page on your site and add all nested components in your main LWC component to this page. Hide the page so it doesn't show up for users as a navigation option.
- On the page that you're adding your custom LWC, add the dependent Flexcards and Omniscripts. Then, for each component, control the visibility via the Visibility tab. For instance, you can switch the **Show Component on Desktop** toggle on and add a condition that says *User ID* equals *Invalid_User*. Since *Invalid_User* is a value that never matches a real user ID, this condition is always false. As a result, the dependent components remain hidden from users but are still available for the LWR site to discover and cache.
- When you open an LWR site from the Digital Experiences page, the site considers you an unauthenticated user and shows only Flexcards that have data available to guest users. To make sure that you see the full data, implement a login page on your site. Then, go to the **Workspaces** link next to the site name. Click **Administration** and find the **Login & Registration** tab in the left navigation pane. Make sure that the Login Page Type dropdown has a value and that the **Allow employees to log in directly to an Experience Cloud site** checkbox is selected.



The screenshot shows the 'Login & Registration' configuration page. Under 'Login Page Setup', there is a dropdown menu labeled 'Login Page Type' with 'Default Page' selected. Below the dropdown is a checked checkbox labeled 'Allow employees to log in directly to an Experience Cloud site'. A small help icon is located next to the checkbox.

- Streaming APIs as data sources don't work on LWR.

Export and Import Flexcards

Import or export Flexcards to use them in another org, such as when you create them in a sandbox or to use sample data packs that Salesforce provides. When exporting or importing data packs for Flexcards, if you encounter Apex limit errors such as heap or CPU limits, reduce the size of the data pack by unselecting dependencies during the export process. Additionally, break the data pack into multiple smaller data packs. As a best practice, we recommend including no more than 10 elements in each data pack.

See Also

- [Extend Omnistudio Lightning Web Components](#)
- [Clone a Flexcard or Create a New Version](#)

Export a Flexcard

To export a Flexcard in the new designer on the standard runtime, use the Salesforce CLI. Omnistudio exports the file in JSON format.

Before you begin:

- [Set up Salesforce CLI](#).
- [Authorize your org](#).
- Export any related Data Mappers, followed by Integration Procedures. See [Export or Import an Omnistudio Data Mapper](#) and [Export or Import an Omnistudio Integration Procedure](#).

1. From a terminal in VS Code, enter this command, and replace the variables with the appropriate values.

```
sfdx force:data:tree:export -q "SELECT Description, AuthorName, OmniUiCardKey, OverrideKey, DataSourceConfig, SampleDataSourceResponse, ClonedFromOmniUiCardKey, VersionNumber, Namespace, Name, IsTrackingEnabled, PropertySetConfig, OmniUiCardType, Id, StylingConfiguration, UniqueName from OmniUiCard where id='flexcard_id'" -u org_alias -p -d export_directory -p
```

org_alias

The alias of the org from where you're exporting.

export_directory

The directory in the project where the exported JSON is stored.

flexcard_id

The ID of the Flexcard that you want to export.

2. Verify whether the JSON file is downloaded in the export directory.

Export a Flexcard from the Designer on a Managed Package

In the designer for a managed package, bulk export Flexcards from the Flexcards list view, or export a single Flexcard from the Flexcard designer as a data pack.

You can't use Flexcards that you export from Omnistudio in Omnistudio for Vlocity.

1. From the App Launcher, find and select Omnistudio **FlexCards**.
2. Expand a Flexcard and select the version to export.
3. If the Flexcard isn't active, activate it.
4. Click , and then select **Export**.
5. Follow the instructions and enter the information as needed.
6. To store the data pack and access it from Omni DataPacks, select **Add To Library**.
7. To store the data pack on your local machine, select **Download**.
8. Click **Done**.

Export a Flexcard Lightning Web Component from the Designer on a Managed Package

After you activate a Flexcard in the designer on a managed package, download the generated Lightning web component to inspect its code. You can't redeploy the Lightning web component back into your org.

1. From the App Launcher, find and select Omnistudio **FlexCards**.
2. Expand a Flexcard and select the version to export.
3. If the Flexcard isn't active, activate it.
4. Click , and then select **Download LWC**.
5. Click **Done**.

Import a Flexcard

To import a Flexcard in the new designer on standard runtime, use the Salesforce CLI. Omnistudio exports the file in JSON format.

Before you begin:

- Set up [Salesforce CLI](#).
- Import any related Data Mappers, followed by Integration Procedures.
See [Export or Import an Omnistudio Data Mapper](#) and [Export or Import an Omnistudio Integration Procedure](#).

 **Note** You cannot import Flexcards with an existing Flexcard name and author combination.

From a terminal in VS Code, enter this command, and replace the variables with the appropriate values.
`sfdx force:data:tree:import -p ./export_directory/OmniUiCard-plan.json -u org_alias`

org_alias

The alias of the org from where you're exporting.

export_directory

The directory in the project where the exported JSON is stored.

Import a Flexcard to the Designer on a Managed Package

Import a Flexcard as a data pack from another environment into your org.

1. From the App Launcher, find and select **Flexcards**.
2. Click **Import** and then follow the instructions.
3. If you use a Flexcard created for Omnistudio for Managed Packages, convert the objects to Omnistudio standard objects:
 - a. Activate the Flexcard.
 - b. Click , and then select **Deploy Standard Runtime Compatible LWC**.

Flexcard Sample DataPacks

Download data packs with configured Flexcard elements and settings to get you started, such as a data pack that passes data from a parent Flexcard to a child Flexcard.

-  **Note** Data packs with the Omnistudio objects can't be imported into an org with Omnistudio for Managed Packages objects. However, if you import data packs with Omnistudio for Managed Packages objects into an org with Omnistudio with standard objects and runtime, the objects are converted into Omnistudio objects.
-  **Important** Data packs are supported only for the designers on a managed package in the Omnistudio standard runtime.

[Download a Sample Custom Event DataPack](#)

Download a sample Custom Event DataPack to import into your org. Enable an action on a child Flexcard or another element to notify the parent Flexcard of an event occurring. Then execute an action based on the event.

[Download a Sample Child Flexcard DataPack](#)

Download a sample child Flexcard DataPack. Embed a reusable Flexcard as a child inside another Flexcard, passing data from the parent to the child.

[Download a Sample Datatable DataPack](#)

Download a sample Datatable DataPack to import into your org. A Datatable element lets you display a tabular structure of the data fetched from a data source on a Flexcard.

[Download a Sample Event Listener DataPack](#)

Download a sample Event Listener DataPack to import to your org. Create an event listener that listens for an event happening and performs an action in response.

[Download a Sample Data Pack that Passes Data from an Omniscript to a Flexcard](#)

Download a sample data pack that demonstrates how to pass data from an Omniscript to a Flexcard. Import the DataPack to your org.

[Download a Sample Pubsub Event DataPack](#)

Download a sample Pubsub Event DataPack to import into your org. Enable an action on a Flexcard to notify another component, such as another Flexcard on a page or application, of an event occurring.

[Download a Sample Toggle DataPack](#)

Download a sample Toggle element Flexcard DataPack into import to your org. Trigger an action when a user clicks a Toggle element on a Flexcard.

[Download a Sample Update Omniscript Action DataPack](#)

Download a sample Update Omniscript DataPack to import into your org. Create an action that updates an LWC Omniscript's data JSON from the Flexcard embedded in the Omniscript.

Download a Sample Custom Event DataPack

Download a sample Custom Event DataPack to import into your org. Enable an action on a child Flexcard or another element to notify the parent Flexcard of an event occurring. Then execute an action based on

the event.

See [Pass Data to Flexcards by Using Events](#).

The sample DataPack demonstrates how to update a data field on a parent Flexcard from an Action element in an embedded child Flexcard.

What's Included

- A child Flexcard, **demoChildFlexCardCustomEvent**, with an Action element that triggers a Custom Event, named *triggerparent*, that passes a parameter to the parent Flexcard listening for the event.
 **Note** Activate the child Flexcard to see it in the parent.
- A parent Flexcard, **demoParentFlexCardCustomEvent**, with the following features:
 - A Field element populated from an SOQL Query data source.
 - An Event Listener configured in the Setup panel listens for the *triggerparent* custom Event from the child Flexcard.

Download and Import

- Download the Omnistudio DataPack here: [demoParentChildCustomEvent-Omnistudio.json](#).
 **Note** Download the Omnistudio DataPack for an org with the Omnistudio package installed.
- To import the DataPack, see [Export and Import Flexcards](#).
 **Note** After importing, if you didn't activate the **demoChildFlexCardCustomEvent** Flexcard during the import process, from the Flexcards home tab, click the Flexcard name, select the checkbox, and click Activate. The child Flexcard must be active to view inside a parent. Click the Trigger Parent action element on the **demoParentFlexCardCustomEvent** Flexcard to see the Custom Event in action.

Download a Sample Child Flexcard DataPack

Download a sample child Flexcard DataPack. Embed a reusable Flexcard as a child inside another Flexcard, passing data from the parent to the child.

See [Embed Flexcard in a Flexcard](#).

Pass Parent Attributes to the Child Flexcard

Display Account information on the parent Flexcard and Contact information on the embedded child Flexcard by passing the parent's contextId to the child as an attribute, where Key = *AccountId*, and Value = *{recordId}*. On the child Flexcard, use *{Parent.AccountId}* as the contextId of the child Flexcard's data source.

Author	Current Version	Status	Cloned From	Is Child Card	Last Modified	Theme	Description
Vlocity	1	Inactive	-	-	25/02/2021, 02:10	Lightning	-

Desktop Add Test Params

- Name:** Edge Communications 1234

Name	Rose Gonzalez	Phone	(512) 757-6000
------	---------------	-------	----------------
- Name:** Sean Forbes

Name	Sean Forbes	Phone	(512) 757-6000
------	-------------	-------	----------------
- Name:** Burlington Textiles Corp of America

Name	Jack Rogers	Phone	(336) 222-7000
------	-------------	-------	----------------
- Name:** Pyramid Construction Inc.

Name	Pat Stumuller	Phone	(014) 427-4427
------	---------------	-------	----------------
- Name:** Dickenson plc

The **demoEmbedChildPFCpassAttribute** parent Flexcard in the DataPack includes the following features:

- A SOQL Query data source gets Account names.
- An embedded child Flexcard that passes the Account Id from the parent as an attribute named *AccountId*.

The **demoEmbedChildCFCpassAttribute** child Flexcard in the DataPack includes the following features:

- A list of Contacts based on the Account Id of the parent Flexcard it's embedded into.
- A data source that uses the `{Parent.AccountId}` context variable as the context Id to get the list of Contacts.

Pass All Parent Records to the Child Flexcard

Display Account information on the parent Flexcard and Contact information on the embedded child Flexcard by passing all records from the parent to the child by using the `{records}` merge field in the **Data Node** property.

The screenshot shows the Omnistudio DataPack interface with the following details:

- FlexCard Title:** demoEmbedChildPFCpassRecords
- Header Buttons:** Design, New Version, Clone, Activate, Help, and a dropdown menu.
- Card Headers:** Author (Product), Current Version (1), Status (Inactive), Cloned From (ApexRemoteClassDS/...), Is Child Card, Last Modified (2/26/2021, 12:53 PM), Theme (Lightning), and Description (-).
- View Selection:** Desktop (selected) and Add Test Params.
- Content Area:**
 - High Priority:**
 - Subject: This doodat not working
 - Id: 5005w00001bMnI4AAK
 - Created: 2020-07-30T20:30:50.000+0000
 - Type: Question
 - Status: Escalated
 - Medium Priority:**
 - Subject: Widget no complying
 - Id: 5005w00001bMnHuAAK
 - Created: 2020-07-30T20:30:05.000+0000
 - Type: Problem
 - Status: New

The **demoEmbedChildPFCpassRecords** parent Flexcard in the DataPack includes the following features:

- The Repeat Records feature is disabled in the Setup panel to use the parent card only as a container for the child card. See [Configure Flexcard Settings](#).
- A SOQL Query data source gets a list of an Account's Cases.
- A Flexcard element passes all parent records through the `{records}` **Data Node** to the child Flexcard.

The **demoEmbedChildFCpassRecords** child Flexcard in the DataPack includes the following features:

- A Text element displays Case information as merge fields.
- The state is 6 columns wide to display cases side-by-side.

Download and Import

- Download the Omnistudio DataPack here: [demoEmbedChildFlexCard-Omnistudio.zip](#)
- To import the DataPack, see [Export and Import Flexcards](#).

Download a Sample Datatable DataPack

Download a sample Datatable DataPack to import into your org. A Datatable element lets you display a tabular structure of the data fetched from a data source on a Flexcard.

The sample DataPack demonstrates how to configure a data table with common settings, such as searchability, sortability, and more. The data table is automatically populated with the data from the configured data source once the element is dragged onto the canvas.

	Text	Number	Currency	DateTime	Email	Checkbox	Percent	Date
1	Text 1	24.045	1994-11-05T08:15:30-05:00	test123@gmail.com	true	20		
2	Text 1	25.045	1994-11-05T08:15:30-05:00	test123@gmail.com	true		2017-02-01	
3	Text 1	26.045	1994-11-05T08:15:30-05:00	test123@gmail.com	true		2017-02-01	
4	Text 1	27.045	1994-11-05T08:15:30-05:00	test123@gmail.com	true			
5	Text 1	28.045	1994-11-05T08:15:30-05:00	test123@gmail.com	true		2017-02-01	
6	Text 2	43	-23.04	2009-12-31 23:59:59	test123@gmail.com	200	2029-02-01	
7	Text 2	43	-23.04	2009-12-31 23:59:59	test123@gmail.com	true	2029-02-01	
8	Text 2	43	-23.04	2009-12-31 23:59:59	test123@gmail.com		2029-02-01	
9	Text 2	43	-23.04	2009-12-31 23:59:59	test123@gmail.com	true	2029-02-01	
10	Text 2	43	-23.04	2009-12-31 23:59:59	test123@gmail.com	true	2029-02-01	

What's Included

A Flexcard with a Datatable element with the following features:

- Grouped by a data field.
- Searchable and sortable.
- Sortable across groups.
- Groups expanded.
- Row and cell-level editing.
- Draggable.
- Deletable row with confirmation upon deletion.
- Pagination.
- Repeat Records in Setup panel disabled.
- Custom data source.

Download and Import the DataPack

- Download the Omnistudio DataPack here: [demoDatatable-Omnistudio.json](#)
- To import the DataPack, see [Export and Import Flexcards](#).

What's Next

Preview your Flexcard. See [Preview and Debug a Flexcard](#).

Download a Sample Event Listener DataPack

Download a sample Event Listener DataPack to import to your org. Create an event listener that listens

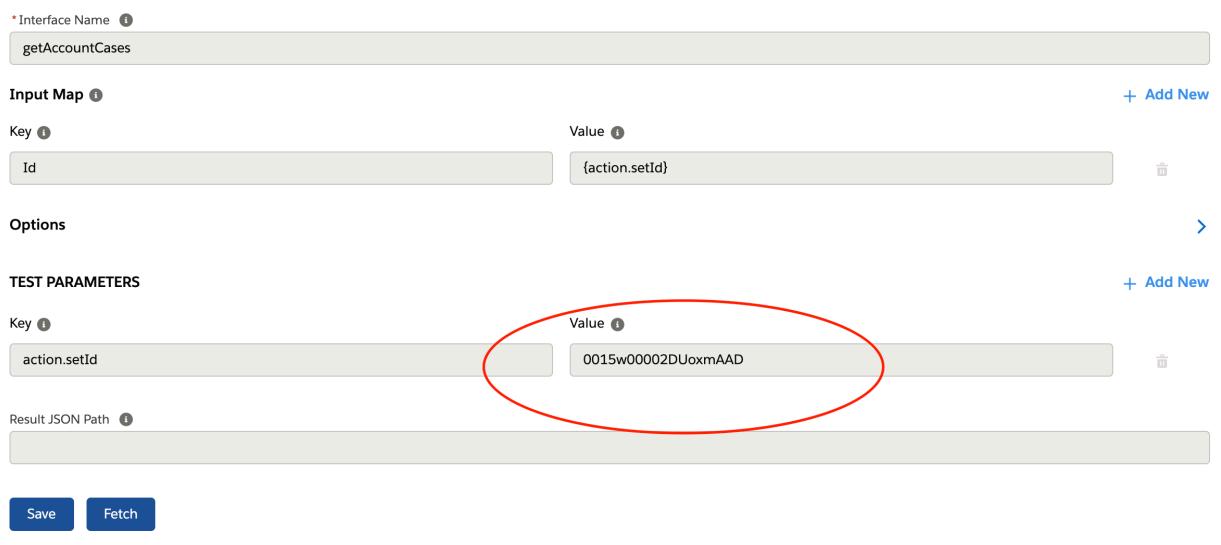
for an event happening and performs an action in response.

The sample DataPack demonstrates how to enable a Flexcard to listen for an event from another Flexcard, and to listen for an event from an action on the same Flexcard. When you click on an action element, an event is fired, and the event listeners perform other actions based on the event fired, such as reloading the Flexcard, updating the data source, and updating data field values.

What's Included

- A **demoUpdateDataSourceEvent** Flexcard with two actions that when clicked, updates the data source on another Flexcard.
- A **demoEventListener** Flexcard that listens for events and performs actions based on the event. This Flexcard has the following elements and properties:
 - Merge fields to display data from an SOQL Query data source.
 - Merge fields to display data from an Omnistudio Data Mapper Extract data source.
 - Event listeners that set values, updates the data source to an SOQL Query data source, updates the data source to a Data Mapper, and reloads the Flexcard.

 **Note** To save and fetch test data from the getAccountCases Data Mapper Extract, update the value of the action.setId in the Test Parameters of the **PubsubUpdateDS:dataraptor** event listener.



- Two action elements that when clicked, fires an event to update values with new data.
- An action element that when clicked, reloads the Flexcard.
- A **getAccountCases** Data Mapper Extract to populate merge fields when the related event is triggered.

 **Note** To return data, check that you have Account Case records in your org.

Download and Import the DataPack

- Download the *Omnistudio* DataPack here: [demoEventListener-Omnistudio.zip](#).
- Import the DataPack. See [Export and Import Flexcards](#).

After importing, if you didn't activate the Flexcards during the import process, from the Flexcards

home tab, for each Flexcard, click the Flexcard name, select the checkbox, and click **Activate**. Add both Flexcards to a Lightning Page to preview and test the event listening feature. Add the **demoEventListener** below the **demoUpdateDataSourceEvent**.

Download a Sample Data Pack that Passes Data from an Omniscript to a Flexcard

Download a sample data pack that demonstrates how to pass data from an Omniscript to a Flexcard. Import the DataPack to your org.

The sample DataPack demonstrates how to pass a recordId, a Parent object, and a set of records from the LWC Omniscript to the embedded Flexcards. After configuring and activating Flexcards and the Omniscript, view the Omniscript to see how four different Flexcards (embedded as Custom LWCs) pass data to the Omniscript.

Component Included in DataPack	Description
demoPassDataParentAttributeFC	Flexcard displays Account data, whose SOQL Query data source uses the {Parent.Id} merge field to determine the contextId..
demoPassDataRecordIdFC	Flexcard displays Account data, whose SOQL Query data source uses the {recordId} merge field to determine the contextId
demoPassDataRecordIdDRFC	Flexcard displays Account Case data, whose Omnistudio Data Mapper data source uses the {recordId} merge field to determine the contextId.
demoPassDataParentDataRecordsFC	Flexcard displays a list of accounts as a data table. The data source is a basic SOQL Query.
demoPassDataLWCOSstoFC	LWC Omniscript with the following features: <ul style="list-style-type: none">• A Set Values Action with three element values: one that defines a ContextId; one that defines a Parent object; and one that displays a list of records.• A Step with 4 Custom LWC elements that display 4 Flexcards.

1. Download the Omnistudio data pack: [demoPassDataLWCOSstoFC-Omnistudio.json](#).

2. Import the data pack.

 **Important** Do not activate yet.

3. Open the **demoPassDataParentAttributeFC** Flexcard, click **Setup**, and update the *Parent.Id* under **Test Parameters** with the Id of an Account record in your org.
4. Open the **demoPassDataRecordIdDRFC** and **demoPassDataRecordIdFC** Flexcards, click **Setup**, and update the *recordId* under **Test Parameters** with the Id of an Account record in your org.
5. Open the **demoPassDataLWCOSToFC** Omniscript, click **SetValues1**, click **Properties**, and click **Edit Properties as JSON**. Replace the values of *ContextId* and *parentObj:Id* with an Ids of Account records in your org.
6. Activate all four Flexcards.
7. To Preview the Flexcards in the LWC Omniscript, activate the LWC Omniscript, and click **Preview**.

See Also

[Embed Flexcards in an Omniscript](#)

[Export and Import Flexcards](#)

Download a Sample Pubsub Event DataPack

Download a sample Pubsub Event DataPack to import into your org. Enable an action on a Flexcard to notify another component, such as another Flexcard on a page or application, of an event occurring.

See [Pass Data to Flexcards by Using Events](#).

The sample DataPack demonstrates how to trigger events that update the values of data fields.

What's Included

- A **demoPubSubEvent** Flexcard with the following elements and features:
 - An Action element configured to trigger a Pubsub Event named *setvalue* that passes a new value for the Field element to the Flexcard.
 - An Icon element configured to trigger a Pubsub Event named *setvalue* that passes a new value for the Field element to the Flexcard.
 - An Event Listener configured in the Setup panel that listens for the *setvalue* event and performs an action that updates the Field value based on parameters passed from the event.
-  **Note** The **Record Index** is set to *1* so that only the Field value in the second record returned from the custom data source is updated.

Download and Import the DataPack

- Download the Omnistudio DataPack here: [demoPubsubEvent-Omnistudio.json](#).
- To import the DataPack, see [Export and Import Flexcards](#).

After importing, from the Flexcards home tab, click the **demoPubSubEvent** to open the Flexcard Designer, and click **Preview**. From here you can click actions to test events.

Download a Sample Toggle DataPack

Download a sample Toggle element Flexcard DataPack into import to your org. Trigger an action when a user clicks a Toggle element on a Flexcard.

The sample DataPack demonstrates how the different toggle types look, and how to update data fields with new values passed from the elements.

The screenshot shows the 'demoToggleDisplayActionUpdate' Flexcard in the Omnistudio interface. At the top, there's a navigation bar with tabs like 'Design', 'New Version', 'Clone', 'Activate', and 'Help'. Below the header, there are details about the card: Author 'CardsInsDaily', Current Version '1', Status 'Inactive', Cloned From 'demoToggleElementTrig...', Is Child Card 'false', Last Modified '11/12/2020, 11:31 AM', Theme 'Lightning', and Description '-'. A dropdown menu shows 'Desktop' and an 'Add Test Params' button. The main content area is titled 'Custom Data Source Fields + Values' and shows a table with one row: 'Active' (Value: true) and 'Value' (Value: One). Below this is a section titled 'Toggle Elements' with two columns. The left column, under 'These Toggle elements will update the data source with new values', shows a 'Checkbox Group' with options 'One' (checked), 'Two', and 'Three', and a 'Toggle' switch set to 'ON'. The right column, under 'These toggle elements do not update the data source', shows a 'Checkbox Button' with a red square icon and a 'Following' button with a checkmark. At the bottom, it says 'This Toggle element has an action that updates the "Value" field above.' and shows a 'Checkbox Group Button' with three tabs: 'One' (selected), 'Two', and 'Three'.

Included in the DataPack is the **demoToggleDisplayActionUpdate** Flexcard, which has the following features:

- Displays all Toggle Types.
- Enables and configures the Checkbox Group, Checkbox Button, and Checkbox Toggle elements to update the data source with new values.
- A Checkbox Group Button Toggle element with a Card action that updates a Field value. See [Set Values](#).

1. Download the Omnistudio DataPack here: [demoToggleDisplayActionUpdate-Omnistudio.json](#).
2. To import the DataPack, see [Export and Import Flexcards](#).

Download a Sample Update Omniscript Action DataPack

Download a sample Update Omniscript DataPack to import into your org. Create an action that updates an LWC Omniscript's data JSON from the Flexcard embedded in the Omniscript.

See [Update an Omniscript's JSON Code from a Flexcard](#).

The sample DataPack demonstrates how to prefill data in an Omniscript when a user clicks an Update Omniscript action from a Flexcard embedded in the Omniscript. The Flexcard action element passes data to the Omniscript.

Included in the DataPack are the following components:

- A **demoPrefillUpdateOSActionFlexCard** Flexcard with an Update Omniscript action that passes data source values for *Id* and *Name* to an Omniscript in a *data* node.
 - A **demoPrefillUpdateOmniScriptOS** LWC Omniscript with the embedded Flexcard and a text block with merge fields to display the incoming data.
1. Download the Omnistudio data pack here: [demoPrefillUpdateOmniScriptAction-Omnistudio.zip](#).
 2. Import the Omniscript. See [Export and Import Omniscripts](#).
 3. Import the Flexcard. See [Export and Import Flexcards](#).
 4. If you didn't activate the Flexcard or Omniscript during the import process:
 - a. From the Flexcards home tab, click the **demoPrefillUpdateOSActionFlexCard** Flexcard, select the checkbox, and click **Activate**.
 - b. From the Omniscripts home tab, click the **demoPrefillUpdateOmniScriptOS** Omniscript to open the LWC Omniscript Designer, and click **Activate**.
 5. Click **Preview** in the Omniscript Designer.

Flexcards Context Variables

To provide context to data sources, actions, and other components, add global and local context variables to a Flexcard. All variables are case-sensitive.

Global Variables

Name	Description	Merge Field
Label	Salesforce Custom Labels.	{Label.mylabel} For example, {Label.AccountName} gets a custom label named AccountName.

Name	Description	Merge Field
objectApiName	<p>Beginning Winter '22, pass the object API Name to a component as an attribute, such as a Custom LWC, when a Flexcard is on a record page. Similar to <code>{recordId}</code>, use Test Parameters to verify in Preview. See Test Data Source Settings on a Flexcard.</p> <p>Supported property fields:</p> <ul style="list-style-type: none"> • Flyout action: Pass as an Attribute to a child Flexcard, Custom LWC, and Omniscript • Custom LWC and child Flexcard elements: Pass as an Attribute • Block element: Show as a Label when Collapsible is enabled • Field element: Show as the Output • Text element: Show as plain text 	<code>{objectApiName}</code> For example, use an SOQL Query to get a list of accounts. Enter <code>{objectApiName}</code> as the object to query. Then add a Test Parameter whose Key is <code>ObjectApiName</code> and Value is <code>Account</code> .
Params	The page parameters passed to the URL.	<code>{Params.fieldName}</code> For example, <code>{Params.Id}</code> gets the context Id. We recommend using <code>{recordId}</code> to get the context Id of a record.
Parent	Reference attributes from the parent Flexcard on the child Flexcard with this variable. Attributes are defined at design time on the parent Flexcard. See Embed Flexcard in a Flexcard .	<code>{Parent.attributeName}</code> For example, <code>Parent.Id</code> gets the <code>{Id}</code> attribute defined on the parent Flexcard.

Name	Description	Merge Field
recordId	Gets the context Id of a Salesforce object's data record. Use Test Parameters to verify in Preview. See Test Data Source Settings on a Flexcard .	{recordId} For example, if using an Omnistudio Data Mapper to get a list of Account Cases, in the Input Map , enter <i>AccountId</i> as Key and {recordId} as the Value . Then add a Test Parameter whose Key is <i>recordId</i> and Value is <i>138238279r9ff</i> to populate your Flexcard with actual data.
records	The records object of the Flexcard, which can be passed on to Child Flexcard, Custom LWC, and Flyout parameters.	{records} For example, {records} gets all records, while {records [0]} gets the first available record.
Session	Variables used to store values from data sources and external systems, and hardcoded values. Properties are defined at design time on the Flexcard. See Configure Flexcard Settings .	{Session.sessionkey} For example, {Session.pageLimit} gets a session variable named <i>pageLimit</i> .
User	User and Contact properties for the logged-in user. 🕒 Note Rendered User context variable is viewable only at run time. View in Preview. Available properties: <ul style="list-style-type: none">• <code>userId</code>: User's Salesforce Id.• <code>userAnLocale</code>: User's personal locale.• <code>userSfLocale</code>: Salesforce	{User.property} For example, {User.userName} gets the logged in user's username.

Name	Description	Merge Field
	<p>org's locale.</p> <ul style="list-style-type: none"> • <code>userCurrency</code> : User's default currency. • <code>userLanguage</code> : User's default language. • <code>userTimeZone</code> : User's timezone. • <code>userName</code> : User's username. • <code>userType</code> : User's license type. • <code>userRole</code> : User's role. • <code>userProfileName</code> : Profile name from the Profile lookup. • <code>userProfileId</code> : Salesforce Id from the Profile lookup. • <code>userAccountId</code> : Salesforce id from the Account lookup. • <code>userContactId</code> : Salesforce id from the Contact lookup. 	

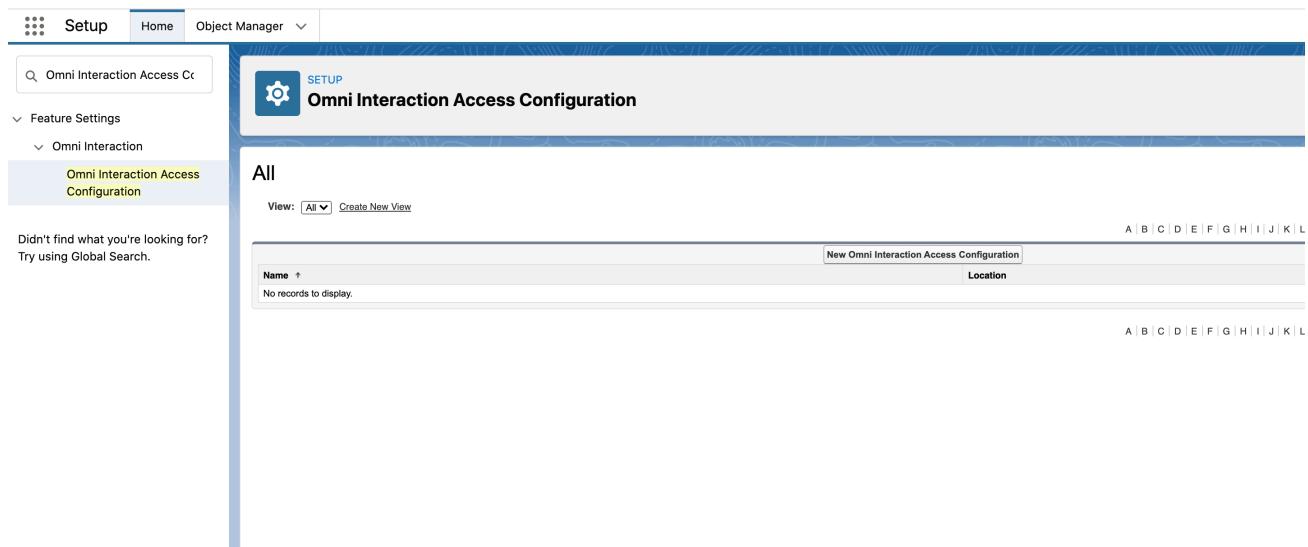
Local Variables

Name	Description	Merge Field
action	<p>The parameters received by an event listener and sent to an action.</p>	<p><code>{action.parameter}</code></p> <p>For example, if your event is sending you a <code>contextId</code> parameter, then use <code>{action.contextId}</code>.</p> <p>To send parameters to an action from a data table, use the <code>{action.result.X}</code> format, where X represents any field name defined within the data source.</p>

Name	Description	Merge Field
element	<p>The values set by the toggle element and tied to an action.</p> <ul style="list-style-type: none"> • Checkbox Toggle type: <code>element.checked</code> • All other Toggle types: <code>element.value</code> 	<code>{element.value}, {element.checked}</code>
record	<p>The record object that belongs to the State, and passed on to Child Flexcard, Custom LWC, and Flyout parameters.</p>	<code>{record}, {record.FieldName}</code> For example, <code>{record}</code> gets the entire record, while <code>{record.Name}</code> gets the Name field from the record.

Flexcards Omni Interaction Access Configuration

In Omnistudio, limit access to data sources and update cache settings for the Omni UI Card object (Flexcard) across your org. Enable and disable data source types for an org, profile, or user level. Disable caching on the Platform Cache for all Flexcards. Enable Flexcards to cache asynchronously without waiting for the full caching process to finish.



Disable Platform Cache for Flexcards in Omnistudio

To get the latest available data from the database, disable caching for all Flexcards in Omnistudio.

Flexcards, custom labels, and profile data load from the **VlocityMetadata** partition of the Platform Cache when the card is active and viewed from a record page or a community page by the end user.

[Enable Async Card Caching for Flexcards in Omnistudio](#)

Use the Future Method to enable card caching to run in sync with the Flexcard UI in Omnistudio. Future method calls are async calls made by an Apex class.

[Disable User Information Caching for Flexcards in Omnistudio](#)

Disable user information caching in the session cache for Flexcards in Omnistudio. Flexcards keep stored user profile data on the **VlocityMetadata** partition of the Platform Cache. The `Cards.UserTrigger` caches the user after their information updates, and removes the cached user after the user is disabled or deleted.

[Manage Flexcard Data Source Org Access in Omnistudio](#)

Enable or disable specific Flexcard data sources in Omnistudio for your entire org. By default, all data source types are enabled for an organization.

[Manage Flexcard Data Source User and Profile Access in Omnistudio](#)

Enable or disable specific Flexcard data sources for an Omnistudio profile or user. For example, prevent admins with limited permissions from using more complex data sources like an Apex REST or Apex Remote.

Disable Platform Cache for Flexcards in Omnistudio

To get the latest available data from the database, disable caching for all Flexcards in Omnistudio.

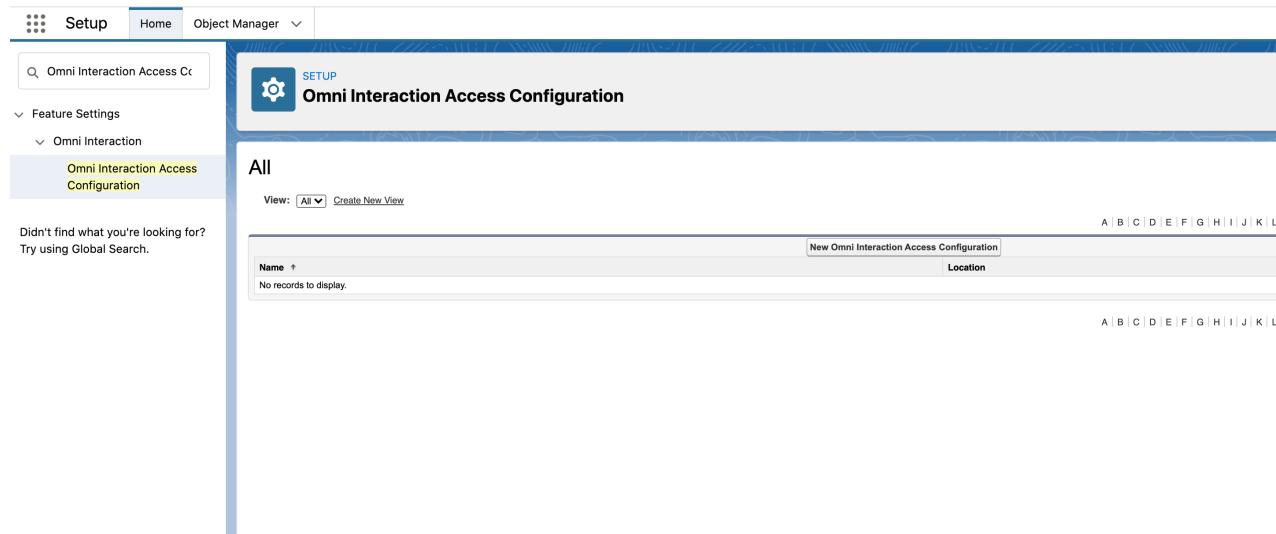
Flexcards, custom labels, and profile data load from the **VlocityMetadata** partition of the Platform Cache when the card is active and viewed from a record page or a community page by the end user.

To confirm your org has enough space allocated to the **VlocityMetadata** partition or to allocate more space, see [Allocating Space in the Platform Cache Partitions](#).

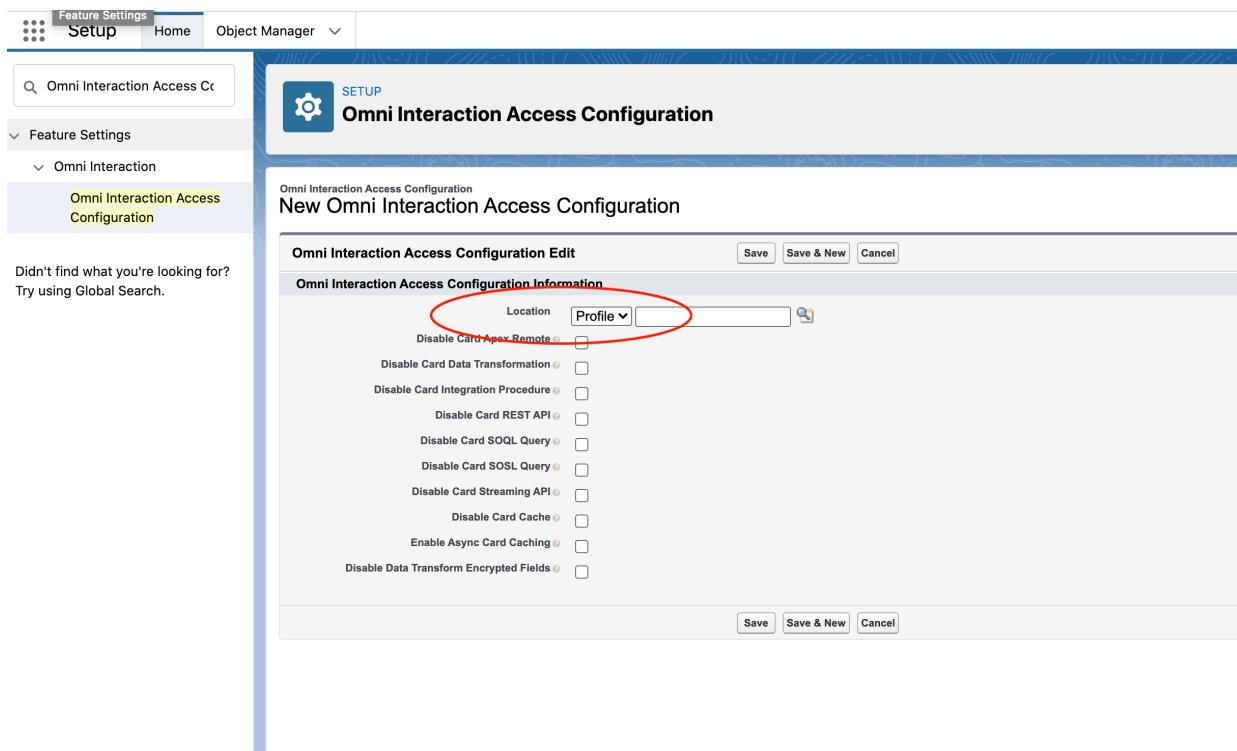
Required Versions

Beginning Winter '22.

1. Go to Setup, and search for and select **Omni Interaction Access Configuration**.



2. To disable cache across all users and profiles:
 - a. If an **Org_Wide** record exists, click **Edit** for **Org_Wide**, check **Disable Card Cache**.
 - b. If an **Org_Wide** record doesn't exist, click **New Omni Interaction Access Configuration** to create a record, and check **Disable Card Cache**.
 - c. Click **Save**.
3. To disable caching for a specific user or profile that already exists, perform the following tasks:
 - a. Click **Edit** for that user or profile, such as *Profile_00eR0073900ZryF* or *User_005R0963000bbZP*.
 - b. Check **Disable Card Cache**.
 - c. Click **Save**.
4. To disable caching for a new user or profile, perform the following tasks:
 - a. Click **New Omni Interaction Access Configuration** to create a record.
 - b. Select **Profile** or **User** from the **Location** list.



- Select **Profile** to manage data sources for a specific profile, such as *System Administrator*.
- Select **User** to manage data sources for a specific user, such as *Jane Doe*.

c. Check **Disable Card Cache**.

d. Click **Save**.

Enable Async Card Caching for Flexcards in Omnistudio

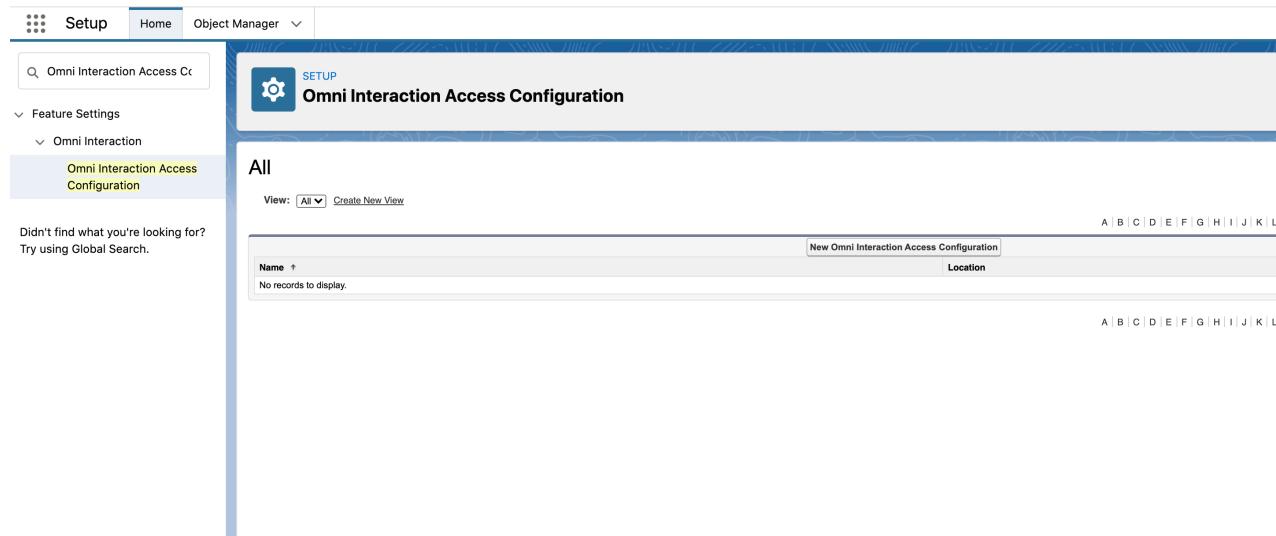
Use the Future Method to enable card caching to run in sync with the Flexcard UI in Omnistudio. Future method calls are async calls made by an Apex class.

See [Future Methods](#).

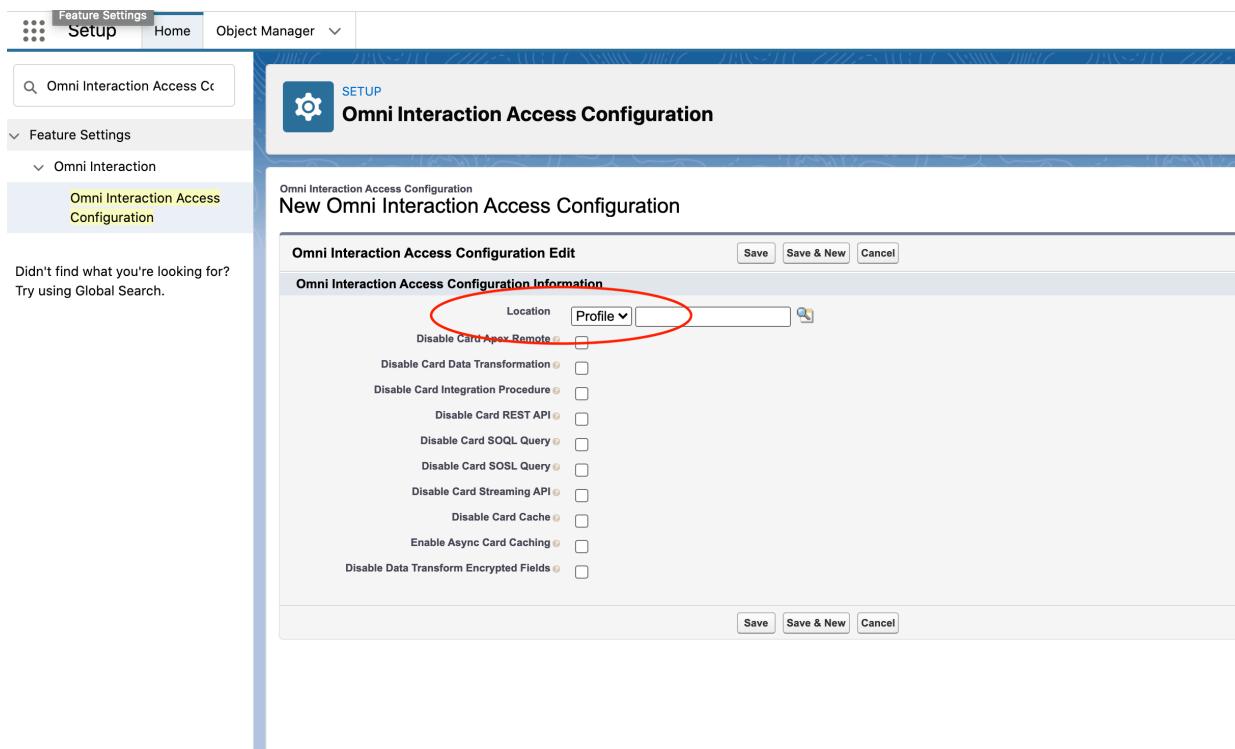
Required Versions

Beginning Winter '22.

1. Go to Setup, and search for and select **Omni Interaction Access Configuration**.



2. To enable async card caching org wide:
 - a. If an **Org_Wide** record exists, click **Edit** for **Org_Wide**, check **Enable Async Card Caching**.
 - b. If an **Org_Wide** record doesn't exist, click **New Omni Interaction Access Configuration** to create a record, and check **Enable Async Card Caching**.
 - c. Click **Save**.
3. To enable async card caching for a specific profile or user, perform the following tasks:
 - a. Click **Edit** for that user or profile, such as *Profile_00eR0073900ZryF* or *User_005R0963000bbZP*.
 - b. Check **Enable Async Card Caching**.
 - c. Click **Save**.
4. To enable async card caching for a new user or profile, perform the following tasks:
 - a. Click **New Omni Interaction Access Configuration** to create a record.
 - b. Select **Profile** or **User** from the **Location** list.



- Select **Profile** to manage data sources for a specific profile, such as *System Administrator*.
- Select **User** to manage data sources for a specific user, such as *Jane Doe*.

c. Check **Enable Async Card Caching**.

d. Click **Save**.

Disable User Information Caching for Flexcards in Omnistudio

Disable user information caching in the session cache for Flexcards in Omnistudio. Flexcards keep stored user profile data on the **VlocityMetadata** partition of the Platform Cache. The **Cards.UserTrigger** caches the user after their information updates, and removes the cached user after the user is disabled or deleted.

Required Versions

Beginning Winter '22.

1. Go to Setup, and search for and select **Omni Interaction Configuration**.

The screenshot shows the Salesforce Setup interface with the 'Omni Interaction Configuration' page selected. The left sidebar lists various setup categories like Analytics, Chatter, Commerce, Data.com, etc. The main content area displays a table of configuration items with columns for Action, Name, and Value. One row is highlighted with a blue background, showing 'Cards_UserTrigger' under 'Name' and 'true' under 'Value'. A 'New Omni Interaction Configuration' button is visible at the top right of the table.

Action	Name	Value
Edit Del	ApexClassCheck	true
Edit Del	Cards_UserTrigger	true
Edit Del	CheckCachedMetadataRecordSecurity	true
Edit Del	DocuSignAccountId	65530a0c-03ad-4bcc-a511-714e7d2f4a3f
Edit Del	DocuSignNameCredential	DocuSign
Edit Del	ErrorLoggingEnabled	true
Edit Del	IPDebugLevel	INFO
Edit Del	LoggingEnabled	true
Edit Del	newportZipUrl	/resource/1617714837000/newportcustom
Edit Del	OmniAnalyticsEnabled	true
Edit Del	OmniAnalyticsTrackingDebug	false
Edit Del	RollbackDRChanges	false
Edit Del	ShowLegacyOmnistudioUI	false
Edit Del	testOmniInteractionConfiguration	test config
Edit Del	TheFirstInstalledOmniPackage	vvelocity_lvc11
Edit Del	Track_InProc	true
Edit Del	Track_OmniScript	true

2. Click **New Omni Interaction Configuration** and perform the following tasks:

- In **Label**, enter **Cards.UserTrigger**.
- Name** populates automatically. Don't edit.
- In **Value**, enter **true**.
- Click **Save**.

The screenshot shows the 'Omni Interaction Configuration Edit' page. It has a header with 'Omni Interaction Configuration' and 'New Omni Interaction Configuration'. Below is a form titled 'Omni Interaction Configuration Information' with fields for Label, Name, and Value. The 'Label' field contains 'Cards.UserTrigger', 'Name' contains 'Cards_UserTrigger', and 'Value' contains 'true'. At the bottom are 'Save', 'Save & New', and 'Cancel' buttons.

3. View **Cards_UserTrigger** listed on the Omni Interaction Configuration page.

Action	Name	Value
Edit Del	ApexClassCheck	true
Edit Del	Cards_UserTrigger	true
Edit Del	CheckCachedMetadataRecordSecurity	true
Edit Del	DocuSignAccountId	65530a0c-03ad-4bcc-a511-714e7d2f1a3f
Edit Del	DocuSignNamedCredential	DocuSign
Edit Del	ErrorLoggingEnabled	true
Edit Del	IPDebugLevel	INFO
Edit Del	LoggingEnabled	true
Edit Del	newport2ZipUrl	/resource/1617714837000/newportcustom
Edit Del	OmniAnalyticsEnabled	true
Edit Del	OmniAnalyticsTrackingDebug	false
Edit Del	RollbackDRCChanges	false
Edit Del	ShowLegacyOmnistudioUi	false
Edit Del	testOmnilnteractionConfiguration	test config
Edit Del	TheFirstInstalledOmniPackage	vlocity_lwc11
Edit Del	Track_IntProc	true
Edit Del	Track_OmniScript	true

Manage Flexcard Data Source Org Access in Omnistudio

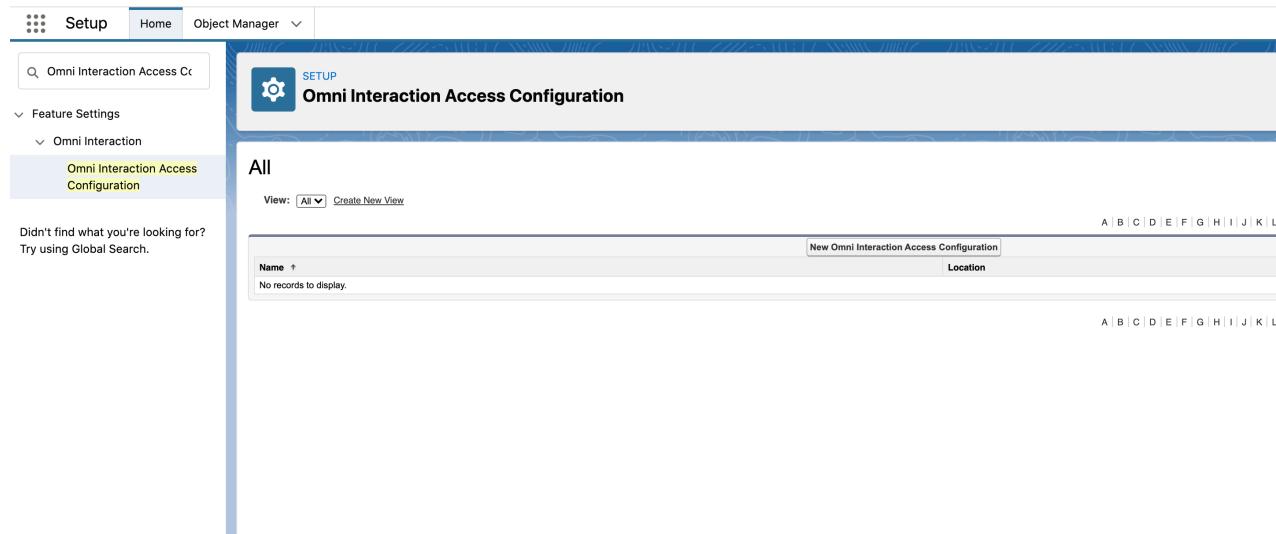
Enable or disable specific Flexcard data sources in Omnistudio for your entire org. By default, all data source types are enabled for an organization.

To enable or disable access to data sources for a specific user or profile, see [Manage Flexcard Data Source User and Profile Access in Omnistudio](#).

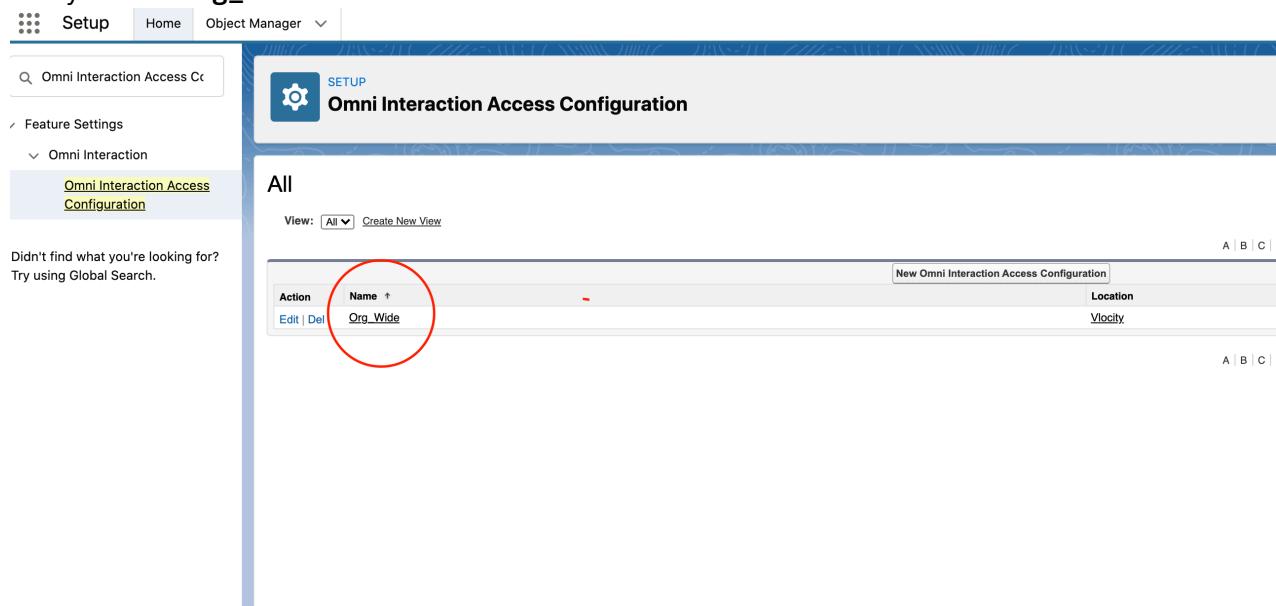
Required Versions

Beginning Winter '22.

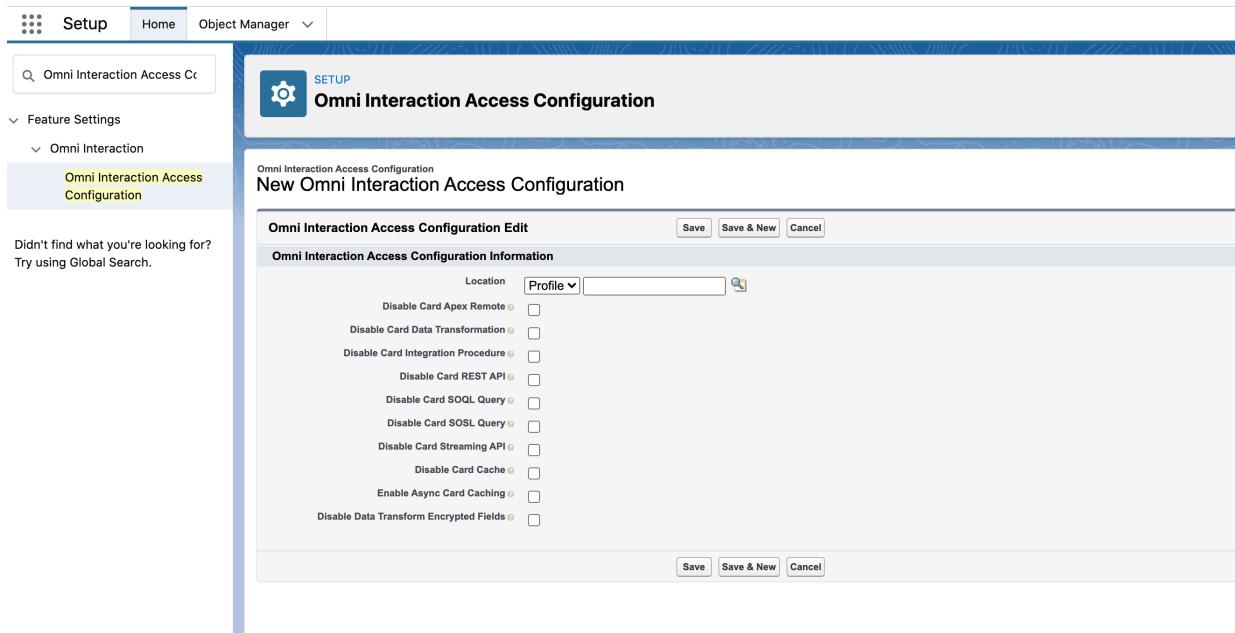
1. Go to Setup, and search for and select **Omni Interaction Access Configuration**.



2. Verify that an **Org_Wide** record exists:



- If so, click **Edit** for **Org_Wide**.
- If not, click **New Omni Interaction Access Configuration** to create a record.



3. Check or uncheck the checkbox for each data source that you want to enable or disable.
4. Click **Save**.

Manage Flexcard Data Source User and Profile Access in Omnistudio

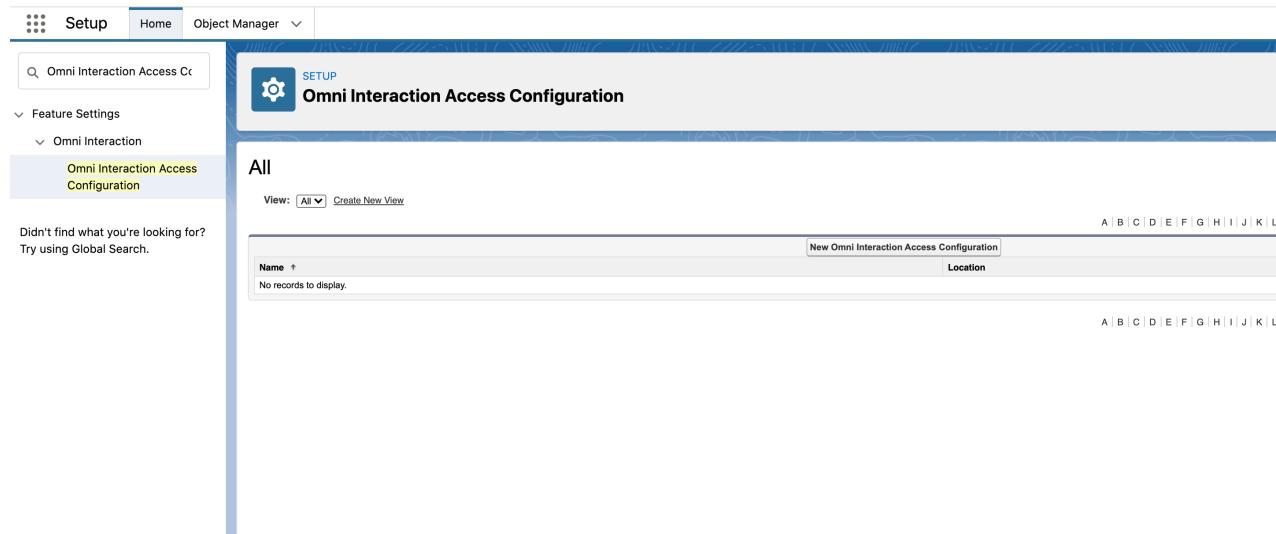
Enable or disable specific Flexcard data sources for an Omnistudio profile or user. For example, prevent admins with limited permissions from using more complex data sources like an Apex REST or Apex Remote.

To enable or disable access to data sources org wide, see [Manage Flexcard Data Source Org Access in Omnistudio](#).

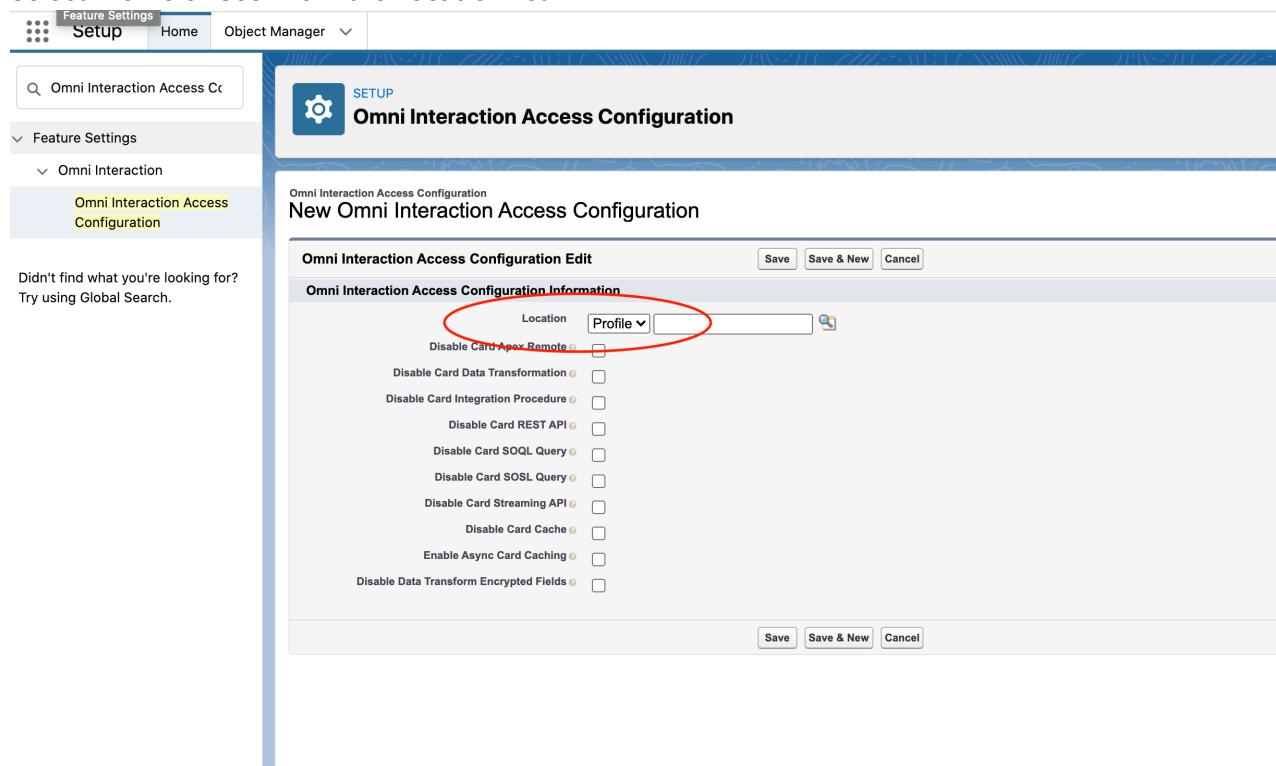
Required Versions

Beginning Winter '22.

1. Go to Setup, and search for and select **Omni Interaction Access Configuration**.



2. Click **New Omni Interaction Access Configuration** to create a record for a specific user or profile.
3. Select **Profile** or **User** from the **Location** list.



- Select **Profile** to manage data sources for a specific profile, such as *System Administrator*.
 - Select **User** to manage data sources for a specific user, such as *Jane Doe*.
4. Check or uncheck the checkbox for each data source that you want to enable or disable.
 5. Click **Save**.

Omnistudio

Use Omnistudio Data Mappers to declaratively retrieve, write, and modify data between Salesforce and other Omnistudio components. While you can process data with custom code (such as Apex classes), Omnistudio Data Mappers are faster to build, easier to update, and simpler to manage. They're also reusable, helping you to move solutions quickly from development to production.

Types of Data Mappers

For each task of retrieving, writing, and transforming data, Omnistudio has a different type of Data Mapper.

Data Mapper Extract, Turbo Extract, and Load can handle [Custom Input and Output for Omnistudio Data Mappers in Omnistudio](#). Data Mappers can access [external objects](#), custom metadata, and Salesforce objects. No special syntax or configuration is required.



- Data Mapper Extract – reads data from one or more Salesforce objects and JSON output, custom data, or XML with field mappings.
- Data Mapper Turbo Extract – reads data from a single Salesforce object along with fields from related objects. The Turbo Extract is faster at run time and is useful for simpler extractions that don't require queries or more complex output mapping.
- Data Mapper Load – creates and updates Salesforce data from JSON, XML, or custom input.
- Data Mapper Transform – transforms input data into a new structure as output. Output examples include renamed fields, restructured XML or JSON, changes with formulas and functions, and converted formats (such as from JSON to PDF). Because it works with data in memory, it doesn't use a Salesforce Object Query Language (SOQL) query or other means to read from Salesforce objects.

Data Mappers in a Workflow

Data Mappers typically supply Salesforce data to Omniscripts, Integration Procedures, Flexcards, and Apex classes, and write the related updates to Salesforce.

For example, an Omniscript guides customers through a workflow to update their account.

1. The Omniscript calls a Data Mapper Extract to read data from Salesforce, such as the Account object.
2. A customer modifies and adds new data in the Omniscript.

3. A Data Mapper Transform checks the input data and changes it to the right formats as needed.
4. The Omniscript calls Data Mapper Load to write the new and modified data to Salesforce.

Data Mapper Extract

Data Mapper Extracts read Salesforce data and return results in JSON, XML, or custom formats. You can filter the data and select the fields to return. You can use formulas, default values, and translations on your data. Data Mapper Extract provides required data to Omniscripts, Integration Procedures, and Flexcards.

Configure the Data Mapper Extract, define the initial extraction, add formulas, add fields, create mapping, and set properties. See [Configure an Omnistudio Data Mapper Extract](#).

Determine whether you need multiple extract steps or relationship notation. See [When to Use Multiple Extract Steps](#) and [Relationship Notation versus Multiple Extract Steps](#).

Data Mapper Turbo Extract

Data Mapper Turbo Extract gets data from a single Salesforce object type and includes fields from related objects. You can filter the data, and select the fields to return by mapping the related objects. You can't use formulas, custom JSON, default values, and transformations on a Data Mapper Turbo Extract.

When compared with a Data Mapper Extract, a Data Mapper Turbo Extract offers simpler configuration and better performance at run time.

Data Mapper Load

Data Mapper Load accepts input data in JSON, XML, or custom formats and writes the data to Salesforce objects. You can use formulas and attributes on your data. For example, a user runs a case-handling Omniscript, finishes entering data, and saves. The script calls a Data Mapper Load to record the data entered.

A Data Mapper Load can get input data in three ways:

- Omniscript or Integration Procedure – during execution, an Omniscript or Integration Procedure builds JSON with the data required for the business use case. When the script invokes the Data Mapper Load, the JSON is sent as input.
- Data Mapper API REST call – if a call invokes a Data Mapper by using a POST action, the payload of the call can include the JSON.
- Apex code – specify the JSON as a parameter in the Apex code to the Data Mapper Load.

To modify the input data, you can define formulas, transform values, and change the output data type. See [Use Formulas in Omnistudio Data Mappers](#). To specify how the resulting data is written to Salesforce objects, you map fields from the output JSON to fields in Salesforce objects. See [Configure an Omnistudio Data Mapper Load](#). When invoked, the Data Mapper Load applies its mappings and formulas to the input data to create the output data. Then it loads the output data into Salesforce

objects according to the mappings.

Data Mapper Transform

Data Mapper Transform performs transformation of intermediate data without reading from or writing to Salesforce objects. You can add formulas to the data.

Data Mapper Transform performs these tasks:

- Convert an input data to an output data, and vice versa. The input and output data can be in JSON or XML format.
- Restructure input data and rename fields.
- Substitute values in fields (all Data Mappers can substitute values)
- Convert data to PDF, Docusign, or document template format.

Data Mapper Transform is essential for Omniscripts that must populate a Docusign template. See [Creating a Data Mapper Interface to Map an OmniScript to DocuSign](#). It's also essential to fill fields in a PDF document. See [Creating a Data Mapper Interface to Map an OmniScript to a PDF](#).

Working with Omnistudio Data Mappers

Use the Data Mapper Designer to build and configure server-side processes that declaratively retrieve, write, and modify data. You can create Data Mappers from the Omnistudio or Data Mappers app. Or you can create them from Data Mapper elements in Integration Procedures and Omniscripts.

Build an Omnistudio Data Mapper

To get, save, and modify Salesforce data, build Data Mappers with the basic configurations: Input source, queries, formulas, mappings of input and output data, and output destination. The available configurations depend on the type of Data Mapper that you're building.

Invoke Omnistudio Data Mappers

Easy to build and reusable, you can call Data Mappers from other Omnistudio tools, such as Integration Procedures, Omniscripts, and Flexcards. But Apex classes, batch jobs, or REST APIs, can also call Data Mappers. With a Data Mapper, centrally manage data transactions to use in multiple applications.

Security for Omnistudio Data Mappers and Integration Procedures

You can control access to Data Mappers and Integration Procedures using settings that reference Sharing Settings and Sharing Sets or Profiles and Permission Sets.

Preview and Test Data Mappers

You can preview and test the input and output of a Data Mapper in the Data Mapper Designer.

Examples of Omnistudio Data Mappers

For each type of Data Mapper, examples show how they're set up for specific, common uses. Each example provides sample JSON for input. Each shows the setup for input, output, mapping, query, and formula, as they apply to the Data Mapper type and the use case.

Working with Omnistudio Data Mappers

Use the Data Mapper Designer to build and configure server-side processes that declaratively retrieve, write, and modify data. You can create Data Mappers from the Omnistudio or Data Mappers app. Or you can create them from Data Mapper elements in Integration Procedures and Omniscripts.

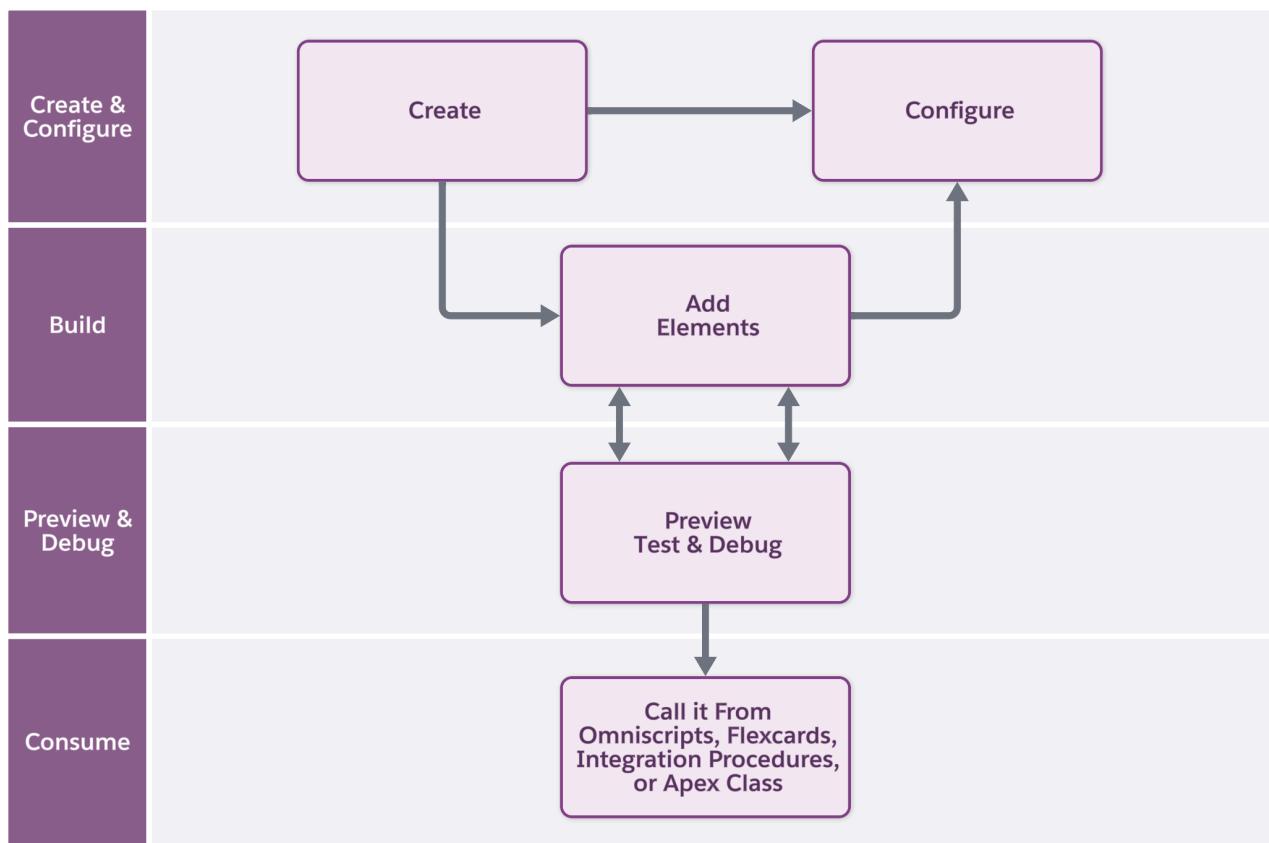
From the App Launcher, open the Omnistudio Data Mapper app. Click New to create a Data Mapper, or click an existing Data Mapper from the list to edit it.

You must provide a unique name, an interface type, and input and output types. Then save your changes, before you can continue working on it. Build the Data Mapper by adding objects, filters, mappings, formulas, and editing input or output data in XML or JSON.

To limit who can run this Data Mapper, assign permissions. Enter roles, profiles, permission sets, or custom permissions that include runtime access. Examples: Role:Architect, Role:Developer, PermSet:OmniStudio Admin Group. See [Syntax of the Required Permission Property](#).

After you create and configure your Data Mapper, you can preview and test it in the Preview pane. Add input parameters and then click Execute to run the procedure. When the Data Mapper finishes, you can see the response output, debug log, execution sequence, any errors, and other detailed information to help you debug and refine your Data Mapper.

When the Data Mapper works as expected, activate it to call it from Omniscripts, Flexcards, Integration Procedures, or Apex classes. As you continue to develop and refine a Data Mapper, you can save new versions to preserve earlier versions of your work. Only one version can be active at a time. When you activate a version, Omnistudio deactivates all other versions of the Data Mapper .



Export or Import an Omnistudio Data Mapper

Export or import Data Mappers to use them in another org, such as when you create them in a sandbox. When exporting or importing data packs for Data Mappers, if you encounter Apex limit errors such as heap or CPU limits, reduce the size of the data pack by unselecting dependencies during the export process. Additionally, break the data pack into multiple smaller data packs. As a best practice, we recommend including no more than 10 elements in each data pack.

Export or Import an Omnistudio Data Mapper

Export or import Data Mappers to use them in another org, such as when you create them in a sandbox. When exporting or importing data packs for Data Mappers, if you encounter Apex limit errors such as heap or CPU limits, reduce the size of the data pack by unselecting dependencies during the export process. Additionally, break the data pack into multiple smaller data packs. As a best practice, we recommend including no more than 10 elements in each data pack.

Export a Data Mapper

You can export a Data Mapper from the new designer with the Salesforce CLI.

Before you begin, [Set Up Salesforce CLI](#)

1. From a terminal in VS Code, enter this command, and replace the listed variables with the appropriate

values.

```
sfdx force:data:tree:export -q "SELECT Id, SourceObject,ExpectedInputOtherData,ExpectedOutputJson,Description,ExpectedOutputXml,IsDeletedOnSuccess,IsProcessSuperBulk,OverrideKey,PreviewOtherData,SynchronousProcessThreshold,TargetOutputDocumentIdentifier,GlobalKey,Name,IsAssignmentRulesUsed,IsXmlDeclarationRemoved,XmlOutputTagsOrder,IsSourceObjectDefault,InputParsingClass,ExpectedOutputOtherData,PreviewSourceObjectData,OutputType,PreviewJsonData,IsRollbackOnError,BatchSize,ResponseCacheType,IsNullInputsIncludedInOutput,VersionNumber,OutputParsingClass,Type,IsErrorIgnored,ExpectedInputJson,ExpectedInputXml,RequiredPermission,PreviewXmlData,InputType,ResponseCacheTtlMinutes,TargetOutputFileName,IsFieldLevelSecurityEnabled,PreprocessorClassName, (SELECT Id,MigrationPattern,InputObjectQuerySequence,FormulaResultPath,FormulaSequence,LinkedFieldName,IsDisabled,MigrationCategory,MigrationType,OutputFieldName,MigrationValue,FilterGroup,LinkedObjectSequence,GlobalKey,Name,OutputCreationSequence,DefaultValue,LookupReturnedFieldName,IsRequiredForUpsert,MigrationProcess,FilterDataType,InputObjectName,FormulaExpression,LookupObjectName,MigrationAttribute,MigrationGroup,FilterValue,FilterOperator,InputFieldName,MigrationKey,IsUpsertKey,LookupByFieldName,OutputFieldFormat,TransformValueMappings,OutputObjectName FROM OmniDataTransformItems) FROM OmniDataTransform" -u org_alias -p -d export_directory -p
```

org_alias

The org from where you're exporting the Data Mapper.

export_directory

The directory in the project where the exported JSON file is stored.

2. Verify whether the file in JSON format is downloaded in the directory.
3. Run the import command.

Import a Data Mapper

You can import a Data Mapper from the new designer with the Salesforce CLI.

Before you begin, [Set Up Salesforce CLI](#).

From a terminal in VS Code, enter this command, and replace *org_alias* with the org from where you're importing the Data Mapper.

```
sfdx force:data:tree:import -p ./export_directory/OmniDataTransform-OmniDataTransformItem-plan.json -u samp-org
```

Export a Data Mapper with Designer on a Managed Package

If you're using the designer on a managed package, you can export a Data Mapper from the Data Mapper Home page.

1. From the App Launcher, find and select **Omnistudio Data Mappers**.
2. Select the version of the Data Mapper that you want to export.
3. Click .
4. Select the item to export, and click **Next**.
5. Review the item to export, and click **Next**.
6. If needed, update the name and description.
7. To store the data pack and access it later from the Omnistudio DataPacks tab, select **Add to Library**.
8. To download the data pack to your computer, select **Download**.
9. Click **Done**.

Import a Data Mapper with Designer on a Managed Package

If you're using the designer on a managed package, you can import a Data Mapper from the Data Mapper Home page.

1. From the App Launcher, find and select **Omnistudio Data Mappers**.
2. Click **Import**.
3. Click **Browse**, select a data pack from your computer that you want to upload, and click **Open**.
4. Click **Next**.
5. Select the item to import, and click **Next**.
6. Review the item to import.
7. Click **Next** and then **Done**.

Build an Omnistudio Data Mapper

To get, save, and modify Salesforce data, build Data Mappers with the basic configurations: Input source, queries, formulas, mappings of input and output data, and output destination. The available configurations depend on the type of Data Mapper that you're building.

- For input and output, select the format, or type, of the data, such as JSON, XML, or sObject. For custom data, you can use an Apex class to parse that data.
- Use queries to define what objects and fields you're extracting data from. In the Data Mapper Designer, use the Extract interface to select the data.
- With formulas, you can modify the input data when needed. You can reuse formula results in other formulas or mappings within the Data Mapper. Each result is stored as a key-value pair: The formula result path defines the key and the formula defines the value.
- With mapping, you connect the data you've extracted and modified with formulas to a structure for your output data.

A Data Mapper Extract has all configurations available. The Data Mapper Transform and Load types don't include queries because they're not retrieving data from Salesforce objects. Instead, these types use data provided in a previous step, such as a Data Mapper Extract, an HTTP action in an Integration Procedure, or an Omniscript. Because a Data Mapper Turbo Extract is a simpler type, it doesn't have formulas or mappings.



Omnistudio Data Mapper Best Practices

To maximize the benefits of Data Mappers, follow the best practices whenever possible.

Configure Date and Time Settings in Data Mappers

Configure the default date format within Omnistudio Data Mappers.

Environment Variables in Omnistudio Data Mappers and Integration Procedures

You can use environment variables to define Default Values and Filter Values, and in Formulas.

Inputs and Outputs for Omnistudio Data Mappers

The configuration of inputs and outputs in Salesforce Omnistudio Data Mappers is highly flexible. The primary data formats are JSON and XML with other options available.

Use Formulas in Omnistudio Data Mappers

To add data to the output of a Data Mapper, define *formulas*. Several types of Data Mappers (Extract, Transform, and Load) support formulas. When you define a formula, you map its output to the output JSON (for extracts and transforms) or Salesforce object field (for loads).

Configure an Omnistudio Data Mapper Extract

Configure a Data Mapper Extract by specifying the object and filters. To enhance the performance of the Data Mapper Extract, you can define formulas, add fields to write data to, create mappings, and configure additional properties.

Configure a Data Mapper Turbo Extract

To configure a Data Mapper Turbo Extract, specify the object type, filters, fields to extract, and options.

Configure an Omnistudio Data Mapper Load

Configure a Data Mapper Load by specifying the object type and filters. To enhance the performance of the Data Mapper Load, you can define formulas, add fields to write data to, and configure additional properties.

Map Inputs and Outputs for Transformations

To map data from the input to the output, go to the Transform tab. If you're using the designer on a managed package, go to the Output tab. You can also handle null values, data types, caching, and list transforms.

Improving Data Mapper Performance with Caching

Omnistudio offers multiple caching options to improve performance and minimize unnecessary data processing in Data Mappers. The two primary user-configurable caching types are Org Cache and Session Cache, which are available in the Data Mapper designer. These caching mechanisms are part of the Salesforce Platform Cache and are implemented through Scale Cache in Omnistudio with standard runtime. Each serves a distinct purpose and comes with specific configuration considerations.

Omnistudio Data Mapper Best Practices

To maximize the benefits of Data Mappers, follow the best practices whenever possible.

- Use unique names for Omniscript elements and Data Mapper response nodes.
- Create targeted Data Mappers that only extract or load the data needed for one operation.
- Use relationship notation (queries) whenever possible to pull data from other SObjects. For more information, see [Relationship Notation versus Multiple Extract Steps](#).
- To ensure data security and maintain compliance with Salesforce encryption access controls, always check that a user has the **View Encrypted Data** permission before displaying or processing decrypted values of encrypted fields.
- To avoid performance issues on the server, keep the number of SObjects to three or fewer.
- Ensure that all filtering and sorting (ORDER BY) operations are on indexed fields. The Id and Name fields are always indexed. For more information, see [Indexes](#) in Salesforce Help.
- Use caching to store frequently accessed, infrequently updated data. See [Improving Data Mapper Performance with Caching](#).
- In a Data Mapper Extract, when you use the `Equals` filter (=) with a null value, the filtering doesn't function correctly due to a known limitation. To prevent this issue, add an additional `Not Equal To` (!=) filter. For example, to filter for null account values that have a phone number associated with them, use `accountId = null AND phone != null`.

To determine whether a Data Mapper or an Integration Procedure is best for your use case, see [When to Use Omnistudio Data Mappers and Integration Procedures](#).

-  **Note** From version 240 to 242.7, Data Mapper versioning is available in Omnistudio, in Setup, under Omnistudio Settings. With version 242.8, Data Mapper versioning isn't available. Data Mapper versioning isn't recommended, but if you enable it, it does work. If you want to disable it, make sure every Data Mapper in your org has only one version, then contact Salesforce support.

Configure Date and Time Settings in Data Mappers

Configure the default date format within Omnistudio Data Mappers.

In Omnistudio, a date, time, or datetime field's input is processed by using a list of predefined formats. However, if you want Omnistudio to use the date/time format associated with a user's locale, use the `UserLocaleDateTime` Omni Interaction Configuration setting.

1. From Setup, enter `Omni Interaction Configuration` in the Quick Find box, and then select **Omni Interaction Configuration**.
2. Click **New Omni Interaction Configuration**.
3. For label and name, enter `UserLocaleDateTime` without space and without changing the case.
4. Enter `true` as the value.
5. Save your changes.

To test if the configuration works, find or create a user with a locale that uses the dd/mm/yyyy date format, for example, English (India). Then, create an Omniscript that uses a Data Mapper (Load or Extract) with a date input. Add a date in the dd/mm/yyyy format.

With the correct configuration, the Data Mapper receives the user's date, time, or datetime input as a string and parses it to a date, time, or datetime by verifying the user's locale settings.

Environment Variables in Omnistudio Data Mappers and Integration Procedures

You can use environment variables to define Default Values and Filter Values, and in Formulas.

For example, the filter `$Vlocity.N_DAYS_AGO:30` extracts the cases created in the last 30 days.

If you're using an environment variable as a Filter value, you must double-quote it. These variables are case-sensitive.

Environment Variable	Description
<code>\$Vlocity.TODAY</code>	Today's date
<code>\$Vlocity.TOMORROW</code>	Tomorrow's date
<code>\$Vlocity.NOW</code>	Current date and time. <code>\$Vlocity.NOW</code> applies to the user or org time zone, while the <code>NOW()</code> function applies UTC time.
<code>\$Vlocity.N_DAYS_FROM_NOW:{N}</code>	Specify the number of days from now
<code>\$Vlocity.N_DAYS_AGO:{N}</code>	Specify the number of days ago
<code>\$Vlocity.NULL</code>	Null
<code>\$Vlocity.StandardPricebookId</code>	ID of the standard price book
<code>\$Vlocity.DocumentsFolderId</code>	Documents folder ID
<code>\$Vlocity.true</code> or <code>\$Vlocity.TRUE</code>	Boolean TRUE
<code>\$Vlocity.false</code> or <code>\$Vlocity.FALSE</code>	Boolean FALSE

Environment Variable	Description
\$Vlocity.UserId	Current user ID
\$Vlocity.Percent	Percent character. Useful for escaping % characters in URLs and text field values in Omniscripts, Integration Procedures, and Data Mappers so they aren't mistaken for merge field syntax.
\$Vlocity.CpuTotal	Apex CPU value
\$Vlocity.DMLStatementsTotal	Number of Data Manipulation Language statements
\$Vlocity.DMLRowsTotal	Number of Data Manipulation Language rows
\$Vlocity.HeapSizeTotal	Heap size value
\$Vlocity.QueriesTotal	Number of queries run
\$Vlocity.QueryRowsTotal	Number of query rows fetched
\$Vlocity.SoslQueriesTotal	Number of SOSL queries run
User.userProfileName	Get the current user's profile name on Flexcard
userProfile	Get the current user's profile name on Omniscript

Inputs and Outputs for Omnistudio Data Mappers

The configuration of inputs and outputs in Salesforce Omnistudio Data Mappers is highly flexible. The primary data formats are JSON and XML with other options available.

JSON is the native and most common format used throughout Omnistudio components. Data Mappers (Extract, Load, Transform) accept JSON payloads passed from Integration Procedures, OmniScripts, or external REST API calls. JSON is also the default and most common format for Extract and Transform Data Mappers, which other Omnistudio components or external REST clients consume.

Use XML for integration with legacy or enterprise systems that rely on XML-based messaging. Load and

Transform Data Mappers can accept XML payloads, which is useful for handling responses from callouts made within an Integration Procedure. If you use XML output for Extract and Transform Data Mappers, Omnistudio can send data to external XML-dependent systems.

For highly non-standard data structures that JSON or XML can't handle, Data Mappers offer custom input (for Load and Transform) and output (for Extract and Transform) as an option through Apex.

Finally, because a Data Mapper Transform's purpose is to manipulate data, it can convert data into formats for document generation. A Transform converts structured JSON into the format to create a PDF. Transforms also structure data to populate fields in a DocuSign envelope or another document template.

Omnistudio Data Mapper Output Data Types

Many data types can be assigned to data in the Output JSON, including boolean, currency, string, various number and list types, and date formats.

XML and JSON in Omnistudio Data Mappers

In addition to JSON, Data Mappers can have XML input and output. The XML format in Data Mappers is an unordered key-value, JSON-like structure.

Custom Input and Output for Omnistudio Data Mappers in Omnistudio

By default, Data Mappers handle JSON and XML data. To handle other data formats, you can configure a Data Mapper to use a custom input or output that you implement as a custom class. For example, you can define a Data Mapper Transform that accepts CSV-formatted data and outputs it as JSON.

Omnistudio Data Mapper Output Data Types

Many data types can be assigned to data in the Output JSON, including boolean, currency, string, various number and list types, and date formats.

- Boolean
- Currency: Converts to Currency using the Salesforce org's currency code.
- CurrencyRounded: Converts to Currency, rounds the number, and removes the decimals.
- Double
- Integer
- JSON: Serialize as JSON. Omit if the transform output type is JSON unless you must embed JSON in JSON (unlikely)
- List<Double>
- List<Integer>
- List<Map>: Converts Map<String, Object> to List<Map<String, Object>>
- List<String>
- Number
- Number(3): Converts to a number with decimals where the decimal precision is specified in parentheses. For example, the output type Number(3) outputs a number with three decimals.
- Object: Deserializes a String as Map<String, Object> or List<Object>
- String: For handling and limitations, see [Primitive Data Types](#) in the *Apex Developer Guide*.
- Multi-Select: Convert to Salesforce multi-select string (semicolon-delimited list)
- Date: Details in the following table.

-  **Note** If you cast a list of values to a primitive type like Double, Integer, String, or Boolean, only the first element of the list is output.

The format of input Date values must be mm/dd/yyyy or ISO-compliant date-time (YYYY-MM-DD hh:mm:ss) with GMT times. You can format output date-time values using any of the [date-time templates supported by Oracle Java](#). Specify the output format as a string argument to the DATE() output data type, for example:

```
Date("EEE, d MMM yyyy HH:mm:ss Z")
```

Here are examples of different output formats for the same date-time value.

Format	Output
Date(MM/dd/YYYY) (default)	07/04/2001
Date("yyyy.MMMM.dd GGG hh:mm aaa")	2001.July.04 AD 12:08 PM
Date("EEE, d MMM yyyy HH:mm:ss Z")	Wed, 4 Jul 2001 12:08:56 +0000

XML and JSON in Omnistudio Data Mappers

In addition to JSON, Data Mappers can have XML input and output. The XML format in Data Mappers is an unordered key-value, JSON-like structure.

To specify the order in which the elements of the XML output are serialized, populate the **Expected XML Output** pane with an XML structure that is composed in the desired order.

This example compares equivalent structures in JSON and XML.

JSON

```
{
  "Root": {
    "#text": "rootValue",
    "@attribute1": "attVal1",
    "#ns": "http://namespace1",
    "#ns#a": "http://namespace2",
    "Child": [
      {
        "@attribute2": "attval2"
      },
      {
        "
```

```
        "#text": "childValue"
    }
]
}
}
```

XML

```
<Root attribute1="attVal1" xmlns="http://namespace1" xmlns:a="http://namespace2">
    rootValue
    <Child attribute2="attval2"></Child>
    <Child>childValue</Child>
</Root>
```

Data Mappers that transform XML to JSON by using these conventions to handle the differences between the formats.

Values in an input XML node are translated to output JSON with the key `#text`.

XML:

```
<Root>rootValue</Root>
```

JSON:

```
{
  "Root": {
    "#text": "rootValue"
  }
}
```

Values in an input XML node are translated to output JSON with a key that is preceded by an at sign (`@`).

XML:

```
<Root attribute1="attVal1"></Root>
```

JSON:

```
{
```

```
"Root": {  
    "@attribute1": "attVal1"  
}  
}
```

Namespaces in an input XML node are prepended with `#ns` in the output JSON.

XML:

```
<Root xmlns="http://namespace1" xmlns:a="http://namespace2">  
</Root>
```

JSON:

```
{  
    "Root": {  
        "#ns": "http://namespace1",  
        "#ns#a": "http://namespace2",  
    }  
}
```

Colons, which are reserved characters in Data Mapper mappings, are replaced by `#`.

XML:

```
<Root:Data>data</Root:Data>
```

JSON:

```
{  
    "Root#Data": {  
        "#text": "data",  
    }  
}
```

Custom Input and Output for Omnistudio Data Mappers in Omnistudio

By default, Data Mappers handle JSON and XML data. To handle other data formats, you can configure a Data Mapper to use a custom input or output that you implement is a custom class. For example, you can define a Data Mapper Transform that accepts CSV-formatted data and outputs it as JSON.

Options are as follows:

- Data Mapper Load: Custom input (output is always a Salesforce object)
- Data Mapper Transform: Custom input and output
- Data Mapper Extract: Custom input and output

To configure a Data Mapper to use a custom input or output, set its type to Custom. Then specify the class that contains the serialize and deserialize methods that perform the operation. The following table shows properties for a Data Mapper Transform configured with a custom input and output.

Property	Value
Data Mapper Interface Name	Custom Serialization Transform
Interface Type	Transform
Input Type	Custom
Custom Input Class	CSVProcessor
Output Type	Custom
Custom Output Class	CSVProcessor

For ease of mapping, you can paste sample input into the Expected Input and Expected Output panes.

To create the logic required to custom input and output, you must define a custom class that implements *Callable*. For custom output, implement the *serialize* method. For custom input, implement the *deserialize* method. You can't rename the input and output parameters.

The following example shows the basic structure of a customer serialization and deserialization class.

```
global with sharing class PreprocessorClassExample implements Callable {
    public Object call(String action, Map<String, Object> args) {

        Map<String, Object> input = (Map<String, Object>)args.get('input');
        Map<String, Object> output = (Map<String, Object>)args.get('output');
        Map<String, Object> options = (Map<String, Object>)args.get('options');

        return invokeMethod(action, input, output, options);
    }

    private Object invokeMethod(String methodName, Map<String, Object> input, Map<String, Object> output, Map<String, Object> options) {
        if (methodName == 'serialize') {
            return serialize(input, output);
        }
        else if (methodName == 'deserialize') {
            return deserialize(input, output);
        }
    }
}
```

```
        return false;
    }
    /*
     * Serializes Apex objects into JSON content. Return String json
     */
    global Boolean serialize(Map<String, Object> input, Map<String, Object> output) {
        String jsonString = '';
        // code
        output.put('output', jsonString); // JSON String
        return true;
    }
    /*
     * Deserializes the specified JSON string into collections of primitive data types. Return Object ((Map<String, Object>))
     */
    global Boolean deserialize(Map<String, Object> input, Map<String, Object> output) {
        Map<String, Object> returnMap = new Map<String, Object>();
        // code
        output.put('output', returnMap); // ---> collections of primitive data types Map<String, Object>, List<Map<String, Object>>()
        return true;
    }
}
```

The following example serializes and deserializes CSV data.

```
global with sharing class CSVProcessor implements Callable {
    public Object call(String action, Map<String, Object> args) {

        Map<String, Object> input = (Map<String, Object>)args.get('input');
        Map<String, Object> output = (Map<String, Object>)args.get('output');
        Map<String, Object> options = (Map<String, Object>)args.get('options');

        return invokeMethod(action, input, output, options);
    }
    private Object invokeMethod(String methodName, Map<String, Object> input, Map<String, Object> output, Map<String, Object> options)
    {
        System.debug('METHOD NAME >>> ' + methodName);
        if (methodName == 'deserialize')
        {
```

```
        return deserialize(input, output);
    }
    return false;
}
/*
Example Output: 'Column2,Column1\nvalue0,value1.0\nvalue0.1,value1.1';
*/
/*
Deserializes the specified JSON string into collections of primitive data types. Return Object ((Map<String, Object>))
Input: 'Column2,Column1\nvalue0,value1.0\nvalue0.1,value1.1';
Output:
[
{
    "Column1Test": "value1.0",
    "Column2Test": "value0"
},
{
    "Column1Test": "value1.1sl",
    "Column2Test": "value0.1"
}
]
*/
public class csvInput {
    @AuraEnabled
    public string VersionData;
}
global Boolean deserialize(Map<String, Object> input, Map<String, Object> output)
{
    System.debug(input);
    System.debug(input.get('input'));
    System.debug(input.toString());
    //csvInput s = (csvInput)JSON.deserialize((string)input.get('input'), csvInput.class);
    Object data = input.get('input');
    csvInput s;
    if(data instanceof String) {
        s = (csvInput)JSON.deserialize(String.valueOf(data), csvInput.class);
    } else { // try to preserve the source formatting
        s = (csvInput)JSON.deserialize(JSON.serialize(data), csvInput.class);
    }
}
```

```
Blob decodedInputBlob = System.EncodingUtil.base64Decode(s.VersionData);
String decodedCsvString = decodedInputBlob.toString();
Object inputValue = decodedCsvString;
System.debug(inputValue);
if (inputValue != null && inputValue instanceof String)
{
    List<String> valueSet = ((String)inputValue).split('\n');
    List<Map<String, String>> csvList = new List<Map<String, String>>();
    g>>();
    List<String> columns = new List<String>();
    for (Integer i = 0; i < valueSet.size(); i++)
    {
        String value = valueSet[i];
        if (String.isBlank(value))
        {
            continue;
        }
        if (i == 0)
        {
            List<String> valSet = value.split(',');
            for (Integer y = 0; y < valSet.size(); y++)
            {
                columns.add(valSet[y]);
            }
        }
        else
        {
            List<String> valSet = value.split(',');
            if (columns.size() >= valSet.size())
            {
                Map<String, String> rows = new Map<String, String>();
                for (Integer z = 0; z < valSet.size(); z++)
                {
                    rows.put(columns[z], valSet[z]);
                }
                csvList.add(rows);
            }
        }
    }
    output.put('output', csvList);
    //Map<String, Object> returnMap = new Map<String, Object>();
    //output.put('output', returnMap);
    System.debug(output);
}
```

```
        return True;
    }
    return False;
}
}
```

For example, suppose your CSV file has this content:

Name	Email	Phone
Mary Baker	mbaker@example.com	415-555-3331
Joe Baker	jbaker@example.com	415-555-3332
Russell Baker	rbaker@example.com	415-555-3333

The example `CSVProcessor` class converts it to this JSON input:

```
[
  {
    "Name": "Mary Baker",
    "Email": "mbaker@example.com",
    "Phone": "415-555-3331"
  },
  {
    "Name": "Joe Baker",
    "Email": "jbaker@example.com",
    "Phone": "415-555-3332"
  },
  {
    "Name": "Russell Baker",
    "Email": "rbaker@example.com",
    "Phone": "415-555-3333"
  }
]
```

Use Formulas in Omnistudio Data Mappers

To add data to the output of a Data Mapper, define *formulas*. Several types of Data Mappers (Extract, Transform, and Load) support formulas. When you define a formula, you map its output to the output JSON (for extracts and transforms) or Salesforce object field (for loads).

For details about the operators and functions that you can use in formulas, see [Omnistudio Formulas and Functions](#).

-  **Note** If a variable name contains spaces or non-alphanumeric characters, enclose the variable name in double quotes and precede it with `var:` in formulas. For example, if the JSON node name is `Primary Guardian`, specify it in formulas as `var:"Primary Guardian"`. If the name of a custom field includes special characters, sometimes you can't reference the field in a formula.

In the Omnistudio standard runtime, entering the incorrect query formula for Data Mappers returns null but doesn't display error logs. If you get a null response from a query, update the query.

1. In the Data Mapper Interface page, go to the **Formulas** tab.
2. Click **Add Formula**. An empty formula is added to the list.
3. In the **Formula** field, specify the desired logic. For example, to determine the total price of the purchase items by a customer, enter a formula such as:

`SUM(Products:Price)`

You can also use relationship notation in formulas to reference fields in a parent object. See [Relationship Notation versus Multiple Extract Steps](#).

4. In the **Formula Result Path** field, specify a JSON node in which to store the formula result.
5. Map the result to the final output.
 - For an Extract Data Mapper, on the **Output** tab, map the formula result to the output structure. Use a colon (:) to delimit levels in the input and output paths in mappings.
 - For a Transform Data Mapper, on the **Transforms** tab, map the formula result to the output structure. Use a colon (:) to delimit levels in the input and output paths in mappings.
 - For a Load Data Mapper, for each sObject that you want to update, on the **Fields** tab, map the formula result to the field that you want to update.
6. Load: For each sObject that you want to update, go to its **Fields** tab and map the formula result to the specific field that you want to update.

Create an Omnistudio Data Mapper Example with a Formula

The following example accepts a list of prices and uses a formula to compute the total price.

To download a DataPack of this example for Omnistudio, click [here](#).

To create this Data Mapper Transform, perform these steps:

1. Go the Omnistudio Data Mapper Designer tab and click **New**. The Create dialog is displayed.
2. Specify a name for the Data Mapper and configure its settings as follows:
 - **Interface Type:** Transform
 - **Input Type:** JSON
 - **Output Type:** JSON
3. Click **Save**. The Data Mapper Interface page is displayed.

4. Go to the Formulas tab and click **Add Formula**.
5. In the **Formula** field, enter *SUM(Products:Price)*.
6. In the **Formula Result Path** field, enter *TotalPrice*.
7. On the Transforms tab, expand the **Input JSON** pane and paste this JSON structure into it:

```
{  
    "CustomerName": "Bob Smith",  
    "Products": [  
        {  
            "Name": "iPhone",  
            "Price": 600  
        },  
        {  
            "Name": "iPhone Case",  
            "Price": 30  
        },  
        {  
            "Name": "Ear Buds",  
            "Price": 200  
        }  
    ]  
}
```

8. Expand the **Expected JSON Output** pane and paste this JSON structure into it:

```
{  
    "CustomerName": "Bob Smith",  
    "TotalPrice": 830,  
    "Products": [  
        {  
            "Name": "iPhone",  
            "Price": 600  
        },  
        {  
            "Name": "iPhone Case",  
            "Price": 30  
        },  
        {  
            "Name": "Ear Buds",  
            "Price": 200  
        }  
    ]  
}
```

9. Click **Quick Match**. In the Quick Match window, click **Auto Match**, then click **Save**.
10. On the Preview tab, click **Execute**.

If you have built the example correctly, the JSON structure in the Response pane matches the Expected JSON Output except for the order of the top-level nodes.

11. In the Input pane, change one or more of the prices, then click **Execute** again.

Note how the `TotalPrice` value changes.

Configure an Omnistudio Data Mapper Extract

Configure a Data Mapper Extract by specifying the object and filters. To enhance the performance of the Data Mapper Extract, you can define formulas, add fields to write data to, create mappings, and configure additional properties.

1. From the App Launcher, find and select **Data Mappers**.
2. Select the Data Mapper Extract that you want to configure.
3. Click **Extract**, and complete these steps:
 - a. To add an object, click **+**. If you're using the designer on a managed package, click **Add Object**.
 - b. Search and select an extraction object.
 - c. Enter an extraction object path.
 - d. Add required filters that determine the data to be read. To filter for null values, use the `$V{velocity.NULL}` variable.

The most basic filter is `Id = Id`, which sets the Id of the object to an input parameter named Id. Data Mapper Extract also supports case-sensitive filter queries using `Equals`, and case-insensitive filter queries using `Not Equal To`. The INCLUDES and EXCLUDES operators apply only to multi-select picklist fields. See [Querying Multi-Select Picklists](#) in Salesforce Help. Turbo Extract has an additional operator option, IN. The IN operator lets you match values to items in an array. For example, if the filter is `Lastname IN Names`, and the JSON input is `{"Names": ["Miller", "Torres"]}`, records with Miller or Torres in their LastName fields are retrieved.

When you use the `Equals` filter (=) with a null value, the filtering doesn't function correctly due to a known limitation. To prevent this issue, add an additional `Not Equal To` (!=) filter. For example, to filter for null account values that have a phone number associated with them, use `accountId = null AND phone != null`.

- e. To configure additional filters, click **▼** and choose filters:
 - **AND**: Add another filter and specify that both filters must be true.
 - **OR**: Add another filter and specify that either filter can be true.
 - **LIMIT**: Valid values are 1 through 50000; the default is 50000.
 - **OFFSET**: To implement paging logic, add a filter to specify where record retrieval starts. The OFFSET must be a dynamic, not a static, value. Choose OFFSET from the dropdown list and specify the name of a node in the input payload where the caller maintains the offset (for example, `offsetValue`).

 **Note** The Data Mapper Extract designer adjusts filter logic when a new OR filter is inserted between existing filters. For more information, see [this Salesforce Knowledge Article](#).

Use OFFSET with LIMIT to specify the first record to appear on a page in a multi-page retrieval. The offset must be an integer. For example, if the LIMIT is 5, the offsets for successive pages are 0, 5, 10, and so on.

 **Note** Salesforce imposes a 2000-record limit on queries that use OFFSET. Allowed OFFSET values are from 0 to 2000. See [Workaround for offset 2000 limit on SOQL query](#).

- **ORDER BY:** Sort the results by the specified field. To sort by multiple fields, specify a comma-separated list of field names in order of precedence, for example `LastName, FirstName`. You can optionally specify ASC or DESC for an ascending or descending sort. You can optionally specify NULLS FIRST or NULLS LAST to place blank values at the beginning or end. For example, you can sort Accounts by the number of employees, listing the companies of unknown size first and the largest employers last. Specify the ORDER BY value `NumberOfEmployees ASC NULLS FIRST`.
 - **DELETE:** Remove a filter.
- f. To add options to the Value list of Filter, click **Data**, and then click **Edit Input Data**. Add and define input values. For example, "BillingCountry": "USA".
- g. To view extraction step data and object fields, click **Data**, and then click **View Extraction Step JSON**. To view all objects, select **Show all sObject Fields**.

 **Note** Salesforce enforces a limit of 100 SOQL queries per Apex transaction. If you exceed this limit while extracting objects, the Data Mapper execution fails, and an error message is displayed. See [Per-Transaction Apex Limits](#).

4. Click **Formula**, add a formula, and define it for the Data Mapper Extract. See [Use Formulas in Omnistudio Data Mappers](#).
 - a. To add a Formula block, click **+**. If you're using the designer on a managed package, click **Add Formula**.
 - b. In the Formula field, enter the desired logic.
 - c. In the Formula Result Path field, specify a JSON node in which you want to store the formula result.
 - d. If you're using this Data Mapper for testing purposes, disable the formula. Click **•••**, and then click **Disable**. If you're using the designer on a managed package, select the **Disabled** checkbox.
5. Click **Mapping**, and then click **Create Mapping**. If you're using the designer on a managed package, click **Output** and then click **+**.

Configure these settings for mapping:

- a. From the extraction step data, choose the source field.
- b. Specify the desired data output path.
- c. If you want to ensure the data is interpreted correctly, choose an output data type from the list.
- d. If needed, add a default value.
- e. To define a data transformation, add key-value pairs. This step replaces the defined key with the value in the output.
- f. Save your changes. If you want to add more mappings, click **Save & New**. If you're using the designer on a managed package, click **+**.

To make mapping easier for a large number of fields, paste the data from the calling Omniscript or Integration Procedure into the Expected Output pane. This populates the data output path list.

6. To quickly match an input field to an output data path, click **Quick Match**, and use one of these options:
 - Drag and drop: This option is available only if you're using the designer on a managed package. Drag the source sObject field from the input field to the output field, or vice versa. For an output that has no source sObject input field, drag the output mapping from the output field to the Matched Mappings column.
 - Pair: Manually select an input field and an output field, and click **Pair**.
 - Auto Match: Click **Auto Match**. Fields with matching names are matched automatically.
7. To create a data output path and add it to the output path list for mapping, click **Data > More > Edit Expected Output Data**, and then add the expected output path.
8. To view the current data output path, click **Data > More > View Current Output Data**.
9. To view extraction step data and object fields, click **Data > View Extraction Step JSON**. To view all objects, select **Show all sObject Fields**.
10. To configure additional properties, click **Options** and configure these properties:
 - Specify the number of minutes that the cached value is retained in the org cache or session cache, if data caching is enabled.
 - To enable data caching, choose a cache type from the Salesforce Platform Cache Type list:
 - - Session Cache: Data related to users and their login sessions.
 - - Org Cache: All other types of data.See [Improving Data Mapper Performance with Caching](#).
 - Configure field-level security to check the user's access permissions for the fields before executing the Data Mapper. Selecting this option disables the Org Cache, but not the Session Cache.
 - To include null values in the Data Mapper response, select **Overwrite target for all null inputs**.
11. Save your changes.

When to Use Multiple Extract Steps

The number of extract steps that you need depends on how the object types that you're extracting from are related. There are three basic scenarios: single, single with relationship notation, and multiple.

Relationship Notation versus Multiple Extract Steps

Using relationship notation improves the performance of Omnistudio Data Mappers that retrieve data from a parent of the primary object. Use this notation in the Extract JSON Paths for the parent object's fields instead of adding a second object in the Data Mapper's Extract tab.

When to Use Multiple Extract Steps

The number of extract steps that you need depends on how the object types that you're extracting from are related. There are three basic scenarios: single, single with relationship notation, and multiple.

- Extracting data from a single object type, for example, Cases, requires only one extract step.
- Extracting data from a primary object and one or more parent objects also requires only one extract step. An example is extracting Cases with some Account data for each Case. See [Relationship Notation versus Multiple Extract Steps](#).
- Extracting data from a primary object and one or more child objects requires at least two extract steps, one for each object type.

For example, to find Cases by Case Number, you can use a filter such as CaseNumber = Number in the extract step.

The objects are queried in the order that you specify them in the extract steps. It's important for the third scenario, when you must use data from a previously read object to filter a subsequent object.

For example, to find all the cases associated with a specific account, you can read the account with a basic filter such as Id = Id in the first extract step. Then read the cases with a filter such as AccountId = Acct:Id in the second extract step. (The AccountId is a Case field; the Acct:Id is a reference to the Id field of the Account.)

This data is read into the extraction step JSON. The Extract Step JSON pane shows the top-level hierarchy defined by the output paths that you specify on this tab.

By default, if a value is null, no node is created for the field in the output JSON. To ensure that a node is created, regardless of whether the field is null, go to Options and check **Overwrite target for all null inputs**.

What's Next

[Relationship Notation versus Multiple Extract Steps](#)

Relationship Notation versus Multiple Extract Steps

Using relationship notation improves the performance of Omnistudio Data Mappers that retrieve data from a parent of the primary object. Use this notation in the Extract JSON Paths for the parent object's fields instead of adding a second object in the Data Mapper's Extract tab.

Relationship notation in Data Mappers is based on relationship queries in Salesforce. For more information about relationship names and other relevant topics, see [Relationship Queries](#) in Salesforce Help. You can use only child-to-parent relationship queries on Data Mappers.

Suppose you want to retrieve Contact data that includes the name of the Account with which the Contact is associated, and the Account is a parent object of Contact. You can use multiple extract steps or relationship notation.

To use multiple extract steps, first define extraction steps for Contact and Account objects on the Data Mapper's Extract tab. Then define Extract JSON paths for the Contact and Account fields.

To use relationship notation, first define an extraction step for the Contact object on the Data Mapper's Extract tab. Then define Extract JSON paths for the Contact and Account fields.

The Extract JSON Path for the Name field of the Account object is `Account.Name`. `Account` is the name of the relationship that the Contact object has with the parent Account object. The relationship name is different from the name of the field that references the parent object, which is AccountId. Most relationship names for standard objects follow this convention and omit the Id suffix.

If you supply a Contact Id on the Preview tab, you get the same Response for that Contact for both Data Mappers:

```
{  
    "CompanyName": "Acme",  
    "LastName": "Tomlin",  
    "FirstName": "Leanne"  
}
```

However, if you look in the Debug Log, you see two SOQL queries for the Data Mapper that uses multiple extract steps:

```
SELECT id, accountid, firstname, lastname, name FROM Contact WHERE (Id = '0031U0000HUob9QAD')  
SELECT id, name FROM Account WHERE (Id = '0011U00000KbYZAQAA3')
```

You see only one SOQL query for the Data Mapper that uses relationship notation:

```
SELECT id, firstname, lastname, account.name FROM Contact WHERE (Id = '0031U0000HUob9QAD')
```

! **Important** Running only one SOQL query per primary object is always noticeably faster than running two. A Data Mapper performs processing steps for each SOQL query that it runs, and Salesforce contributes overhead for each SOQL query as well. With one query rather than two, you have half the overhead.

Configure a Data Mapper Turbo Extract

To configure a Data Mapper Turbo Extract, specify the object type, filters, fields to extract, and options.

1. From the App Launcher, find and select **Data Mappers**.
2. Select the Data Mapper Turbo Extract that you want to configure.
3. Search and select an extraction object.
4. Enter an extraction object path.
5. Add required filters that determine the data to be read. To filter for null values, use the `$V{velocity.NULL}` variable.

The most basic filter is `Id = Id`, which sets the ID of the object to an input parameter ID. The `INCLUDES` and `EXCLUDES` operators apply only to multiselect picklist fields. See [Querying Multi-Select Picklists](#) in Salesforce Help.

Turbo Extract has an additional operator option `IN`. The `IN` operator lets you match values to items in an array. For example, if the filter is `LastNames IN Names`, and the JSON input is `{"Names":`

`["Miller", "Torres"]}`, it retrieves all records with Miller or Torres in their LastName fields.

Data Mapper Turbo Extract doesn't support multiple input fields with single quotation marks in the field values. Use a Data Mapper Extract instead. For instance, adding one input called FirstName with the value T'an, and another input called LastName with the value C'os, isn't supported on a Data Mapper Turbo Extract.

If you're using the designer on a managed package, you can download a data pack of this example for Omnistudio from [here](#).

6. To configure additional filters, click the down arrow:

AND	Add another filter and specify that both filters must be true
OR	Add another filter and specify that either filter can be true.
LIMIT	Specify the maximum number of records returned. Valid values are 1 through 50000; the default is 50000.
OFFSET	<p>To implement paging logic, add a filter to specify where record retrieval starts. Choose OFFSET from the dropdown list and specify the name of a node in the input payload where the caller maintains a dynamic offset (for example, <code>offsetValue</code>).</p> <p>Use OFFSET with LIMIT to specify the first record to appear on a page in a multi-page retrieval. The offset must be an integer. For example, if the LIMIT is 5, the offsets for successive pages are 0, 5, 10, and so on.</p> <p> Note Salesforce imposes a 2000-record limit on queries that use OFFSET. Allowed OFFSET values are from 0 to 2000. See Workaround for offset 2000 limit on SOQL query.</p>
ORDER BY	Sort the results by the specified field. To sort by multiple fields, specify a comma-separated list of field names in order of precedence, for example <code>LastName, FirstName</code>

	You can optionally specify ASC for ascending sort or DESC for descending sort. You can optionally specify NULLS FIRST or NULLS LAST to place blank values at the beginning or end. For example, you can sort Accounts by the number of employees, listing the companies of unknown size first and the largest employers last. Specify the ORDER BY value <code>NumberOfEmployees ASC NULLS FIRST</code> .
DELETE	Remove a filter.

- To retrieve data from a parent of the primary object, select related objects. For example, `Contact.Account`. For information about the notation, see [Relationship Notation versus Multiple Extract Steps](#).

After you select a relationship with another object, its fields appear in Available Fields. Clicking these fields reveals second-level relationships, enabling navigation through five levels. To go back a level, simply click the list and select the previous level.

- In Field Mapping, search for the available fields. This option isn't available if you're using the designer on a managed package.
- Select the fields that you want to extract, and move them from Available to Selected Fields. The Id field is always included in Selected Fields.
- Click **Options**, and configure additional settings:
 - Specify the number of minutes that the cached value is retained in the org cache or session cache, if data caching is enabled. The minimum value is 5 minutes.
 - To enable data caching, choose a cache type from the Salesforce Platform Cache Type picklist. See [Improving Data Mapper Performance with Caching](#).

Session Cache - Data related to users and their login sessions.

Org Cache - All other types of data.

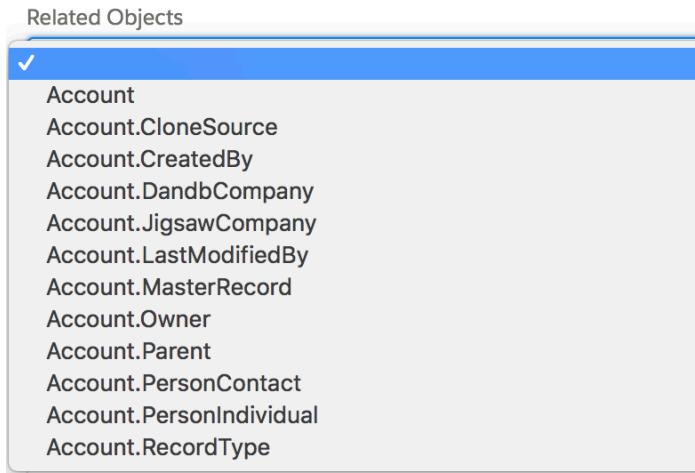
- Configure field-level security to check the user's access permissions for the fields before executing the Data Mapper. Selecting this option disables the Org Cache, but not the Session Cache.
 - To include null values in the Data Mapper response, select **Overwrite target for all null inputs**.
 - Save your changes.
- To observe the effects of caching when testing the Data Mapper Turbo Extract, deselect **Ignore Cache**. See [Improving Data Mapper Performance with Caching](#).

Related Object Fields in Omnistudio Data Mapper Turbo Extracts

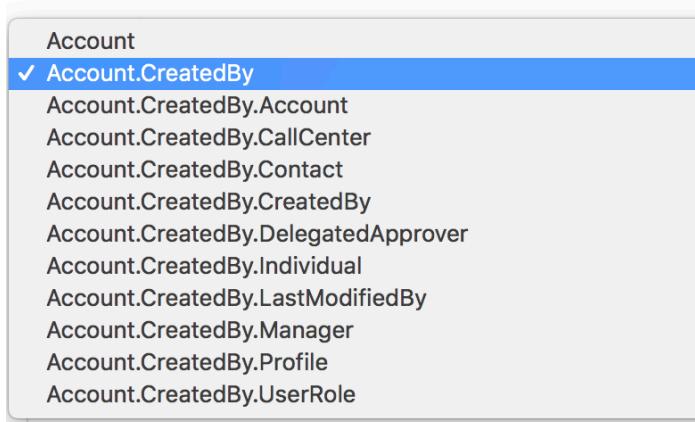
Although a Data Mapper Turbo Extract retrieves from a single object type, you can select specific fields in related objects.

For more information about relationship names and other relevant topics, see [Relationship Queries](#) in Salesforce Help. Data Mappers support only child-to-parent relationship queries.

When you click the Related Objects dropdown list, relationships to other objects appear, for example:

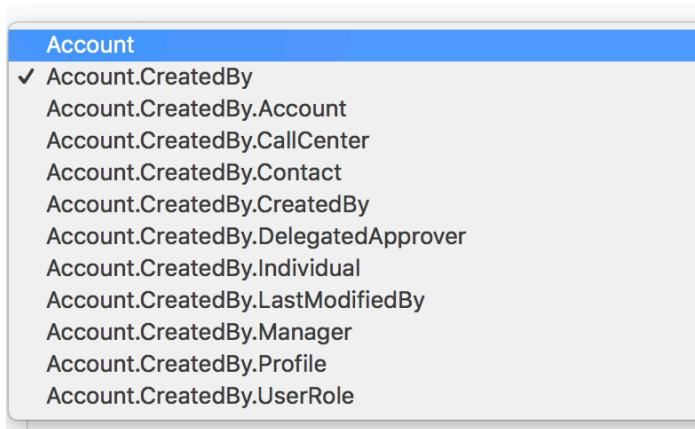


If you select one of those relationships and click the list again, second-level relationships to other objects appear, for example:



You can traverse up to five relationship levels.

To back up a level, click the list and select the previous level, for example:



After you've selected a relationship with another object, fields in that object appear in the left field-selection list. You can enter a search value and move the fields to the right list as you would the base object fields.

Configure an Omnistudio Data Mapper Load

Configure a Data Mapper Load by specifying the object type and filters. To enhance the performance of the Data Mapper Load, you can define formulas, add fields to write data to, and configure additional properties.

1. From the App Launcher, find and select **Data Mappers**.
2. Select the Data Mapper Load that you want to configure.
3. Click **Objects**, and add a Data Mapper Load Object:
 - To add an object, click **+**. If you're using the designer on a managed package, click **Add Object**.
 - From the Load Object list, search, and select an object.
4. Click **Formula**, add a formula, and define it for the Data Mapper Load. See [Use Formulas in Omnistudio Data Mappers](#).
 - a. To add a Formula block, click **+**. If you're using the designer on a managed package, click **Add Formula**.
 - b. In the Formula field, enter the desired logic.
 - c. In the Formula Result Path field, specify a JSON node in which you want to store the formula result.
 - d. If you're using this Data Mapper for testing purposes, disable the formula. Click **•••**, and then click **Disable**. If you're using the designer on a managed package, select the **Disabled** checkbox.
5. Click **Mapping**, and then click **Create Mapping**. If you're using the designer on a managed package, click **Fields** and then click **+**.

You can also use the Quick Match option to quickly match an input field to an output object field. You can match fields either manually using the Pair option or automatically using the Auto Match option.

- a. Configure these settings for mapping:
 - **Input JSON Path:** The key of the JSON node containing the data you want to write to Salesforce.
 - **Domain Object:** The Salesforce object that you want to map to.
 - **Domain Object Field:** The field in the Salesforce object that you want to update. In fields that

contain JSON data, you can't map JSON nodes individually. Stringify all JSON nodes that have values to ensure that no JSON data is lost. For example:

```
"JsonField": "{\"JsonNode1\":\"Text\", \"JsonNode2\":100, \"JsonNode3\":true}"
```

If you're updating multiple Salesforce objects, the designer displays a separate tab for the mappings of each target object. See [Multiple Related Object Loads \(Link Mappings\)](#).

- **Add Key-Value Pair:** These mapping settings copy the value to a top-level JSON node specified as a key.
 - b. To control how the mapping is performed, you can configure these optional settings. If you're using the designer on a managed package, click **All**, and configure these settings:
 - **Output Data Type:** Must be compatible with the data type of the target field. The Output Data Type setting must be compatible with the target field in Salesforce. If you configure incompatible types, the load operation can fail. For a list of valid output types, see [Omnistudio Data Mapper Output Data Types](#).
 - **Default Value:** Value to be loaded if the field in the Data Mapper output JSON is null. To specify an empty string for string fields that are null, enter a pair of double quotes (""). Omit if **Required for Upsert** is enabled.
 - **Disabled:** Prevents the field from being loaded.
 - **Upsert Key:** Specifies that the field is a key for the Salesforce object being loaded. The Data Mapper uses the value to determine whether it updates an existing record or inserts a new one. If the **Upsert** property is enabled for multiple field mappings, the result is a multi-field upsert key. The Data Mapper uses this key to check the object for unique values to determine whether to create or update an object. Don't use an ID as the upsert key. If the input contains an ID, the Data Mapper updates the record. If the input doesn't contain an ID, Data Mapper creates a record.
 - **Required for Upsert:** If this field is null, it prevents an object record from being updated.
 - **Lookup:** Uses the field value to query for the specified Salesforce data, and writes the result to the output field.
 - c. Save your changes. If you want to add more mappings, click **Save & New**. If you're using the designer on a managed package, click **+**.
6. To configure additional properties, click **Options** and configure these properties:
- **Ignore Errors:** Execute the Data Mapper even if errors occur, skipping only the steps that are having errors. This option is useful when you know a record will fail with limited data and future steps don't rely on previous steps.
 - **Rollback on Error:** Don't create or update the sObjects if errors occur. For more information, see [Apex Transactions](#) and [Transaction Control](#).
 - **Use Assignment Rules:** Use assignment rules for sObjects such as Cases that have user assignment fields. For more information, see [Set Up Assignment Rules](#). Unauthenticated guest users can't trigger assignment rules. If emails are configured in Case assignment rules, checking this option automatically sends emails to users when Cases are assigned.
 - **Overwrite Target For All Null Inputs:** If an input doesn't have a value, set the corresponding output value to NULL.
 - **Delete on Success:** When the input type is sObject, use this option to automatically delete bulk records after successfully bulk-loading data.
 - **Is Default for Interface:** When the input type is sObject, specify this Data Mapper as the default

bundle for the specified interface object.

Create Interface Objects for a Data Mapper Load

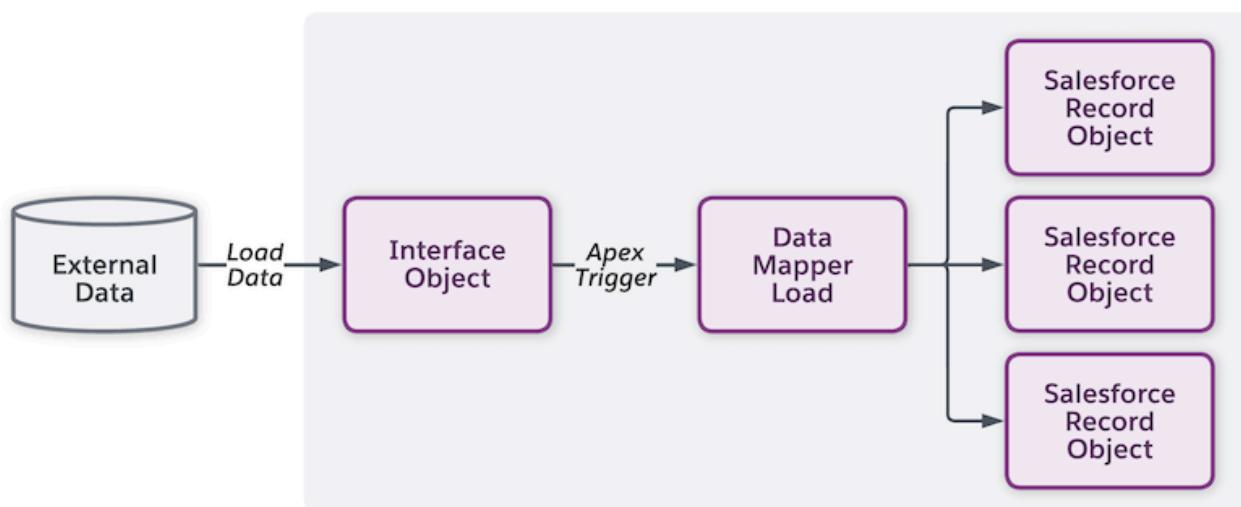
Provide external data, such as a CSV file, as an input to a Data Mapper Load. The external data can be imported into an interface object, which is a custom Salesforce object using a data import tool. Then, map the fields from the custom object to the target Salesforce records using a Data Mapper Load. When you load data to a Salesforce custom object, it fires an Apex trigger that calls a Data Mapper Load. The Data Mapper takes data from the interface object and writes the resulting data to other Salesforce record objects.

Multiple Related Object Loads (Link Mappings)

When loading data into a sequence of objects, you can propagate data directly from one object to another related object. For example, you can use an Omnistudio Data Mapper Load to support an Omniscript that creates an Account and a Contact for the Account.

Create Interface Objects for a Data Mapper Load

Provide external data, such as a CSV file, as an input to a Data Mapper Load. The external data can be imported into an interface object, which is a custom Salesforce object using a data import tool. Then, map the fields from the custom object to the target Salesforce records using a Data Mapper Load. When you load data to a Salesforce custom object, it fires an Apex trigger that calls a Data Mapper Load. The Data Mapper takes data from the interface object and writes the resulting data to other Salesforce record objects.



You can load data to a Salesforce custom object by using Salesforce's Data Import Wizard or a tool such as the Salesforce Data Loader, Informatica, or Talend.

1. In Salesforce, create a custom object and add fields corresponding to the data in the external source.
 - a. Add these string fields to the custom object:
 - DMError (Data Type: Long Text Area)
 - DMProgressData (Data Type: Long Text Area)

- DMName (Data Type: Text): This field is required.
-  **Note** In Omnistudio for Managed Packages, DRBundleName is a required field.
- DMStatus (Data Type: Long Text Area)

These fields are reserved by Data Mappers for internal use.

- Add fields corresponding to the data in the external source.

2. Create an Apex class `DMInterfaceObjectFuture`.

- From Setup, find and select **Apex Classes**.
- Create a new Apex class and add this code.

```
public class DMInterfaceObjectFuture {
    @future(callout=true)
    public static void call(String sObjectIdsAsString) {
        List<String> sObjectIds = (List<String>) JSON.deserialize(sObjectIdsAsString, List<String>.class);
        ConnectApi.DataMapperInterfaceObjectInput input = new ConnectApi.DataMapperInterfaceObjectInput();
        input.interfaceObjectIds = sObjectIds;
        input.interfaceObjectName = String.valueOf(((Id)sObjectIds.get(0)).getObjectType());
        ConnectApi.DataMapperInterfaceObjectConnect.executeDataMapperLoadForInterfaceObjects(input);
    }
}
```

3. Create a new trigger on the custom object that invokes the Data Mapper Load.

```
trigger trigger_name on interface_object_api_name (after insert, after update)
{
    String orgNamespace = [SELECT NamespacePrefix FROM Organization LIMIT 1].NamespacePrefix;
    String prefix = String.isEmpty(orgNamespace) ? '' : orgNamespace + '__';
    List<String> sObjectIds = new List<String>();
    for (sObject sObj : Trigger.new) {
        if (Trigger.isUpdate) {
            if (Trigger.oldMap.get(sObj.Id).get(prefix + 'DMStatus__c') != Trigger.newMap.get(sObj.Id).get(prefix + 'DMStatus__c') ||
                Trigger.oldMap.get(sObj.Id).get(prefix + 'DMError__c') != Trigger.newMap.get(sObj.Id).get(prefix + 'DMError__c') ||
                Trigger.oldMap.get(sObj.Id).get(prefix + 'DMProgressData__c') != Trigger.newMap.get(sObj.Id).get(prefix + 'DMProgressData__c') ||
                Trigger.oldMap.get(sObj.Id).get(prefix + 'DMName__c') != Trigger.newMap.get(sObj.Id).get(prefix + 'DMName__c')) {

```

```
        continue;
    }
}

sObjectIds.add(sObj.Id);
}

if (!sObjectIds.isEmpty()) {
    DMInterfaceObjectFuture.call(JSON.serialize(sObjectIds));
}
}
```

4. Create and configure the Data Mapper Load.
 - a. Create a Data Mapper Load with **SObject** as the input type.
 - b. Select the interface object that you created for the input data from the dropdown list.
 - c. In the Data Mapper Load, map the interface object fields with the Salesforce records that you want to update.
 - d. To verify that your Data Mapper Load can access data in the interface object, load a small amount of test data into the interface object and run the Data Mapper.
5. Import the external data into the interface object by using the Salesforce bulk load tool or your preferred bulk load tool.

You can now use interface objects in Data Mappers in the same way as standard Salesforce objects.



Tip If you use interface objects regularly, rather than for a one-time bulk load, create a tab for the object to facilitate quick and easy access.

Create Interface Objects for a Data Mapper Load Example

1. Create an interface object **TestInterfaceObject__c** with these fields:
 - Data__c: This is the data field to add custom data.
 - DMStatus__c
 - DMErro__c
 - DMProgress
 - DMName__c
2. Create a Data Mapper Load with the **TestInterfaceObject__c** object as the input that creates records of Account and Contact.
3. Configure the Data Mapper Load mapping.
 - **TestInterfaceObject__c.Data__c** to **Account.Name**
 - **TestInterfaceObject__c.Data__c** to **Contact.LastName**
4. Create a new record for **TestInterfaceObject__c**.
For example, when you add a new record, Account1, in the Data_c field of the interface object, a new Account record with Name = Account1 is created and a new Contact record with LastName = Account1 is created.

These fields of the TestInterfaceObject__c object are updated:

- DMStatus__c: {"Account": "success", "Contact": "success"}
- DMProgressData__c: {"Account": "<created account id>", "Contact": "<created contact id>"}

Multiple Related Object Loads (Link Mappings)

When loading data into a sequence of objects, you can propagate data directly from one object to another related object. For example, you can use an Omnistudio Data Mapper Load to support an Omniscript that creates an Account and a Contact for the Account.

To link the Account to its Contact, the Data Mapper must first create the Account and the Contact. Then the Data Mapper populates the Contact record's AccountId field with the Id of the Account record.

To handle this type of relationship, link the objects in the Data Mapper Designer.

1. On the Objects tab, navigate to the source object (Contact in the preceding example) and click **Add Link**.
2. For each field you want to link, configure these settings:
 - **Domain Object Field:** The source field from this object (AccountId in the preceding example).
 - **Linked Object:** Another object in the sequence (Account in the preceding example).
 - **Linked Field:** The target field in the linked object (Id in the preceding example).



Example For an example that uses link mapping, see [Create a Contact for an Existing Account](#).

Map Inputs and Outputs for Transformations

To map data from the input to the output, go to the Transform tab. If you're using the designer on a managed package, go to the Output tab. You can also handle null values, data types, caching, and list transforms.

You can map data by using one of these methods:

- Use Quick Match to quickly map input and output data in a Data Mapper Transform. See [Use Quick Match to Map Data](#).
- Specify each mapping individually using the Create Mapping option.

Both methods apply to JSON input and output and to other input and output types.

Data Restructuring and Renaming

All types of Data Mappers can restructure data and rename fields using mappings. To restructure or rename a field, you map the input path to the desired output path.

This example shows JSON input being mapped to XML output. The incoming top-level Case fields, CaseNumber, AccountId, and AccountName, are reparented under an XML element named <XCASEDETAILS>. Although the list of contacts is unchanged, note the difference between the representation of a list in JSON and in XML. In JSON, the Contacts list is enclosed in square brackets, [], then each item in the Contacts list is enclosed in brackets, {}. In XML, each item in the Contacts list is enclosed in a <CONTACTS> element. For details about converting between JSON and XML, see [XML and JSON in Omnistudio Data Mappers](#).

For the PDF, Docusign, and Document Template output types, output mappings are generated based on the fields in the Target Output file you selected when you created the Data Mapper Transform.

By default, if a value is null, no node is created for the field in the output JSON. To ensure that a node is created, regardless of whether the field is null, go to the Options tab and check **Overwrite target for all null inputs**.

Input Value to Output Value Mappings

To verify that your mappings create the desired structure, go to the Preview tab, paste sample input into the Input pane on the left side of the screen, and click Execute. The results are displayed in the Response pane.

To verify that your mappings create the desired structure and output, go to the Preview tab, paste the sample input into the Input pane in the correct format, and click **Execute**. The results appear in the Response pane. The Data Mapper Transform generates the desired output only if the input data format is correct. For example, for the Date data type, you must enter the supported format, which is mm/dd/yyyy or yyyy-mm-dd.

To define key value pairs for the output, open the mapping, go to Transform Map Values, and enter values in the Key Value pair fields. For example, if the incoming field contains TRUE or FALSE, and your OmniScript requires a Y or N value, enter TRUE or FALSE in the Key field, and Y or N in the Value field, respectively.

Output Data Types

On the Output tab, you can specify the data type of the output data. If you change the type, be sure to choose a compatible output type. (For example, changing a numeric type to a string is fine, but changing a string to a numeric is risky, because a string can contain non-numeric characters.) See [Omnistudio Data Mapper Output Data Types](#).

Data Caching

You can configure optional caching settings on the Options tab:

Time To Live In Minutes: If data caching is enabled, it determines how long data remains in the cache. The minimum value is 5.

Salesforce Platform Cache Type: Enables data caching. Set to **Session Cache** for data related to users and their login sessions, or to **Org Cache** for all other types of data. See [Improving Data Mapper Performance with Caching](#).

Maintain List Order When Combining Multiple Inputs

When you map multiple input lists to the same output list in a Data Mapper Transform, the system does not guarantee the order of the items in the output. The results may vary between executions.

To ensure a predictable and consistent order, use a formula to combine and filter the lists instead of using direct mappings.

Example Scenario

You have three input lists: `inputList1`, `inputList2`, and `inputListNew`. You want to combine them into a single `OutputList` where the items from `inputList1` always appear first, followed by `inputList2`, and then `inputListNew`.

Required Configuration

- Remove direct mappings: Remove any individual mappings that target the `OutputList`.
- Add a formula: On the **Formulas** tab, create a formula to merge the lists in your preferred order.

Formula	Formula Result Path	Description
<code>LIST(inputList1, inputList2, inputListNew)</code>	CombinedList	Merges the lists in the exact order specified.
<code>FILTER(LIST(inputList1, inputList2, inputListNew), "PolicyNumber != null")</code>	OutputList	Merges the lists in order and excludes items with null Policy Numbers.

Mapping the Result

On the **Transforms** tab, map the result of your formula to the final output path.

Input JSON Path	Output JSON Path
OutputList	OutputList

 **Note** Make sure that the output data type for your mapping is set to `List<Map>`.

Use Quick Match to Map Data

Quick Match is an easy way to map input and output data in an Omnistudio Data Mapper Transform.

Transforming Lists with Data Mappers

Data Mappers can perform list transformations. For mapping input to output, Data Mappers require a list input in which every value is paired with a key.

Use Quick Match to Map Data

Quick Match is an easy way to map input and output data in an Omnistudio Data Mapper Transform.

 **Note** Some transforms are too complex for Quick Match. See [Transforming Lists with Data Mappers](#).

1. Expand the Input JSON pane and paste the input data structure into it.
2. Expand the Expected JSON Output pane and paste the desired output structure into it.
3. To quickly match an input field to an output data path, click **Quick Match**, and use one of these options:
 - Drag and drop: This option is available only if you're using the designer on a managed package. Drag the source sObject field from the input field to the output field, or vice versa. For an output that has no source sObject input field, drag the output mapping from the output field to the Matched Mappings column.
 - Pair: Manually select an input field and an output field, and click **Pair**.
 - Auto Match: Click **Auto Match**. Fields with matching names are matched automatically.
4. Save your changes. The Quick Match page closes and the Transforms tab shows the individual mappings.
5. Verify the structure in the Current JSON Output pane. Make sure it matches the structure in the Expected JSON Output pane.

Transforming Lists with Data Mappers

Data Mappers can perform list transformations. For mapping input to output, Data Mappers require a list input in which every value is paired with a key.

For example, the following lists are valid, and each can be converted to the other.

List 1:

```
{  
  "Oldest": "Huey",  
  "Middle": "Dewey",  
  "Youngest": "Louie"  
}
```

List 2:

```
{  
  "Nephews": [  
    ...  
  ]  
}
```

```
{  
    "Name": "Huey"  
,  
{  
    "Name": "Dewey"  
,  
{  
    "Name": "Louie"  
}  
]  
}
```

List 1 lacks a JSON node for the list as a whole, and each value has a different key. List 2 has its own JSON node, and each list item has the same key.

-  **Note** Loop Block components in Integration Procedures require input that is structured like List 2. The list-item key itself can contain key-value pairs. For examples, see [Work with Data and Lists](#).

To convert List 1 to List 2, use the following mappings on the Transforms tab of a Data Mapper Transform. The |1|, |2|, and |3| are list position indexes.

Input JSON Path	Output JSON Path
Oldest	Nephews 1:Name
Middle	Nephews 2:Name
Youngest	Nephews 3:Name

To convert List 2 to List 1, use the reverse mappings.

To download DataPacks of Data Mapper Transforms for these mappings, click these links:

- [List 1 to List 2 example](#) for Omnistudio
- [List 2 to List 1 example](#) for Omnistudio

In the following list, values don't have keys.

List 3:

```
{  
    "Nephews": [  
        "Huey",  
        "Dewey",  
        "Louie"  
    ]  
}
```

```
        "Dewey",
        "Louie"
    ]
}
```

You can convert List 1 to List 3 using the following mappings:

Input JSON Path	Output JSON Path
Oldest	Nephews 1
Middle	Nephews 2
Youngest	Nephews 3

However, you can't convert List 3 to either List 1 or List 2.

To convert List 2 to List 3, and to perform such conversions of lists of any length, see [Convert a List of Objects to a List of Values](#).

If each object of a list has a different key, Data Mapper Transform creates a single map with combined key and value pairs from that list, and ignores any empty values. Here's an example for products for an ecommerce website:

```
{
  "Cleaning products": [
    {
      "Name": "WonderProduct"
      "Property": ""
    },
    {
      "Name": ""
      "Property": "Fast-acting"
    },
    {
      "Scent": "Lavender"
    }
  ]
}
```

Transformed output list:

```
{  
    "Cleaning products": [  
        {  
            "Name": "WonderProduct"  
            "Property": "Fast-acting"  
            "Scent": ":Lavender"  
        }  
    ]  
}
```

For more that you can do with lists, see:

- AVG, FILTER, IF, LIST, LISTMERGE, LISTMERGEPRIMARY, LISTSIZE, MAX, MIN, SORTBY, and SUM functions in the [Supported Data Mapper and Integration Procedure Functions](#)
- [Work with Data and Lists](#) in Integration Procedures
- [List Action for Integration Procedures](#) in Integration Procedures
- [Integration Procedure Action for Integration Procedures](#) in Integration Procedures (the example transforms a list)

Expected Behavior for Array Inputs

When a list input (array) is mapped to an output that is expected to be a single object (non-array), the Data Mapper Transform automatically flattens the array or merges the objects based on the number of records in the input array.

- If the list input array has two records, the Data Mapper Transform flattens the array and attempts to merge the records into a single JSON object in the output. The output node is a single object `{ }` instead of an array `[]`.

Example input:

```
{  
    "In": [  
        { "field1": "data1" },  
        { "field2": "data2" }  
    ]  
}
```

Example output:

```
{  
    "Out": {  
        "field1": "data1",  
        "field2": "data2"  
    }  
}
```

- If the input array contains more than two records (multiple objects), the Data Mapper Transform retains the array structure and does not flatten the records. The array is returned in the output as-is.
- Example input:

```
{  
  "In": [  
    { "field1": "data1" },  
    { "field2": "data2" },  
    { "field3": "data3" }  
  ]  
}
```

Example output:

```
{  
  "Out": [  
    { "field1": "data1" },  
    { "field2": "data2" },  
    { "field3": "data3" }  
  ]  
}
```

Improving Data Mapper Performance with Caching

Omnistudio offers multiple caching options to improve performance and minimize unnecessary data processing in Data Mappers. The two primary user-configurable caching types are Org Cache and Session Cache, which are available in the Data Mapper designer. These caching mechanisms are part of the Salesforce Platform Cache and are implemented through Scale Cache in Omnistudio with standard runtime. Each serves a distinct purpose and comes with specific configuration considerations.

Omnistudio also offers Metadata Cache, which is for internal use only. It's not configurable by users. It stores Omnistudio component metadata or definitions, and not the actual data. You can configure the data to remain in the cache (time to live) for a maximum of 48 to 72 hours.

Omnistudio caches two different types of content:

- Metadata: The definition that includes the structure and configuration of Integration Procedures and Data Mappers, such as steps, mappings, and formulas. This is stored in Metadata Cache and is used to improve load times.
- Data: The response of a Data Mapper or an Integration Procedure that is cached in the Session Cache or Org Cache. Every time, when the same request input is entered or provided, the cached data is returned.

Types of data cache in Data Mappers include:

- Org Cache: The cached data is shared among all users in the same org. The org cache is recommended

only if the data is non-sensitive or not user-specific. For example, picklists, configurations, or object data without any access restrictions. You can configure the data to remain in the cache (time to live) for a maximum of 48 hours.

 **Important** Use org cache only for non-sensitive data.

- **Session Cache:** It stores user-specific data and is limited to active user sessions. It is safer for personalized data and helps reduce redundant processing for data that varies by user profile or permissions. You can configure the data to remain in the cache (time to live) for a maximum of 8 hours.

You can use caching with Data Mappers in three ways:

- Data Mapper metadata is automatically cached. If you use the Scale Cache, this automatic caching occurs unless you turn off the Scale Cache. If you use the Platform Cache, you must allocate space in the VlocityMetadata cache partition to enable this automatic caching.
- You can configure data caching on the Options tab of Data Mapper Extracts, Turbo Extracts, and Transforms.
- If you call a Data Mapper from an Integration Procedure that uses caching, the Data Mapper data is cached along with the Integration Procedure data.

You can also perform a record-level security check for cached data. See [Security for Omnistudio Data Mappers and Integration Procedures](#).

Methods to Clear Metadata from Data Mapper Cache

You can clear metadata from the cache for a Data Mapper or all Data Mappers at once.

1. Go to the Developer Console.
2. Click the user menu and select **Developer Console** from the menu.
3. Select **Debug | Open Execute Anonymous Window**.
4. To clear metadata cache for a Data Mapper, in the Apex code window, execute this code:

```
ConnectApi.DatamapperCacheInputParamRepresentation finalInput = new ConnectA  
pi.DatamapperCacheInputParamRepresentation();  
ConnectApi.DataMapperParamRepresentation param = new ConnectApi.DataMapperPa  
ramRepresentation();  
param.dataMapperName = DataMapperName;  
finalInput.dataMapperList = new List();  
finalInput.dataMapperList.add(param);  
finalInput.cacheStorageType = ConnectApi.CacheStorageType.Metadata;  
ConnectApi.DatamapperClearCacheOutputRepresentation response = ConnectApi.Om  
niDesignerConnect.clearDatamapperCache(finalInput);
```



Note Starting with Summer '25, replace the

`namespace.DRGlobal.clearCacheForDataRaptor('DataMapperName')` method with this Connect API.

- To clear the metadata cache for all Data Mappers, in the Apex code window, execute this code:

```
ConnectApi.DataMapperCacheInputParamRepresentation finalInput = new ConnectA  
pi.DataMapperCacheInputParamRepresentation();  
finalInput.cacheStorageType = ConnectApi.CacheStorageType.Metadata;  
ConnectApi.DataMapperClearCacheOutputRepresentation response = ConnectApi.Om  
niDesignerConnect.clearDataMapperCache(finalInput);
```



Note Starting with Summer '25, replace the `namespace.DRGlobal.clearCacheForAllDataRaptor()` method with the Connect API to clear cache for all Data Mappers.

Configure Caching for a Data Mapper

To configure top-level caching for a Data Mapper, go to the Procedure Configuration and set the **Salesforce Platform Cache Type** and **Time To Live In Minutes** properties.

Connect APIs for Cache Management in Data Mappers

See this table for a summary of Connect APIs to call Data Mappers from Apex classes. Starting with Summer '25, replace the existing methods in the `DRGlobal` Apex classes with the Connect APIs.

Configure Caching for a Data Mapper

To configure top-level caching for a Data Mapper, go to the Procedure Configuration and set the **Salesforce Platform Cache Type** and **Time To Live In Minutes** properties.

- Go to the Data Mapper configuration page and click **Options**.
- Set the **Salesforce Platform Cache Type** to one of these:
 - None** – For no caching
 - Note** Do not cache when working with confidential or regulated information unless security settings are enforced.
 - Session Cache** – For data related to users and their login sessions
 - Org Cache** – For all other types of data
- Important** Use Org Cache only for non-sensitive data.
- Specify a value for **Time To Live In Minutes**. This setting determines how long data remains in the cache. The minimum value is 5. Default and maximum values depend on the cache type:
 - Session Cache** – The default value is 5. The maximum value is 480, equivalent to 8 hours. The cache is cleared when the user's session expires regardless of this value.
 - Org Cache** – The default value is 1440, equivalent to 24 hours. The maximum value is 2880, equivalent to 48 hours.
- Configure field-level security to check the user's access permissions for the fields before executing the Data Mapper. Selecting this option disables the Org Cache, but not the Session Cache.
- To include null values in the Data Mapper response, select **Overwrite target for all null inputs**.
- Save your changes.
- To observe the effects of caching when testing the Data Mapper, deselect **Ignore Cache**.

Connect APIs for Cache Management in Data Mappers

See this table for a summary of Connect APIs to call Data Mappers from Apex classes. Starting with Summer '25, replace the existing methods in the *DRGlobal* Apex classes with the Connect APIs.

Purpose	Connect API	Existing Method (Replace with Connect API)
Clear metadata cache for a Data Mapper	<pre>ConnectApi.DataMapperCacheInputParamRepresentation finalInput = new ConnectApi.DataMapperCacheInputParamRepresentation(); ConnectApi.DataMapperParamRepresentation param = new ConnectApi.DataMapperParamRepresentation(); param.dataMapperName = DataMapperName; finalInput.dataMapperList = new List<ConnectApi.DataMapperParamRepresentation>(); finalInput.dataMapperList.add(param); finalInput.cacheStorageType = ConnectApi.CacheStorageType.Metadata; ConnectApi.DataMapperClearCacheOutputRepresentation response = ConnectApi.OmniDesignerConnect.clearDataMapperCache(finalInput);</pre>	<pre>DRGlobal.clearCacheForDataRaptor('DataMapperName');</pre>

Example:

```
ConnectApi.DataMapperCacheInputParamRepresentation finalInput = new ConnectApi.DataMapper
```

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre data-bbox="633 291 997 1269"> rCacheInputParamRepresentation(); ConnectApi.DataMapperParamRepresentation param = new ConnectApi.DataMapperParamRepresentation(); param.dataMapperName = 'DM_Create_Contact'; finalInput.dataMapperList = new List<ConnectApi.DataMapperParamRepresentation>(); finalInput.dataMapperList.add(param); finalInput.cacheStorageType = ConnectApi.CacheStorageType.Metadata; ConnectApi.DatamapperClearCacheOutputRepresentation response = ConnectApi.OmniDesignerConnect.clearDatamapperCache(finalInput); </pre>	
Clear session cache for all Data Mappers	<pre data-bbox="633 1353 997 1881"> ConnectApi.DatamapperCacheInputParamRepresentation finalInput = new ConnectApi.DatamapperCacheInputParamRepresentation(); finalInput.cacheStorageType = ConnectApi.CacheStorageType.Metadata; ConnectApi.DatamapperClearCacheOutputRepresentation response = ConnectApi.OmniDesignerCo </pre>	<pre data-bbox="1062 1353 1426 1438"> DRGlobal.clearCacheForAllDataRaptor(); </pre>

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre>connect.clearDatamapperCache(finalInput);</pre> <p>Example:</p> <pre>ConnectApi.DatamapperCacheInputParamRepresentation finalInput = new ConnectApi.DatamapperCacheInputParamRepresentation(); finalInput.cacheStorageType = ConnectApi.CacheStorageType.Metadata; ConnectApi.DatamapperClearCacheOutputRepresentation response = ConnectApi.OmniDesignerConnect.clearDatamapperCache(finalInput);</pre>	

Invoke Omnistudio Data Mappers

Easy to build and reusable, you can call Data Mappers from other Omnistudio tools, such as Integration Procedures, Omniscripts, and Flexcards. But Apex classes, batch jobs, or REST APIs, can also call Data Mappers. With a Data Mapper, centrally manage data transactions to use in multiple applications.

For example, an organization receives regularly updated files with new user information in an XML format. They configure a Data Mapper Load to parse, transform, and load this XML data into multiple related Salesforce objects (such as Account, User, and Contact).

As another example, a Salesforce Flow creates or updates an Account, several related Contacts, and a custom Policy record at one time from user input. Rather than creating multiple steps in Flow, create a single Data Mapper Load to handle the entire multi-object upsert. A Flow can call a simple Integration Procedure that invokes the Data Mapper Load.

Omnistudio Data Mapper REST API

You can invoke any type of Data Mapper using the Data Mapper REST API. To update Salesforce

objects using a Data Mapper Load, issue a POST request that includes a JSON payload that is formatted to comply with the input that the Data Mapper load expects. To retrieve data from Salesforce, issue a GET call to a Data Mapper Extract, specifying the Id or parameters identifying the data to be retrieved. The data is returned in the response in JSON format.

Omnistudio Data Mapper Calls From Apex

Starting with Summer' 25, to call a Data Mapper from Apex, call the `ConnectApi.OmniDesignerConnect.executeDataMapper(bundleName, apexInput)` Connect API specifying the name of the Data Mapper and the input data that it requires. This API replaces the `vlocity_ins.DRGlobal.processObjectsJSON()` method. This Connect API removes the dependency on the managed package and provides up to 60% better performance for Data Mapper calls from an Apex class compared to the previous method.

DRGlobal Class and Methods

Use the `ConnectApi.OmniDesignerConnect.executeDataMapper(bundleName, apexInput)` Connect API to call Data Mappers from Apex through the DRGlobal Apex class. Starting with Summer '25, replace the existing methods in the `DRGlobal` Apex classes with the Connect API.

Omnistudio Data Mapper REST API

You can invoke any type of Data Mapper using the Data Mapper REST API. To update Salesforce objects using a Data Mapper Load, issue a POST request that includes a JSON payload that is formatted to comply with the input that the Data Mapper load expects. To retrieve data from Salesforce, issue a GET call to a Data Mapper Extract, specifying the Id or parameters identifying the data to be retrieved. The data is returned in the response in JSON format.

Example

POST Data

```
{  
    "bundleName" : "AccountUpload",  
    "objectList" : {  
        "Agency Information": {  
            "Agency Name": "Vlocity",  
            "Agency Address": "50 Fremont",  
            "Agency City": "San Francisco",  
            "Agency State": "CA",  
            "Agency Zip": "94110",  
        }  
    },  
    "bulkUpload" : false  
}
```

Result

```
{  
    "createdObjectsByOrder": {  
        "Open Account": {  
            "1": [  
                "a10o00000022xVEAAY"  
            ]  
        }  
    },  
    "createdObjectsByType": {  
        "Open Account": {  
            "Account": [  
                "a10o00000022xVEAAY"  
            ]  
        }  
    },  
    "errors": {},  
    "returnResultsData": []  
}
```

Data Mapper Extract Invocation Using GET

To retrieve data from Salesforce using a REST call, issue a GET statement that invokes a Data Mapper Extract. To identify the data to be retrieved you can either specify the object Id or one or more parameters to be matched. Data is returned in the response in the output JSON format that the Data Mapper Extract creates.

Use an ID to Retrieve Data

To retrieve Salesforce data by specifying the Id, issue a GET request that invokes a Data Mapper Extract, using a URL formatted as follows:

```
/services/apexrest/{myOrgNamespace}/v2/DataRaptor/{DataMapperName}/Id
```

Example Request

The following request uses the Id of a Contact to retrieve all open Cases for the Contact.

```
GET /services/apexrest/vlocity_cmt/v2/DataRaptor/OpenCases/a10o00000022xVE
```

Example Result

```
{  
    "Contact": {  
        "Contact Name" : "Dennis Reynolds",  
        "Case Information": [  
    ]  
}
```

```
{  
    "Title": "Wrong widget shipped..."},  
    {  
        "Title": "Overcharged for gizmo..."},  
    {  
        "Title": "Damaged item..."}  
]
```

Use Parameters to Retrieve Data

To retrieve Salesforce data by passing parameters to a Data Mapper extract, issue a GET request using a URL formatted as follows:

```
GET /services/apexrest/{myOrgNamespace}/v2/DataRaptor/{DataMapperName}/?${Param1}=${Val1}&${Param2}=${Val2}...
```

Example Request

The following request passes the first and last name of a contact as input parameters to a Data Mapper extract, which uses them to query for the cases opened by the contact.

```
GET /services/apexrest/vvelocity_cmt/v2/DataRaptor/Open_Cases/?FirstName=Dennis&LastName=Reynolds
```

Data Mapper Load Invocation Using POST

To update Salesforce data, issue a POST request with a URL formatted as follows:

```
/services/apexrest/{myOrgNamespace}/v2/DataRaptor/
```

In the POST data, specify the following parameters:

- `bundleName` – Name of the Data Mapper Load to invoke
- `objectList` – JSON data to be loaded. Must match the format expected by the Data Mapper Load.
- `filesList` – (Optional) Map of keys to base 64-encoded files.
- `bulkUpload` – TRUE to use batch Apex.

Omnistudio Data Mapper Calls From Apex

Starting with Summer' 25, to call a Data Mapper from Apex, call the

`ConnectApi.OmniDesignerConnect.executeDataMapper(bundleName, apexInput)` Connect API specifying the name of the Data Mapper and the input data that it requires. This API replaces the `vlocity_ins.DRGlobal.processObjectsJSON()` method. This Connect API removes the dependency on the managed package and provides up to 60% better performance for Data Mapper calls from an Apex class compared to the previous method.

! **Important** The performance figures in this document are provided for informational purposes only and are based on internal validation and testing under specific conditions. Actual results may vary depending on your component design, production environment, and other factors. These figures provide general guidance and aren't a guarantee of performance.

Specify the input as follows:

- For a JSON object, use `Map<String, Object>`. Set the String to the JSON key and the Object to the JSON value.
- For an array of JSON objects, use `List<Map<String, Object>>`. Set the String to the JSON key and the Object to the JSON value.
- As an alternative, you can specify the input as a string containing the JSON input required by the Data Mapper.

For details about the methods of the `DRGlobal` class, which offer multiple ways to call Data Mappers, see [DRGlobal Class and Methods](#).

Data Mapper Extract or Transform Example

To process the results returned by a Data Mapper extract or transform in Apex, deserialize the output to a `Map<String, Object>` or `List<Map<String, Object>>`.

```
/* Specify Data Mapper extract or transform to call */
String bundleName = 'DataMapperName';

/* Populate the input JSON */
Map<String, Object> objectList = new Map<String, Object>{'MyKey'=>'MyValue'};

/* Call the Data Mapper */
String jsonString = JSON.serialize(objectList);
List<String> jsonInputData = new List<String>();
jsonInputData.add(jsonString);
ConnectApi.DataMapperExecuteInputRepresentation apexInput = new ConnectApi.DataMapperExecuteInputRepresentation();
apexInput.dataMapperInput = jsonInputData;
apexInput.inputType = 'JSON';
```

```
ConnectApi.DataMapperExecuteOptionsRepresentation options = new ConnectApi.DataMapperExecuteOptionsRepresentation();
options.locale = null;
options.shouldSendLegacyResponse = true;
apexInput.options = options;
ConnectApi.DataMapperExecuteOutputRepresentation output = ConnectApi.OmniDesignerConnect.executeDataMapper(bundleName, apexInput);

/* Process the results returned by a Data Mapper Extract or Transform */
List<String> innerResponse = output.response;
for (String currentResponse : innerResponse) {
Map<String, Object> outerMap = (Map<String, Object>) JSON.deserializeUntyped(currentResponse);
List<String> keys = new List<String>(outerMap.keySet());
System.debug(outerMap.get('response'));
}
```

Data Mapper Load Example

Data Mapper loads return JSON containing data about the Salesforce objects that were created or updated by the operation. To process the results returned by a Data Mapper load, deserialize the output to a map. From the resulting map, you can extract data about the objects that were created and any errors that occurred. The Data Mapper load example below shows how to access this data from the result map.

```
String objectList = '{"accountName":"Vlocity", "contractCode":"SKS9181"}';

List<String> jsonInputData = new List<String>();
jsonInputData.add(objectList);

ConnectApi.DataMapperExecuteInputRepresentation apexInput = new ConnectApi.DataMapperExecuteInputRepresentation();

apexInput.dataMapperInput = jsonInputData;
apexInput.inputType = 'JSON';

ConnectApi.DataMapperExecuteOptionsRepresentation options = new ConnectApi.DataMapperExecuteOptionsRepresentation();
options.ignoreCache = false;
options.shouldSendLegacyResponse = true;
apexInput.options = options;

ConnectApi.DataMapperExecuteOutputRepresentation output = ConnectApi.OmniDesign
```

```

erConnect.executeDataMapper(bundleName, apexInput);
List<String> innerResponse = output.response;
String currentResponse = innerResponse[0];
System.debug(currentResponse);

Map<String, Object> outerMap = (Map<String, Object>) JSON.deserializeUntyped(c
urrentResponse);
List<String> keys = new List<String>(outerMap.keySet());
System.debug(outerMap.get('drSObjectResults'));

/*
Process the results of the load: these methods return details about objects af
fected by the Data Mapper Load, in addition to any errors that occurred
*/

Map<String, Object> createdObjectsByType = (Map<String, Object>) resultMap.ge
t('createdObjectsByType');
Map<String, Object> createdObjectsByTypeForBundle = (Map<String, Object>) creat
edObjectsByType.get('bundleName');
Map<String, Object> createdObjectsByOrder = (Map<String, Object>) resultMap.ge
t('createdObjectsByOrder');
Map<String, Object> errors = (Map<String, Object>) resultMap.get('errors');
Map<String, Object> errorsByField = (Map<String, Object>) resultMap.get('errors
ByField');
List<Object> errorsAsJson = (List<Object>) outerMap.get('errorsAsJson'); // Ret
urns input JSON plus per-node errors

```

Data Mapper Load Example with the bulkUpload Parameter

This example code passes the **bulkUpload** parameter to a Data Mapper Load.

```

String objectList = '[{"ProductCode__c": 11050665}, {"ProductCode__c": 1107010
0}]'; // replace this with the input for your Data Mapper Load
String bundleName = 'DRLoadPrice'; // replace this with your Data Mapper name
Map<String, Object> bodyData = new Map<String, Object>();
bodyData.put('bundleName', bundleName);
bodyData.put('objectList', objectList);

List<String> jsonInputData = new List<String>();
jsonInputData.add(bodyData.get('objectList'));

ConnectApi.DataMapperExecuteInputRepresentation apexInput = new ConnectApi.Dat
aMapperExecuteInputRepresentation();

```

```

apexInput.dataMapperInput = jsonInputData;
apexInput.inputType = 'JSON';
ConnectApi.DataMapperExecuteOptionsRepresentation options = new ConnectApi.DataMapperExecuteOptionsRepresentation();
options.ignoreCache = false;
apexInput.options = options;
options.shouldSendLegacyResponse = true;
ConnectApi.DataMapperExecuteOutputRepresentation output = ConnectApi.OmniDesignerConnect.executeDataMapper(bodyData.get('bundleName'), apexInput);

List<String> innerResponse = output.response;

Map<String, Object> outerMap = (Map<String, Object>) JSON.deserializeUntyped(innerResponse[0]);
List<String> keys = new List<String>(outerMap.keySet());
System.debug(outerMap.get('drSObjectResults'));

```

DRGlobal Class and Methods

Use the `ConnectApi.OmniDesignerConnect.executeDataMapper(bundleName, apexInput)` Connect API to call Data Mappers from Apex through the DRGlobal Apex class. Starting with Summer '25, replace the existing methods in the `DRGlobal` Apex classes with the Connect API.

Process a JSON String Object List Using a Specified Data Mapper

The objectList must be a JSON String.

Connect API	Existing Method (Replace with Connect API)
<pre> List<String> jsonInputData = new List<String>(); jsonInputData.add(objectList); ConnectApi.DataMapperExecuteInputRepresentation apexInput = new ConnectApi.DataMapperExecuteInputRepresentation(); apexInput.dataMapperInput = jsonInputData; apexInput.inputType = 'JSON'; ConnectApi.DataMapperExecuteOptionsR </pre>	<pre> static namespace.DRProcessResult processObjectsJSON(String objectList, String bundleName) namespace.DRProcessResult res = namespace.DrGlobal.processObjectsJSON(objectList, bundleName); </pre>

Connect API	Existing Method (Replace with Connect API)
<pre> epresentation options = new ConnectA pi.DataMapperExecuteOptionsRepresent ation(); options.ignoreCache = false; options.shouldSendLegacyResponse = true; apexInput.options = options; ConnectApi.DataMapperExecuteOutputRepr esentation output = ConnectApi.OmniDesign erConnect.executeDataMapper(bundleNam e, apexInput); List<String> innerResponse = outpu t.response; String currentResponse = innerRespon se[0]; System.debug(currentResponse); Map<String, Object> outerMap = (Ma p<String, Object>) JSON.deserializeU ntyped(currentResponse); List<String> keys = new List<Strin g>(outerMap.keySet()); System.debug(outerMap.get('drSObject Results')); </pre>	

Process a List of sObjects Using a Specified Data Mapper

Connect API	Existing Method (Replace with Connect API)
<pre> String jsonstring = JSON.serialize(o bjectList); List<String> jsonInputData = new Lis </pre>	<pre> static namespace.DRP rocessResult processObjects(List<SOb ject> objectList) </pre>

Connect API	Existing Method (Replace with Connect API)
<pre>t<String>(); jsonInputData.add(jsonString); ConnectApi.DataMapperExecuteInputRepresentation apexInput = new ConnectApi.DataMapperExecuteInputRepresentation(); apexInput.dataMapperInput = jsonInputData; apexInput.inputType = 'JSON'; ConnectApi.DataMapperExecuteOptionsRepresentation options = new ConnectApi.DataMapperExecuteOptionsRepresentation(); options.ignoreCache = false; options.shouldSendLegacyResponse = true; apexInput.options = options; ConnectApi.DataMapperExecuteOutputRepresentation output = ConnectApi.OmniDesignerConnect.executeDataMapper(bundleName, apexInput); List<String> innerResponse = output.response; for (String currentResponse : innerResponse) { System.debug(currentResponse); Map<String, Object> outerMap = (Map<String, Object>) JSON.deserializeUntyped(currentResponse); List<String> keys = new List<String>(outerMap.keySet()); System.debug(outerMap.get('drSOBJectResults')); }</pre>	<p>This signature, which doesn't require a DRName, is only for a Data Mapper Load.</p>

Connect API	Existing Method (Replace with Connect API)
<pre> String jsonString = ""; if (additionalInfo != null) { jsonString = JSON.serialize(additionalInfo); } else { jsonString = JSON.serialize(objectList); } List<String> jsonInputData = new List<String>(); jsonInputData.add(jsonString); ConnectApi.DataMapperExecuteInputRepresentation apexInput = new ConnectA pi.DataMapperExecuteInputRepresentat ion(); apexInput.dataMapperInput = jsonInpu tData; apexInput.inputType = 'JSON'; ConnectApi.DataMapperExecuteOptionsR epresentation options = new ConnectA pi.DataMapperExecuteOptionsRepresent ation(); options.ignoreCache = false; apexInput.options = options; options.shouldSendLegacyResponse = true; ConnectApi.DataMapperExecuteOutputRe presentation output = ConnectApi.Omn iDesignerConnect.executeDataMapper(b undleName, apexInput); List<String> innerResponse = outpu t.response; for (String currentResponse : innerR esponse) { </pre>	<pre> static namespace.DRProcessResult pro cessObjects(List<SObject> objectLis t, String bundleName, Map<String, Obje ct> additionalInfo, Map<String, Obje ct> filesMap) </pre> <p>The additionalInfo is a Map that applies to every SObject, such as extra data for processing.</p>

Connect API	Existing Method (Replace with Connect API)
<pre>Map<String, Object> outerMap = (Map<String, Object>) JSON.deserializeUntyped(currentResponse); List<String> keys = new List<String>(outerMap.keySet()); System.debug(outerMap.get('drSObjectResults')); }</pre>	
<pre>String jsonString = ""; if(additionalInfo !=null){ jsonString= JSON.serialize(additionalInfo); }else{ jsonString= JSON.serialize(objectList); } List<String> jsonInputData = new List<String>(); jsonInputData.add(jsonString); ConnectApi.DataMapperExecuteInputRepresentation apexInput = new ConnectApi.DataMapperExecuteInputRepresentation(); apexInput.dataMapperInput = jsonInputData; apexInput.inputType = 'JSON'; ConnectApi.DataMapperExecuteOptionsRepresentation options = new ConnectApi.DataMapperExecuteOptionsRepresentation(); options.shouldSendLegacyResponse = true; apexInput.options = options; ConnectApi.DataMapperExecuteOutputRe</pre>	<pre>static namespace.DRProcessResult processObjects(List<SObject> objectList, String bundleName, Map<String, Object> additionalInfo)</pre> <p>The additionalInfo is a Map that applies to every SObject, such as extra data for processing.</p>

Connect API	Existing Method (Replace with Connect API)
<pre> presentation output = ConnectApi.Omni niDesignerConnect.executeDataMapper (bundleName, apexInput); List<String> innerResponse = output .response; for (String currentResponse : innerR esponse) { Map<String, Object> outerMap = (Ma p<String, Object>) JSON.deserializeU ntyped(currentResponse); List<String> keys = new List<Strin g>(outerMap.keySet()); System.debug(outerMap.get('drSObject Results')); } </pre>	

Process a list or map of objects, bypassing sharing rules, with a specified Data Mapper and locale

These methods ignore Sharing Rules, which ensures that the Data Mapper being invoked is private. See [Sharing Rules](#) in the Salesforce Help.

Connect API	Existing Method (Replace with Connect API)
<pre> String jsonString = JSON.serialize(o bjectList); List<String> jsonInputData = new Li st<String>(); jsonInputData.add(jsonString); ConnectApi.DataMapperExecuteInputRe presentation apexInput = new ConnectA pi.DataMapperExecuteInputRepresentat ion(); apexInput.dataMapperInput = jsonInpu tData; apexInput.inputType = 'JSON'; </pre>	<pre> static namespace.DRProcessResult pro cessFromApex(List<Map<String, Object>> objectLis t, String bundleName, String locale) </pre>

Connect API	Existing Method (Replace with Connect API)
<pre>ConnectApi.DataMapperExecuteOptionsR epresentation options = new ConnectA pi.DataMapperExecuteOptionsRepresent ation(); options.locale = locale; options.shouldSendLegacyResponse = t rue; apexInput.options = options; ConnectApi.DataMapperExecuteOutputRe presentation output = ConnectApi.Om niDesignerConnect.executeDataMappe r(bundleName, apexInput); List<String> innerResponse = outpu t.response; for (String currentResponse : innerR esponse) { Map<String, Object> outerMap = (Ma p<String, Object>) JSON.deserializeU ntyped(currentResponse); List<String> keys = new List<Strin g>(outerMap.keySet()); System.debug(outerMap.get('drSObject Results')); } }</pre>	
<p> Note Users must have appropriate permissions to execute the Data Mapper.</p> <pre>String jsonString = JSON.serialize(o bjectList); List<String> jsonInputData = new Lis t<String>(); jsonInputData.add(jsonString); ConnectApi.DataMapperExecuteInputRep resentation apexInput = new ConnectA</pre>	<pre>static namespace.DRProcessResult pro cessFromApex(Map<String, Object> objectList, String bundleName, String locale)</pre>

Connect API	Existing Method (Replace with Connect API)
<pre> pi.DataMapperExecuteInputRepresentat ion(); apexInput.dataMapperInput = jsonInpu tData; apexInput.inputType = 'JSON'; ConnectApi.DataMapperExecuteOptionsR epresentation options = new ConnectA pi.DataMapperExecuteOptionsRepresent ation(); options.locale = locale; options.shouldSendLegacyResponse = t rue; apexInput.options = options; ConnectApi.DataMapperExecuteOutputRe presentation output = ConnectApi.Omn iDesignerConnect.executeDataMapper(b undleName, apexInput); List<String> innerResponse = outpu t.response; for (String currentResponse : innerR esponse) { Map<String, Object> outerMap = (Ma p<String, Object>) JSON.deserializeUn typed(currentResponse); List<String> keys = new List<Strin g>(outerMap.keySet()); System.debug(outerMap.get('drSObject Results')); } </pre>	

 **Note** Users must have appropriate permissions to execute the Data Mapper.

Process a list or map of objects with a specified Data Mapper and locale, considering sharing rules

Connect API	Existing Method (Replace with Connect API)

Connect API	Existing Method (Replace with Connect API)
<pre> String jsonString = JSON.serialize(o bjectList); List<String> jsonInputData = new Lis t<String>(); jsonInputData.add(jsonString); ConnectApi.DataMapperExecuteInputRep resentation apexInput = new ConnectA pi.DataMapperExecuteInputRepresentat ion(); apexInput.dataMapperInput = jsonInpu tData; apexInput.inputType = 'JSON'; ConnectApi.DataMapperExecuteOptionsRepr esentation options = new ConnectApi.DataM apperExecuteOptionsRepresentation(); options.locale = locale; options.shouldSendLegacyResponse = true; apexInput.options = options; ConnectApi.DataMapperExecuteOutputRe presentation output = ConnectApi.Omn iDesignerConnect.executeDataMapper(b undleName, apexInput); List<String> innerResponse = outpu t.response; for (String currentResponse : innerR esponse) { Map<String, Object> outerMap = (Ma p<String, Object>) JSON.deserializeUn typed(currentResponse); List<String> keys = new List<Strin g>(outerMap.keySet()); System.debug(outerMap.get('drSObject Results')); } </pre>	<pre> static namespace.DRProcessResult pro cess(List<Map<String, Object>> objectLis t, String bundleName, String locale) </pre>
<pre> String jsonString = JSON.serialize(o bjectList); </pre>	<pre> static namespace.DRProcessResult pro </pre>

Connect API	Existing Method (Replace with Connect API)
<pre> objectList); List<String> jsonInputData = new List<String>(); jsonInputData.add(jsonString); ConnectApi.DataMapperExecuteInputRepresentation apexInput = new ConnectApi.DataMapperExecuteInputRepresentation(); apexInput.dataMapperInput = jsonInputData; apexInput.inputType = 'JSON'; ConnectApi.DataMapperExecuteOptionsRepresentation options = new ConnectApi.DataMapperExecuteOptionsRepresentation(); options.locale = locale; options.shouldSendLegacyResponse = true; apexInput.options = options; ConnectApi.DataMapperExecuteOutputRepresentation output = ConnectApi.OmniDesignerConnect.executeDataMapper(bundleName, apexInput); List<String> innerResponse = output.response; for (String currentResponse : innerResponse) { Map<String, Object> outerMap = (Map<String, Object>) JSON.deserializeUntyped(currentResponse); List<String> keys = new List<String>(outerMap.keySet()); System.debug(outerMap.get('drSObjectResults')); } </pre>	<pre> cess(Map<String, Object> objectList, String bundleName, String locale) </pre>

Upload data with the bulkupload parameter

This method can only call a Data Mapper Load. No other Apex method can pass the bulkUpload parameter to a Data Mapper. See [Omnistudio Data Mapper Calls From Apex](#).

Connect API	Existing Method (Replace with Connect API)
<p>Synchronous Processing:</p> <pre> List < Account > accounts = new List<Account>(); accounts.add(new Account(Name = 'Tes terAccountWithBundleName', vlocity_i ns__Active__c='true')); accounts.add(new Account(Name = 'Tes terAccount2WithBundleName', vlocit y_ins__Active__c='true')); accounts.add(new Account(Name = 'Tes terAccount3WithBundleName', vlocit y_ins__Active__c='true')); accounts.add(new Account(Name = 'Tes terAccount2WithBundleName', vlocit y_ins__Active__c='true')); accounts.add(new Account(Name = 'Test erAccount200WithBundleName', vlocit y_ins__Active__c='true')); List<String> jsonInputData = new Lis t<String>(); jsonInputData.add(JSON.serialize(account s)); ConnectApi.DataMapperExecuteInputRep resentation apexInput = new ConnectA pi.DataMapperExecuteInputRepresentat ion(); apexInput.dataMapperInput = jsonInpu tData; apexInput.inputType = 'JSON'; ConnectApi.DataMapperExecuteOptionsR epresentation options = new ConnectA </pre>	<p>Existing approach for synchronous processing</p> <pre> global static Map<String, Object> pr ocessPost(Map<String, Object> bodyDa ta) </pre> <pre> String bundleName = 'DRWithLoad'; // Data Mapper name String bulkUpload = 'false'; Map<String, Object> bodyData = new Ma p<String, Object>(); bodyData.put('bundleName', bundleNam e); bodyData.put('bulkUpload', bulkUploa d); bodyData.put('objectList', objectLis t); Map<String, Object> result= vlocity_i ns.DRGlobal.processPost(bodyData); System.debug(result); Map<String, Object> result= vlocity_i ns.DRGlobal.processPost(bodyData); System.debug(JSON.serialize(resul t)); </pre>

Connect API	Existing Method (Replace with Connect API)
<pre> pi.DataMapperExecuteOptionsRepresent ation(); options.ignoreCache = false; apexInput.options = options; options.shouldSendLegacyResponse = true; ConnectApi.DataMapperExecuteOutputRe presentation output = ConnectApi.Omn iDesignerConnect.executeDataMapper(b undleName, apexInput); List<String> innerResponse = outpu t.response; Map<String, Object> outerMap = (Ma p<String, Object>) JSON.deserializeU ntyped(innerResponse[0]); List<String> keys = new List<Strin g>(outerMap.keySet()); System.debug(outerMap.get('drSObject Results')); </pre>	

Here's the output:

```

{Account_1={{Id=001xx000003GpXjAAK,
Name=TesterAccountWithBundleName, Up
sertSObjectType=Account, UpsertSucce
ss=true},
{Id=001xx000003GpXkAAK, Name=TesterA
ccount2WithBundleName, UpsertSObject
Type=Account, UpsertSuccess=true},
{Id=001xx000003GpXlAAK, Name=TesterA
ccount3WithBundleName, UpsertSObject
Type=Account, UpsertSuccess=true},
{Id=001xx000003GpXmAAK, Name=TesterA
ccount4WithBundleName, UpsertSObject
Type=Account, UpsertSuccess=true}
..... ) }

```

There is a mismatch in the output during the synchronous processing. Perform these steps to match the exact output.

Connect API	Existing Method (Replace with Connect API)
<pre>Map<String, Object> mapReturn = ne w Map<String, Object>(); mapReturn.put('createdObjectsByTyp e', createdObjectsByType); mapReturn.put('createdObjectsByOrd er', createdObjectsByOrder); if (overrideErrors != null) { mapReturn.put('errors', overrideEr rors); } else { mapReturn.put('errors', errors); } mapReturn.put('errorsByField', err orsByField); mapReturn.put('interfaceInfo', int erfaceInfo); //if (String.isBlank(responseType) responseType == 'SObject') { mapReturn.put('errorsAsJson', erro rsAsJson()); } mapReturn.put('hasErrors', hasAnyE rror); if (responseCache != null && Strin g.isNotBlank(responseTypeCache)) { mapReturn.put('returnResultsData', responseCache); mapReturn.put('responseType', resp onseTypeCache); } else if (responseType != null && r esponseType.equalsIgnoreCase('xml')) { JsonToXml json2xml = new JsonToXm l((Map<String, Object>)toJson(), new Map<String, Object> {</pre>	

Connect API	Existing Method (Replace with Connect API)
<pre> 'fieldWriteOrder' => xmlOutputSequence, 'removeDeclaration' => xmlRemoveDeclaration }); mapReturn.put('responseType', 'XML'); mapReturn.put('returnResultsData', json2xml.getXml()); } else if (responseType != null && responseType == 'Custom') { mapReturn.put('responseType', 'Custom'); mapReturn.put('returnResultsData', processCustomOutputType()); } else { mapReturn.put('responseType', 'JSON'); mapReturn.put('returnResultsData', toJson()); } if (fileAttachmentSyncData.size() > 0) { mapReturn.put('fileAttachmentSyncData', fileAttachmentSyncData); } if (Logger.DebugRecordingMode) { mapReturn.put('debugLog', Logger.recordedDebugStatements); mapReturn.put('CpuTime', Limits.getCPUTime()); mapReturn.put('ActualTime', DateTime.now().getTime() - startForServerTiming); } </pre>	

Connect API	Existing Method (Replace with Connect API)
<pre data-bbox="235 240 507 270">return mapReturn;</pre> <p>Connect API to asynchronously upload data using the 'bulkupload' parameter is not available.</p>	<pre data-bbox="850 409 1428 1110"> String bundleName = 'DRWithLoad'; // Data Mapper name String bulkUpload = 'true'; Map<String, Object> bodyData = new Map<String, Object>(); bodyData.put('bundleName', bundleName); bodyData.put('bulkUpload', bulkUpload); bodyData.put('objectList', objectList); Map<String, Object> result = vlocity_ns.DRGlobal.processPost(bodyData); System.debug(result); </pre>

Process an XML String Using a Specified Data Mapper

Connect API	Existing Method
<pre data-bbox="202 1400 784 1881"> List<String> jsonInputData = new List<String>(); jsonInputData.add(objectList); ConnectApi.DataMapperExecuteInputRepresentation apexInput = new ConnectApi.DataMapperExecuteInputRepresentation(); apexInput.dataMapperInput = jsonInputData; apexInput.inputType = 'XML'; </pre>	<pre data-bbox="850 1400 1428 1516"> static namespace.DRProcessResult processString(String input, String bundleName) </pre>

Connect API	Existing Method
<pre>ConnectApi.DataMapperExecuteOptionsRepresentation options = new ConnectApi.DataMapperExecuteOptionsRepresentation(); options.ignoreCache = false; options.shouldSendLegacyResponse = true; apexInput.options = options; ConnectApi.DataMapperExecuteOutputRepresentation output = ConnectApi.OmnisDesignerConnect.executeDataMapper(bundleName, apexInput); List<String> innerResponse = output.response; Map<String, Object> outerMap = (Map<String, Object>) JSON.deserializeUntyped(innerResponse[0]); List<String> keys = new List<String>(outerMap.keySet()); System.debug(outerMap.get('drSObjectResults'));</pre>	

Security for Omnistudio Data Mappers and Integration Procedures

You can control access to Data Mappers and Integration Procedures using settings that reference Sharing Settings and Sharing Sets or Profiles and Permission Sets.

! **Important** Guest Users, also called anonymous users, cannot access any records by default. Criteria-based Sharing Rules grant them read-only access. This affects all Salesforce orgs. For details, see [Guest User Record Access Development Best Practices](#). Vlocity allows guest users to create and update the records to which [Sharing Rules](#) grant access. No additional configuration is necessary for this expanded access.

You can use Salesforce [Sharing Settings](#) to secure access to Data Mappers and Integration Procedures. If

you use [caching](#), you must set **CheckCachedMetadataRecordSecurity** to true as described here.

You can allow access to a Data Mapper or Integration Procedure based on the Custom Permissions enabled in a user's Salesforce [Profiles](#) or [Permission Sets](#). An Apex class added to your Salesforce Org allows the Vlocity Managed Package to check user Custom Permissions. The custom settings described here are related to this approach. Vlocity recommends using Custom Permissions in Profiles or Permission Sets for ease of use and better performance.

For Salesforce access basics, see [Control Who Sees What, Who Sees What – Overview Video](#), and [Salesforce Data Security Model – Explained Visually](#). For Vlocity-specific information about Profiles, see [Overview of Profiles and Security for Vlocity](#).

Sharing Settings, Sharing Sets, Profiles, and Permission Sets control access to Data Mappers and Integration Procedures as object records.

To ensure field-level security for a Data Mapper, go to the Data Mapper's Options tab and select **Check Field Level Security**. To automatically enforce field-level security for all Data Mappers, enable *EnforceDMFLSAndDataEncryption* in the Omni Interaction Configuration.

If you're using the Omnistudio standard designer, and if the user has the View Encrypted Data permission, the classic encrypted fields are shown in plain text for that user by default. However, if you are using Omnistudio managed package designer, you must enable *EnforceDMFLSAndDataEncryption* in the Omni Interaction Configuration to enforce this behavior.

To enable *EnforceDMFLSAndDataEncryption* in the Omni Interaction Configuration, follow these steps: From Setup, search and open Omni Interaction Configuration. Click **New Omni Interaction Configuration**, enter *EnforceDMFLSAndDataEncryption* for both name and label, set the value to *true*, and save your changes.

 **Warning** When Check Field Level Security and EnforceDMFLSAndDataEncryption are both disabled, Data Mapper Extract and Turbo Extract work differently. If a user doesn't have access to the standard or custom fields, the query fails for Turbo Extract and passes for Extract. For this reason, we recommend that you always use the Check Field Level Security checkbox or the EnforceDMFLSAndDataEncryption Omni Interaction Configuration.

 **Important** A user's access to a Data Mapper or Integration Procedure includes more than the ability to run it directly. Access also applies if an application the user is using [calls](#) the Data Mapper or Integration Procedure. If a user has access to a parent Integration Procedure, the parent can invoke child Integration Procedures and Data Mappers to which the user doesn't have direct access.

[Configure Omnistudio Data Mapper and Integration Procedure Security Settings](#)

You can change settings for Data Mapper and Integration Procedure security in Setup.

[Omnistudio Data Mapper and Integration Procedure Security Settings](#)

These settings affect Data Mapper and Integration Procedure security.

[Syntax of the Required Permission Property](#)

Omnistudio Data Mappers and Integration Procedures have a Required Permission property, which

determines who has runtime access. You can specify roles, profiles, permission sets, custom permissions, or any combination. If Required Permission is blank, any user can run the Data Mapper or Integration Procedure unless the DefaultRequiredPermission property is set.

Implement the `VlocityRequiredPermissionCheck` Class

For the **DefaultRequiredPermission** setting to work, you must implement the `VlocityRequiredPermissionCheck` class manually because Salesforce handles classes in managed and unmanaged packages differently. This class doesn't work properly if it's included in the Vlocity managed package.

Configure Omnistudio Data Mapper and Integration Procedure Security Settings

You can change settings for Data Mapper and Integration Procedure security in Setup.

The settings you configure in this task are listed in [Data Mapper and Integration Procedure Security Settings](#).

These steps apply if you use Omnistudio with standard runtime.

1. Go to Setup.
 - In Lightning Experience, click the gear icon and select **Setup** from the menu.
 - In Salesforce Classic, click the user menu and select **Setup** from the menu.
2. In the **Quick Find** box, type *omni*, then select **Omni Interaction Configuration**.
3. Click **New Omni Interaction Configuration**.
4. In the **Label** and **Value** field, type the name and value of the setting.

For a list of settings, see [Data Mapper and Integration Procedure Security Settings](#).

5. Click **Save**.

Omnistudio Data Mapper and Integration Procedure Security Settings

These settings affect Data Mapper and Integration Procedure security.

To configure these settings, see [Configure Data Mapper and Integration Procedure Security Settings](#) if you use Omnistudio.

Setting	Description	Data Type	Default Value
DefaultRequiredPermission	Specifies the default value for the Required	String	(none)

Setting	Description	Data Type	Default Value
	<p>Permission setting, which determines which users can run Data Mappers and Integration Procedures.</p> <p>The Required Permission setting, which you can optionally specify when creating a Data Mapper or Integration Procedure, overrides this setting.</p> <p>If this setting is absent or blank, all users can run any Data Mappers or Integration Procedures that don't have Required Permission values.</p> <p>The syntax for this setting matches the Required Permission syntax. See Syntax of the Required Permission Property.</p>		
CheckCachedMetadataRecordSecurity	<p>By default, the cached metadata is not secured when Salesforce Sharing Settings or Sharing Sets are used to control access.</p> <p>This setting is disabled by default. If set to true, this setting performs a record-level security check for cached</p>	True or False	False

Setting	Description	Data Type	Default Value
	metadata. This check lessens the performance benefit of metadata caching slightly. This setting isn't needed if you use Custom Permissions to secure access.		
EnforceDMFLSAndDataEncryption	To automatically enforce field-level security for all Data Mappers, enable <i>EnforceDMFLSAndDataEncryption</i> in the Omni Interaction Configuration. See Security for Omnistudio Data Mappers and Integration Procedures .	True or False	(none)

Syntax of the Required Permission Property

Omnistudio Data Mappers and Integration Procedures have a Required Permission property, which determines who has runtime access. You can specify roles, profiles, permission sets, custom permissions, or any combination. If Required Permission is blank, any user can run the Data Mapper or Integration Procedure unless the DefaultRequiredPermission property is set.

When an Integration Procedure calls a Data Mapper or another Integration Procedure, the Required Permission setting of the calling Integration Procedure takes precedence. It overrides the Required Permission setting of the called Data Mapper or Integration Procedure.



Note In Winter '22 and earlier releases of Omnistudio, you can only specify a comma-separated list of custom permission names. For backward compatibility, this simpler syntax is still supported if you only need to specify custom permissions.

The syntax of the Required Permission property is:

```
prefix:name, prefix:name, ...
```

The prefix can be `Role`, `Profile`, `PermSet`, or `CustomPerm`. The name is the name of a role, profile, permission set or permission set group, or custom permission. For example, here's a Required Permission value that includes two roles and one permission set group:

```
Role:Architect,Role:Developer,PermSet:OmniStudio Admin Group
```

Implement the VlocityRequiredPermissionCheck Class

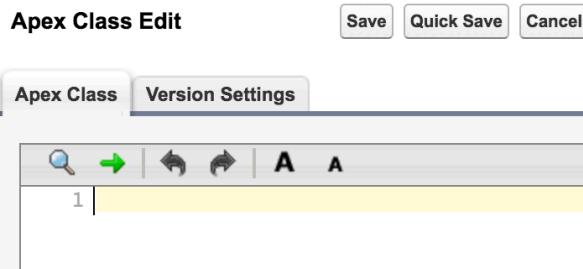
For the **DefaultRequiredPermission** setting to work, you must implement the `VlocityRequiredPermissionCheck` class manually because Salesforce handles classes in managed and unmanaged packages differently. This class doesn't work properly if it's included in the Vlocity managed package.

-  **Note** If you're using Omnistudio Spring '22 or greater, and the **Managed Package Runtime** is set to **Disabled**, you don't have to implement this class.

For **DefaultRequiredPermission** details, see [Omnistudio Data Mapper and Integration Procedure Security Settings](#).

1. From Setup, in the **Quick Find** box, type `apex`.
2. Click **Apex Classes**.
3. Click **New**.

Apex Class



4. Enter the following Apex code in the **Apex Class** tab:

```
global class VlocityRequiredPermissionCheck implements Callable
{
    global Boolean call(String action, Map<String, Object> args)
    {
        if (action == 'checkPermission')
        {
            return checkPermission((String)args.get('requiredPermission'));
        }
        return false;
    }
}
```

```
private Boolean checkPermission(String requiredPermission)
{
    Boolean hasCustomPermission = false;
    List<String> customPermissionsName = requiredPermission.split(',');
    for (String permissionName : customPermissionsName)
    {
        Boolean hasPermission = FeatureManagement.checkPermission(permissionName.normalizeSpace());
        if (hasPermission == true)
        {
            hasCustomPermission = true;
            break;
        }
    }
    return hasCustomPermission;
}
```

5. Click **Save**.

Preview and Test Data Mappers

You can preview and test the input and output of a Data Mapper in the Data Mapper Designer.

1. In your Data Mapper, click **Preview**.
2. Click **Edit as Params**.
3. In the Input Parameters panel, add a key-value pair.

Alternatively, you can specify the input in JSON format.

4. To observe the effects of caching when testing the Data Mapper Load, deselect **Ignore Cache**.
See [Improving Data Mapper Performance with Caching](#).
5. Click **Execute**.

Load Results – The Objects Created panel lists the resulting objects, which are saved permanently. The Debug Log panel displays the results of the Salesforce queries issued by the Data Mapper.

Extract Results – The Response panel displays the resulting data, and the Debug Log panel shows the results of the Salesforce queries issued by the Data Mapper. For example, if the Data Mapper filters Accounts based on an Id provided by an Omniscript, a key-value pair lets you test the Data Mapper using a specific Account Id.

Extract Turbo Results – The Response panel displays the resulting data, and the Debug Log panel displays the results of the Salesforce queries issued by the Data Mapper. For example, if the Data Mapper filters

accounts based on an ID, a key-value pair lets you test the Data Mapper using a specific Account Id.

Transform Results – The Response panel displays the resulting data, and the Debug Log panel displays the results of the Salesforce queries issued by the Data Mapper.

-  **Note** The Data Mapper Transform removes trailing zeros from decimal values when displayed in the Response panel. For example, when you enter 1,234.50 in the Input Parameters panel, the result shows as 1,234.5 in the Response panel after processing.

Examples of Omnistudio Data Mappers

For each type of Data Mapper, examples show how they're set up for specific, common uses. Each example provides sample JSON for input. Each shows the setup for input, output, mapping, query, and formula, as they apply to the Data Mapper type and the use case.

Omnistudio Data Mapper Extract Examples

Data Mapper Extract examples demonstrate object relationships, paging with data values, paging with offsets, and relationship notation.

Omnistudio Data Mapper Load Examples

Data Mapper Load examples demonstrate formulas and object relationships.

Omnistudio Data Mapper Transform Examples

Data Mapper Transform examples demonstrate how to convert a single item into a list with one item and how to convert a list of objects to a list of values.

Omnistudio Data Mapper Extract Examples

Data Mapper Extract examples demonstrate object relationships, paging with data values, paging with offsets, and relationship notation.

For additional examples that show how you can improve Data Mapper performance for some use cases using relationship notation, see [Relationship Notation versus Multiple Extract Steps](#).

Extract Fields from the Account Object Example

In this example, a Data Mapper extracts an account ID parameter and returns the account name so that an Omniscript can use the extracted account data.

Extract Data from Three Related Objects

Create an Omnistudio Data Mapper Extract for use by a Case-handling Omniscript. The agents who handle Cases can look up an Account, list all the Cases for the Account, and find the Contact for each Case.

Use Data Values and Page Through Sorted Data

If an Omnistudio Data Mapper Extract is expected to retrieve numerous records, you can use *paging* to retrieve a few at a time based on field values. For example, you can page through Accounts by Id.

Page Through Sorted Data Using Offsets

If an Omnistudio Data Mapper Extract is expected to retrieve numerous records, you can use *paging* to retrieve a few at a time based on OFFSET values. For example, you can page through Contacts by last name.

Extract Fields from the Account Object Example

In this example, a Data Mapper extracts an account ID parameter and returns the account name so that an Omniscript can use the extracted account data.

Create Settings

Field	value	notes
Interface Type	Extract	
Input Type	JSON	If using with another Omnistudio component such as an Omniscript, set the input as JSON.
Output Type	JSON	

Extract Settings

Map the extracted object with the path in the JSON, and create a filter for the ID.

Field	Value	notes
Extraction Object	Account	
Extract Object Path	Account	
Filter	Id = InputID	

Mapping Settings

Map the input data to a JSON node so that an Omniscript (or other component) can access the data. To define a data transformation, you can add key-value pairs. This addition replaces the defined key with the value in the output.

Extract JSON Path	JSON Output Path	notes
Account:Name	Account:Name	Choose a data type for the data type, such as string. If needed,

Extract JSON Path	JSON Output Path	notes
		you can set a default value.

Extract Preview

In Preview, add InputId as the key and a valid Salesforce Account object ID as the value, such as 0016100001OBpRi.

After you execute the sample Data Mapper in Preview, the desired output is shown, such as this example.

```
{
  "Account": [
    {
      "Name": "Smith Incorporated"
    }
  ]
}
```

Extract Data from Three Related Objects

Create an Omnistudio Data Mapper Extract for use by a Case-handling Omniscript. The agents who handle Cases can look up an Account, list all the Cases for the Account, and find the Contact for each Case.

In this example, it's important to specify the complete object hierarchy in Extraction Object Path, filters, and Extract JSON Path values. Accounts and Contacts are directly related and these objects are related through Cases. This example uses the latter relationship. The object hierarchy tells the Data Mapper which relationship to use.

Create Settings

Field	value	notes
Interface Type	Extract	
Input Type	JSON	If using with another Omnistudio component such as an Omniscript, set the input as JSON.
Output Type	JSON	

Extract Settings

Map the extracted objects with the paths in the JSON, and create a filter for the ID.

Extraction Object	Extract Object Path	Filter
Account	Account	Id = AccountId
Case	Account:Case	AccountId = Account:Id
Contact	Account:Case:Contact	Id = Account:Case:ContactId

Mapping Settings

Map the input fields from the extraction step JSON to the output JSON so that an Omniscript (or other component) can access the data. To define a data transformation, you can add key-value pairs. This addition replaces the defined key with the value in the output.

Extract JSON Path	JSON Output Path	notes
Account:Name	Account:Name	Choose a data type for the data type, such as string. If needed, you can set a default value.
Account:Case:CaseNumber	Account:Case:CaseNumber	
Account:Case:Subject	Account:Case:Subject	
Account:Case:Contact:Name	Account:Case:Contact:Name	

Verify the Current JSON Output panel displays a desired JSON structure, as in this example.

```
{
    "Account": {
        "Case": {
            "CaseNumber": "Text",
            "Contact": {
                "Name": "Text"
            },
            "Subject": "Text"
        },
        "Name": "Text"
    }
}
```

The order of fields is not important, but the structure is.

Extract Preview

In Preview, add AccountId as the key and a valid Salesforce Account object ID as the value, such as 0015e000003XIkMAAS.

After you execute the sample Data Mapper in Preview, the desired output with the associated cases is shown, such as this example.

```
{
    "Account": {
        "Case": [
            {
                "Contact": {
                    "Name": "Edward Stamos",
                    "Subject": "Cannot track our order",
                    "CaseNumber": "00001004"
                },
                {
                    "Contact": {
                        "Name": "Leanne Tomlin",
                        "Subject": "Wrong size widgets",
                        "CaseNumber": "00001003"
                    },
                    {
                        "Contact": {
                            "Name": "Jeff Hunt",
                            "Subject": "Update account phone number",
                            "CaseNumber": "00001005"
                        },
                        {
                            "Contact": {
                                "Name": "Edward Stamos",
                                "Subject": "Billing status",
                                "CaseNumber": "00001006"
                            }
                        ],
                        "Name": "Acme"
                    }
                }
            }
        ]
    }
}
```

Use Data Values and Page Through Sorted Data

If an Omnistudio Data Mapper Extract is expected to retrieve numerous records, you can use *paging* to retrieve a few at a time based on field values. For example, you can page through Accounts by Id.

Paging through results requires creating the ORDER BY and LIMIT settings for extraction.

- For ORDER BY, set to sort the data based on a specific field.
- For LIMIT, specify how many records to retrieve at a time.
- After retrieving each set of records, use the sort field value from the last record retrieved to retrieve the next set.

Create Settings

Field	value	notes
Interface Type	Extract	
Input Type	JSON	If using with another Omnistudio component such as an Omniscript, set the input as JSON.
Output Type	JSON	

Extract Settings

Map the extracted object with the path in the JSON, and create a filter for the ID.

Extraction Object	Extract Object Path	Filter
Account	Account	Id = lastAccountId

For ORDER BY, and enter `Id` as its value. Then, choose LIMIT and add a value of `2`.

Mapping Settings

Map the input data to a JSON node so that an Omniscript (or other component) can access the data. To define a data transformation, you can add key-value pairs. This addition replaces the defined key with the value in the output.

Extract JSON Path	JSON Output Path
account:id	id
Account:Name	name

Extract Preview

In Preview, add lastAccountId as the key and a valid Salesforce Account object ID as the value, such as 0016100001OBpRi.

After you execute the sample Data Mapper in Preview, the first page of Account records is shown. From the second record in the response, copy the Id value, and paste it in the Value field in the Input Parameters pane, and execute the sample. The second page of Account records is retrieved

Page Through Sorted Data Using Offsets

If an Omnistudio Data Mapper Extract is expected to retrieve numerous records, you can use *paging* to retrieve a few at a time based on OFFSET values. For example, you can page through Contacts by last name.

Paging through results requires creating the ORDER BY, OFFSET, and LIMIT settings for extraction.

- For LIMIT, specify how many records to retrieve at a time.
- For OFFSET, specify a multiple of the LIMIT value.
- Optional but recommended for predictable results: Use the ORDER BY setting to sort the data based on a specific field.

Create Settings

Field	value	notes
Interface Type	Extract	
Input Type	JSON	If using with another Omnistudio component such as an Omniscript, set the input as JSON.
Output Type	JSON	

Extract Settings

Map the extracted object with the path in the JSON. To show how you can filter results based on a field, this example lists only contacts who can be called.

Extraction Object	Extract Object Path	Filter
Contact	Contact	DoNotCall = "false"

For **ORDER BY**, enter `Lastname` as its value. Then, choose **LIMIT** and add a value of `3`. Finally, choose **OFFSET**, and enter `AfterRecord` as its value.

Mapping Settings

Map the input data to a JSON node so that an Omniscript (or other component) can access the data. To define a data transformation, you can add key-value pairs. This addition replaces the defined key with the value in the output.

Extract JSON Path	JSON Output Path
contact:FirstName	FirstName
contact:LastName	LastName
contact:Phone	Phone

Extract Preview

In Preview, add AfterRecord as the key and 0 as the value, such as 0016100001OBpRi.

After you execute the sample Data Mapper in Preview, the first page of Contact records is shown. Change the AfterRecord value to 3, and execute the sample. The second page of Contact records is retrieved.

Omnistudio Data Mapper Load Examples

Data Mapper Load examples demonstrate formulas and object relationships.

Create a Contact and Use a Formula

A common task for an Omnistudio Data Mapper Load is to create a record in a Salesforce object. Depending on the source data, the input might need light cleanup or a check for a particular field status. In this example, the Data Mapper load creates a record in the Contact object and sets the Authorized field to true if the contact is over 18 years old.

Create a Contact for an Existing Account

A common task for an Omnistudio Data Mapper Load is to create a record in a Salesforce object that's linked to another object. In this example, a Data Mapper Load creates a record in a Contact object. A link to an Account object record with a specific Id ensures that the new Contact is related to that Account.

Create a Contact and Use a Formula

A common task for an Omnistudio Data Mapper Load is to create a record in a Salesforce object. Depending on the source data, the input might need light cleanup or a check for a particular field status. In this example, the Data Mapper load creates a record in the Contact object and sets the Authorized field to true if the contact is over 18 years old.

The input JSON for this example includes the contact details, and the data is in expected formats. But the names aren't the same as the Contact fields. In this scenario, a custom checkbox field named Authorized is added to the Contact object as Authorized__c and to the Contact layout.

```
{  
    "ContactDetails": {  
        "Birthdate": "10/10/1954",  
        "LastName": "Singh",  
        "Telephone": "5106345789",  
        "FirstName": "Sanjay"  
    }  
}
```

Create Settings

Field	value	notes
Interface Type	Load	
Input Type	JSON	In this example, the input is provided as JSON and not pulled from another source.
Output Type	sObject	

Object Settings

Click + and select Contact.

Mapping Settings

Map data from the input JSON to the target Salesforce object and field. If you're using the designer on a managed package, click **Fields**. See [Configure an Omnistudio Data Mapper Load](#).

Input JSON Path	Domain Object Field
ContactDetails:Authorized	Authorized__c
ContactDetails:Birthdate	Birthdate
ContactDetails:FirstName	FirstName
ContactDetails:LastName	LastName
ContactDetails:Telephone	Phone

Formula Settings

Click **Formula**, and define a formula for the Data Mapper Load. See [Use Formulas in Omnistudio Data Mappers](#).

- **Formula:** `IF(AGE(ContactDetails:Birthdate) > 18, "true", "false")`
- **Formula Result Path:** `ContactDetails:Authorized`

Load Preview

In Preview, enter the sample JSON as input.

```
{
  "ContactDetails": {
    "Birthdate": "10/10/1954",
    "LastName": "MyLastName",
    "Telephone": "5106345789",
    "FirstName": "MyFirstName"
  }
}
```

After you execute the sample Data Mapper in Preview, a link to the object is displayed in the Objects Created pane. Click the link and verify that the Authorized checkbox is checked only if the contact is over 18 years old. (The field might be visible only in edit mode.)

Create a Contact for an Existing Account

A common task for an Omnistudio Data Mapper Load is to create a record in a Salesforce object that's linked to another object. In this example, a Data Mapper Load creates a record in a Contact object. A link

to an Account object record with a specific Id ensures that the new Contact is related to that Account.

The input JSON for this example includes the the Account Id and contact name, and the data is in expected formats. But the names aren't the same as the Contact fields.

```
{
    "AccountId": "0016100001BKL4uAAH",
    "Name": {
        "First": "Jane",
        "Last": "Doe"
    }
}
```

Create Settings

Field	value	notes
Interface Type	Load	
Input Type	JSON	In this example, the input is provided as JSON and not pulled from another source.
Output Type	sObject	

Object Settings

Click + and select Account and then Contact.

For Contact, add a link with these settings:

- **Domain Object Field:** AccountId
- **Linked Object:** 1 - Account
- **Linked Object Field (after the period):** Id

Mapping Settings

Map data from the input JSON to the target Salesforce objects and fields. See [Configure an Omnistudio Data Mapper Load](#).

For Account, create these mappings.

- **Input JSON Path:** AccountId
- **Domain Object Field:** Id
- **Upsert Key:** checked

For Contact, create these mappings. The AccountId mapping is already present.

Input JSON path	Domain Object Field
Name:First	FirstName
Name:Last	LastName

Load Preview

In Preview, enter the sample JSON as input, and replace the Account Id with one from your org.

After you execute the sample Data Mapper in Preview, links to the updated Account and new Contact objects are displayed in the Objects Created pane. Click the links to verify.

Omnistudio Data Mapper Transform Examples

Data Mapper Transform examples demonstrate how to convert a single item into a list with one item and how to convert a list of objects to a list of values.

For additional Data Mapper Transform examples, see [Transforming Lists with Data Mappers](#) and [Use Formulas in Omnistudio Data Mappers](#).

[Convert a Single Item to a List with One Item](#)

Sometimes, a component requires input as a list, but the input isn't a list. You can use the Output Data Type setting in an Omnistudio Data Mapper Transform to convert JSON data to a list. Use a formula to remove the second level of the hierarchy.

[Convert a List of Objects to a List of Values](#)

You can convert a list of JSON objects (key-value pairs in braces) to a list of single values, regardless of the length of the list. The trick is to use a formula to remove the second level of the hierarchy.

[Change the Hierarchical Level of a List](#)

If a list is defined at the wrong level of a JSON node hierarchy, you can use an Omnistudio Data Mapper Transform to change the level at which the list is defined. This approach only works for a short list with a known number of items.

Convert a Single Item to a List with One Item

Sometimes, a component requires input as a list, but the input isn't a list. You can use the Output Data Type setting in an Omnistudio Data Mapper Transform to convert JSON data to a list. Use a formula to remove the second level of the hierarchy.

Create Settings

Field	value	notes
Interface Type	Transform	
Input Type	JSON	In this example, the input is provided as JSON and not pulled from another source.
Output Type	JSON	

Transform Settings

Field	value	notes
Input JSON Path	Contact	You only need to create a mapping for the top-level node.
Output JSON Path	Contact	
Output Data Type	List<Map>	The Apex key-value pair converts to a list.

Transformation Preview

In Preview, enter the sample JSON as input.

```
{
    "Contact": {
        "Id": "0036100000423z8AAA",
        "FirstName": "Amy",
        "LastName": "Smith"
    }
}
```

After you execute the sample Data Mapper in Preview, the input map is transformed into a list, as indicated by the square brackets.

```
{
    "Contacts": [
        {
            "LastName": "Smith",
            "FirstName": "Amy",
            "MiddleName": "M"
        }
    ]
}
```

```

    "Id": "0036100000423z8AAA"
}
]
}

```

Convert a List of Objects to a List of Values

You can convert a list of JSON objects (key-value pairs in braces) to a list of single values, regardless of the length of the list. The trick is to use a formula to remove the second level of the hierarchy.

Create Settings

Field	value	notes
Interface Type	Transform	
Input Type	JSON	In this example, the input is provided as JSON and not pulled from another source.
Output Type	JSON	

Formula Settings

Field	value	notes
Formula	ObjectList:Property	
Formula Result Path	FormulaList	

Transform Settings

In the Manage Input/Output Type window, use these values.

Field	value	notes
Input JSON	<pre>{ "ObjectList": [{ "Property": "P 9034" }] }</pre>	

Field	value	notes
]	
Expected JSON Output	{ "PropertyValueList": ["P9034"] }	

Then, create a mapping for the top-level nodes.

Field	value	notes
Input JSON Path	FormulaList	
Output JSON Path	PropertyValueList	

If you check the **Current JSON Output** view, the content is similar to this example:

```
{
  "PropertyValueList": "Text"
}
```

Transformation Preview

In Preview, enter the sample JSON as input.

```
{
  "ObjectList": [
    {
      "Property": "P9034"
    },
    {
      "Property": "P6538"
    },
    {
      "Property": "P9034"
    }
  ]
}
```

```
        "Property": "P1234"  
    },  
    {  
        "Property": "P5678"  
    },  
    {  
        "Property": "P2345"  
    },  
    {  
        "Property": "P7654"  
    }  
]  
}
```

After you execute the sample Data Mapper in Preview, the input is transformed into a list, as indicated by the square brackets.

```
{  
    "PropertyValueList": [  
        "P9034",  
        "P6538",  
        "P1234",  
        "P5678",  
        "P2345",  
        "P7654"  
    ]  
}
```

Add or subtract list items in the JSON data in the Input pane and click **Execute** again. The output changes accordingly.

Change the Hierarchical Level of a List

If a list is defined at the wrong level of a JSON node hierarchy, you can use an Omnistudio Data Mapper Transform to change the level at which the list is defined. This approach only works for a short list with a known number of items.

This example describes how to create a Data Mapper Transform that converts a list defined at `Contacts:Name` to a list defined at `Contacts`.

Create Settings

Field	value	notes
Interface Type	Transform	
Input Type	JSON	In this example, the input is provided as JSON and not pulled from another source.
Output Type	JSON	

Transform Settings

In the Manage Input/Output Type window, use these values.

Input JSON

```
{
  "Contacts": {
    "Name": [
      {
        "First": "John",
        "Last": "Doe"
      },
      {
        "First": "June",
        "Last": "Doe"
      }
    ]
  }
}
```

Expected JSON Output

```
{
  "Contacts": {
    "Name": [
      {
        "First": "John",
        "Last": "Doe"
      },
      {
        "First": "June",
        "Last": "Doe"
      }
    ]
  }
}
```

```

        "Last": "Doe"
    }
]
}
}

```

Click **Quick Match**, and in the Quick Match dialog, click **Auto Match** and save.

To add list position indexes, edit the mappings.

Field	value	notes
Input JSON Path	Contacts:Name 1:First	
Output JSON Path	Contacts 1:Name:First	

Click **Quick Match** again, click **Auto Match** and save. Edit the new mappings to add list position indexes.

Field	value	notes
Input JSON Path	Contacts:Name 2:First	
Output JSON Path	Contacts 2:Name:First	

Transformation Preview

In Preview, the input is similar to the sample JSON.

```

{
  "Contacts": {
    "Name": [
      {
        "First": "John",
        "Last": "Doe"
      },
      {
        "First": "June",
        "Last": "Doe"
      }
    ]
  }
}

```

After you execute the sample Data Mapper in Preview, the input map is transformed into a list, as

indicated by the square brackets.

```
{  
  "Contacts": [  
    {  
      "Name": {  
        "First": "John",  
        "Last": "Doe"  
      }  
    },  
    {  
      "Name": {  
        "First": "June",  
        "Last": "Doe"  
      }  
    }  
  ]  
}
```

Omnistudio

Use Omnistudio Integration Procedures to declaratively automate data interactions between Salesforce and external third-party applications. Integration Procedures handle complex data transformations, API calls, and event-driven automation, and they can execute multiple actions in a single server call.

Use Integration Procedures when no user interaction is required during the execution, and you want to:

- Retrieve, transform, and send data between Salesforce and external systems
- Offload processing to the server to improve performance and scalability
- Bundle multiple operations in a single server transaction
- Enable data caching for frequently accessed information

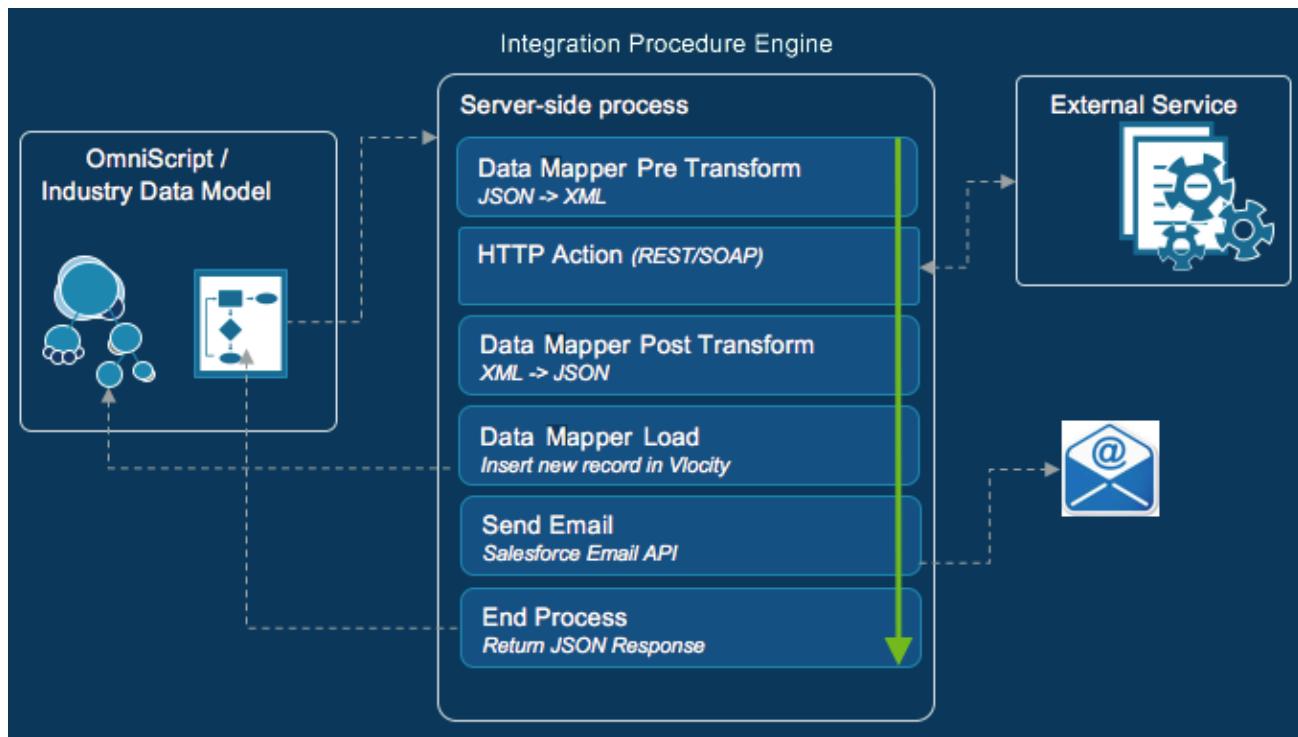
A single Integration Procedure can retrieve data from multiple sources, transform the data, and return a single payload with the transformed data. For improved performance, you can cache frequently accessed data, including the response data for an entire Integration Procedure or for individual Integration Procedure steps.

Integration Procedures can read and write data from Salesforce and from external systems, make REST API calls, and call Apex code. You can call an Integration Procedure from an Omniscript, a Flexcard, other Integration Procedures, a REST API, a Flow, or an Apex method.

The most important building blocks of Integration Procedures are *actions* and *blocks*.

- Actions perform tasks like reading data, making HTTP requests, or sending email.
- Blocks group actions so you can configure their behavior together.

This diagram shows the relationship between an Omniscript and an Integration Procedure that it calls.



An Integration Procedure can also use variables to store and manipulate data within itself, and Data Mappers to translate data between different systems or formats.

Work with Integration Procedures

Use the Integration Procedure Designer to build and configure server-side processes that automate data operations between Salesforce and external systems. Integration Procedures support actions like API calls, data transformations, conditional logic, and chaining in a single server call. The Designer also includes tools for debugging and performance optimization, making it ideal for use with Omniscripts, Flexcards, and external integrations.

Build an Integration Procedure

To build an Integration Procedure, define its steps and logic by using a combination of Blocks and Actions.

Integration Procedures on Agentforce

Integration Procedures in Omnistudio are powerful tools used to automate data integration and transfer between systems. They enable the creation of complex workflows, data transformations, and business logic execution, all within a single, unified environment. Agentforce is Salesforce's agent-focused workspace, designed to streamline customer service operations by providing a centralized platform for case management, customer interaction, and system integration.

Invoke an Integration Procedure

You can invoke Integration Procedures from other Omnistudio tools such as Omniscripts and Cards. You can also invoke Integration Procedures from Apex classes, batch jobs, REST APIs, or Salesforce flows.

Long-Running Integration Procedures

When invoking long-running Integration Procedures you can avoid hitting Salesforce governor limits by using chainable and queueable chainable settings or Apex Continuations, or by chaining on one or

more specific long-running steps.

Security for Omnistudio Data Mappers and Integration Procedures

You can control access to Data Mappers and Integration Procedures using settings that reference Sharing Settings and Sharing Sets or Profiles and Permission Sets.

Test Procedures: Integration Procedures for Unit Testing

An Integration Procedure that performs a unit test is a *Test Procedure*. You can use a Test Procedure to unit test almost anything an Integration Procedure can invoke, such as an Omnistudio Data Mapper, a Calculation Matrix, an Apex class, or another Integration Procedure.

Integration Procedure Actions

To compose an Integration Procedure, you add actions that run sequentially. These actions can set data values, perform functions, call Omnistudio Data Mappers, invoke Apex classes, send emails, invoke REST endpoints, run other Integration Procedures, and more.

Integration Procedure Blocks

In an Omnistudio Integration Procedure, you can run a group of related steps as a unit inside a block to execute them conditionally, cache them, repeat them for each item in a list, or return an error if they fail.

Work with Integration Procedures

Use the Integration Procedure Designer to build and configure server-side processes that automate data operations between Salesforce and external systems. Integration Procedures support actions like API calls, data transformations, conditional logic, and chaining in a single server call. The Designer also includes tools for debugging and performance optimization, making it ideal for use with Omniscripts, Flexcards, and external integrations.

From the App Launcher, open the Integration Procedures app. Click New to create an Integration Procedure, or click an existing Integration Procedure from the list to edit it.

You must provide a Name, Type, and Subtype to uniquely identify the Integration Procedure, then save your changes, before you can continue working on it.

After you create and configure your Integration Procedure, you can preview and test it in the Preview pane. Add input parameters and then click Execute to run the procedure. When the Integration Procedure finishes, you can see the response output, debug log, execution sequence, any errors, and other detailed information to help you debug and refine your Integration Procedure.

When the Integration Procedure works as expected, activate it to make it available to others. As you continue to develop and refine an Integration Procedure, you can save new versions to preserve earlier versions of your work. Only one version can be active at a time. When you activate a version, Omnistudio deactivates all other versions of the Integration Procedure.

Export or Import an Omnistudio Integration Procedure

Export or import Integration Procedures to use them in another org, such as when you create them in a sandbox. When exporting or importing data packs for Integration Procedures, if you encounter Apex limit errors such as heap or CPU limits, reduce the size of the data pack by unselecting dependencies during the export process. Additionally, break the data pack into multiple smaller data packs. As a best practice, we recommend including no more than 10 elements in each data pack.

Export or Import an Omnistudio Integration Procedure

Export or import Integration Procedures to use them in another org, such as when you create them in a sandbox. When exporting or importing data packs for Integration Procedures, if you encounter Apex limit errors such as heap or CPU limits, reduce the size of the data pack by unselecting dependencies during the export process. Additionally, break the data pack into multiple smaller data packs. As a best practice, we recommend including no more than 10 elements in each data pack.

Export an Integration Procedure

You can export an Integration Procedure by using the Salesforce CLI.

Before you begin:

- [Set up Salesforce CLI](#)
- If you want to export Integration Procedures that call associated Data Mappers, see [Export or Import Omnistudio Data Mappers](#) and complete the steps.

1. From a terminal in VS Code, enter this command, and replace the variables with values for your org.

```
sfdx force:data:tree:export -q "SELECT IsMetadataCacheDisabled, IsTestProcedure, Description, OverrideKey, Name, OmniProcessKey, Language, PropertySetConfig, LastPreviewPage, OmniProcessType, ElementTypeComponentMapping, SubType, ResponseCacheType, IsOmniScriptEmbeddable, CustomJavaScript, IsIntegrationProcedure, VersionNumber, DesignerCustomizationType, Namespace, Type, RequiredPermission, WebComponentKey, IsWebCompEnabled, (SELECT Description, DesignerCustomizationType, Name, EmbeddedOmniScriptKey, IsActive, Type, ParentElementId, PropertySetConfig, SequenceNumber, Level, Id from OmniProcessElement s) from OmniProcess where OmniProcessType='Integration Procedure' AND id='integration_procedure_id'" -u org_alias -p -d export_directory -p
```

org_alias

The org from where you're exporting the Integration Procedure.

export_directory

The directory in the project where the exported JSON file is stored.

integration_procedure_id

The ID of the Integration Procedure that you want to export.



Note Don't include `isActive=true` on OmniProcess when you export the Integration Procedure.

2. Verify whether the file in JSON format is downloaded in the directory.
3. Run the import command.

Import an Integration Procedure

You can import an Integration Procedure by using the Salesforce CLI.

Before you begin, [Set Up Salesforce CLI](#).

From a terminal in VS Code, enter this command, and replace `org_alias` with the org from where you're importing the Integration Procedure.

```
sfdx force:data:tree:import -p ./export_directory/OmniProcess-OmniProcessElement-plan.json -u {org_alias}
```

org_alias

The alias of the org from where you're exporting.

export_directory

The directory in the project where the exported JSON is stored.

 **Note** If the exported JSON file has `isActive=true`, the import operation fails.

Export an Integration Procedure from the Designer on a Managed Package

If you're using the designer on a managed package, you can export an Integration Procedure from the Integration Procedure Home page.

1. From the App Launcher, find and select **Omnistudio Integration Procedures**.
2. Select the version of the Integration Procedure that you want to export.
3. Click .
4. Select the item to export, and click **Next**.
5. Review the item to export, and click **Next**.
6. If needed, update the Name and Description.
7. To store the data pack and access it later from the Omnistudio DataPacks tab, select **Add to Library**.
8. To download the data pack to your computer, select **Download**.
9. Click **Done**.

Import an Integration Procedure to the Designer on a Managed Package

If you're using the designer on a managed package, you can import an Integration Procedure from the Integration Procedure Home page.

1. From the App Launcher, find and select **Omnistudio Integration Procedures**.
2. Click **Import**.
3. Click **Browse**, select a data pack from your computer that you want to upload, and click **Open**.
4. Click **Next**.
5. Select the item to import, and click **Next**.
6. Review the item to import.
7. Click **Next** and then **Done**.

Build an Integration Procedure

To build an Integration Procedure, define its steps and logic by using a combination of Blocks and Actions.

Actions perform specific tasks such as sending email, reading and writing Salesforce data, and running batch jobs.

Blocks allow you to group and control multiple Actions. For example, use Blocks to cache the output, or provide error handling, for the Actions in the Block. You can nest blocks within other blocks. For example, you can nest a Loop Block within a Try-Catch Block or a Cache Block.

You can control a lot about the Integration Procedure as a whole in the Procedure Configuration under the Properties tab. For example:

- Define which data is returned in the response JSON.
- Set the required permissions to run the procedure.
- Track custom data about the performance of the procedure.
- Split the Integration Procedure across multiple transactions to avoid hitting Salesforce governor limits.
- Improve performance by caching data.
- Configure the procedure as a Test Procedure to validate components such as OmniStudio Data Mappers, Apex classes, and other Integration Procedures.

Like the Integration Procedure itself, Actions and Blocks can be configured by using fields under the Properties tab. Actions and Blocks access objects, lists, and other data by name. For example, to use a Loop Block to process items in a list, you enter the name of the list in the Loop List parameter. All Actions and Blocks include a field called Execution Conditional Formula, which you can use to execute the Action or Block only in specific circumstances. One type of Block, the Conditional Block, provides more sophisticated conditional control over a group of Actions.

In the Required Permission field, define roles, profiles, permission sets, or custom permissions.

To write the results of each action to the root level of the Data JSON, you can select Include All Actions in Response.

 **Note** When you enable the Include All Actions in Response setting, any Loop Block in the Integration Procedure can't process an array of records. This occurs because the `stepResultFinal` array exceeds a size of 1 and results in an error: `Cannot Write Array to Root`.

At the end of an Integration Procedure, you can use a [Response Action for Integration Procedures](#) to trim the data and only return what is needed. For specific trimming strategies, see [Manipulate JSON with the Send/Response Transformations Properties](#). To allow an Integration Procedure to exit early under appropriate conditions, use multiple Response Actions with different Execution Conditional Formulas.

By default, all the actions in an Integration Procedure run in a single transaction. If the transaction exceeds a Salesforce governor limit, Salesforce ends the transaction and the Integration Procedure fails. You can't set a limit that exceeds the maximum imposed by Salesforce. For details about governor limits, see the relevant Salesforce topic.

[Access Integration Procedure Data JSON with Merge Fields](#)

Enable Integration Procedure elements and element properties to access data JSON using merge fields. A merge field is a variable that references the value of a JSON node. For example, if a JSON node is `"FirstName": "John"`, then the merge field `%FirstName%` returns `John`.

Environment Variables in Omnistudio Data Mappers and Integration Procedures

You can use environment variables to define Default Values and Filter Values, and in Formulas.

External Objects in Integration Procedures

Integration Procedures support Salesforce External Objects. The external data can come from any source that can be accessed using a REST endpoint. The returned data can be rendered as JSON or XML.

Work with Data and Lists

Use Integration Procedures to read, write, and transform complex, hierarchical data from Salesforce objects and other sources. Use actions to work with data, and blocks to process it.

Define Execution Logic

Define conditional flow in Integration Procedures by using the Execution Conditional Formula field on an action or block, or by using a Conditional Block to compose a series of mutually exclusive conditions.

Send Email from an Integration Procedure

To send email from an Integration Procedure, use an Email Action. You can either specify all the email field values or use a Salesforce email template.

Make an HTTP Call from an Integration Procedure

This Integration Procedure example retrieves the astronomy picture of the day from NASA using NASA's publicly available REST API.

Call an Apex Class or an Invocable Action from an Integration Procedure

To call an Apex class and method, or an invocable action, use a Remote Action and pass in invocation options and data.

Improve Performance by Using Caching

Omnistudio offers multiple caching options to improve performance and minimize unnecessary data processing in Integration Procedures. The two primary user-configurable caching types are Org Cache and Session Cache, which are available in Integration Procedure designer. These caching mechanisms are part of the Salesforce Platform Cache and are implemented through Scale Cache in Omnistudio with standard runtime. Each serves a distinct purpose and comes with specific configuration considerations.

Handle Errors in Integration Procedures

You can configure the conditions for success or failure of an Integration Procedure action or group of actions.

Integration Procedure Best Practices

To maximize the benefits of Integration Procedures, follow the best practices whenever possible.

See Also

- [Integration Procedure Event Tracking](#)
- [Syntax of the Required Permission Property](#)
- [Settings for Long-Running Integration Procedures](#)
- [Improve Performance by Using Caching](#)
- [Integration Procedure Actions](#)
- [Integration Procedure Blocks](#)

Access Integration Procedure Data JSON with Merge Fields

Enable Integration Procedure elements and element properties to access data JSON using merge fields.

A merge field is a variable that references the value of a JSON node. For example, if a JSON node is `"FirstName": "John"`, then the merge field `%FirstName%` returns `John`.

Merge fields access data JSON using syntax to indicate to an Integration Procedure element that a merge field is present. The syntax requires you to wrap a full JSON path with a percent sign on both ends.

 **Note** Only certain element properties support merge fields. Text between two percent (%) signs in a Set Values formula or text field is treated as a merge field. To represent literal percent (%) signs, use the `$V{velocity.Percent}` environment variable.

Common Use Cases

- Setting values to rename elements, access JSON nodes, run formulas, and populate elements. See [Set Values Action for Integration Procedures](#).
- Access data stored in Integration Procedure elements in a formula or in a future step.
- Access data returned from an action. For example, an Omnistudio Data Mapper Action or an HTTP Action returns data from Salesforce or an external source. See [Integration Procedure Actions](#).

Access the Data JSON

Use existing data JSON in an element property by indicating the use of a merge field.

1. Locate the name of the JSON node in the Integration Procedure's data JSON.
2. Enter the name of the JSON node and wrap the name in percentage signs to indicate it's a merge field. For example, a merge field accessing a JSON node named `firstName` must use the syntax `%firstName%`.
3. Preview the Integration Procedure to ensure the merge field works correctly.

Additional Syntax

This table provides additional syntax examples for nested JSON.

JSON Node	Merge Field Syntax Example
<pre>"ContactInfo": { "FirstName": "John" }</pre>	Use a colon symbol <code>:</code> to access a nested JSON node.

JSON Node	Merge Field Syntax Example
<pre data-bbox="220 390 652 897"> "ContactInfo": { "ContactInfoList": [{ "FirstName": "John" }, { "FirstName": "Adam" }, { "FirstName": "Steve" }] } </pre>	<p data-bbox="825 401 1395 475">Use <code> n</code> to access a specific node in a list. This merge field returns <code>Steve</code>:</p> <div data-bbox="840 538 1428 601" style="border: 1px solid #ccc; padding: 5px; width: fit-content;"> <code>%ContactInfo:ContactInfoList 3:FirstName%</code> </div>

Environment Variables in Omnistudio Data Mappers and Integration Procedures

You can use environment variables to define Default Values and Filter Values, and in Formulas.

For example, the filter `$Vlocity.N_DAYS_AGO:30` extracts the cases created in the last 30 days.

If you're using an environment variable as a Filter value, you must double-quote it. These variables are case-sensitive.

Environment Variable	Description
\$Vlocity.TODAY	Today's date
\$Vlocity.TOMORROW	Tomorrow's date
\$Vlocity.NOW	Current date and time. \$Vlocity.NOW applies to the user or org time zone, while the NOW() function applies UTC time.
\$Vlocity.N_DAYS_FROM_NOW:{N}	Specify the number of days from now

Environment Variable	Description
\$Vlocity.N_DAYS_AGO:{N}	Specify the number of days ago
\$Vlocity.NULL	Null
\$Vlocity.StandardPricebookId	ID of the standard price book
\$Vlocity.DocumentsFolderId	Documents folder ID
\$Vlocity.true or \$Vlocity.TRUE	Boolean TRUE
\$Vlocity.false or \$Vlocity.FALSE	Boolean FALSE
\$Vlocity.UserId	Current user ID
\$Vlocity.Percent	Percent character. Useful for escaping % characters in URLs and text field values in Omniscripts, Integration Procedures, and Data Mappers so they aren't mistaken for merge field syntax.
\$Vlocity.CpuTotal	Apex CPU value
\$Vlocity.DMLStatementsTotal	Number of Data Manipulation Language statements
\$Vlocity.DMLRowsTotal	Number of Data Manipulation Language rows
\$Vlocity.HeapSizeTotal	Heap size value
\$Vlocity.QueriesTotal	Number of queries run
\$Vlocity.QueryRowsTotal	Number of query rows fetched
\$Vlocity.SoslQueriesTotal	Number of SOSL queries run
User.userProfileName	Get the current user's profile name on Flexcard

Environment Variable	Description
userProfile	Get the current user's profile name on Omniscript

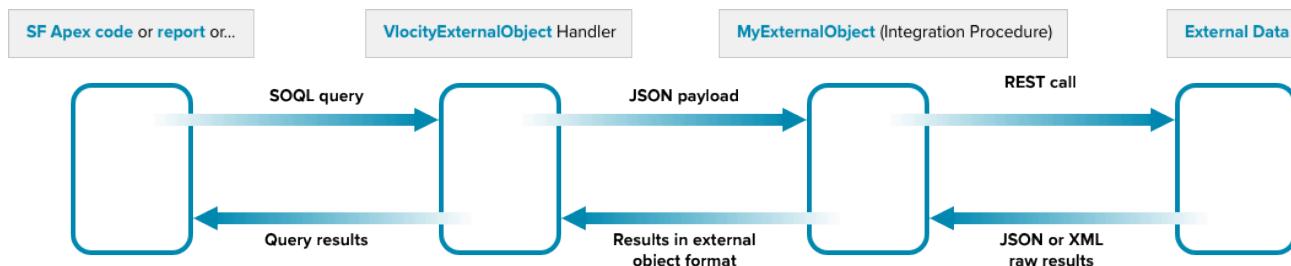
External Objects in Integration Procedures

Integration Procedures support Salesforce External Objects. The external data can come from any source that can be accessed using a REST endpoint. The returned data can be rendered as JSON or XML.

Here's how it works:

- In Salesforce, you perform an operation against the Vlocity external object, for example, in Apex code, in a report, or by issuing a SOQL query.
- The Vlocity external object handler calls the Integration Procedure that implements the external object, passing in a JSON payload that contains details about the request.
- Based on the contents of the JSON payload, the Integration Procedure accesses the external data using, for example, REST endpoints or Omnistudio Data Mapper calls, and returns JSON or XML results.
- The Integration Procedure returns its results to the Vlocity external object handler in an intermediate format.
- The Vlocity handler transforms the results into a JSON payload and returns them to the caller.

The following diagram illustrates the process.



The advantage of this approach to accessing external data is that it is completely declarative, no code required. Note that, while the external object functions as a single object, your Integration Procedure can recruit data from multiple sources.

Query-Handling Logic in the Integration Procedure

To indicate the type of operation being requested of the Integration Procedure, the Vlocity handler sets a "Context" entry in the input JSON. To define the actions for handling each type of query in your Integration Procedure, set the actions' Execution Conditional Formula field (for example: Context = "QueryAll"). The following list describes the values passed in the Context node for each type of request, with SOQL examples and the corresponding JSON payload sent to the Integration Procedure.

QueryAll: SELECT returning all rows (no WHERE clause)

Example query: `SELECT ExternalId FROM myExternalObject;`

Corresponding JSON input: `{"Context": "QueryAll"}`

Query: SELECT with WHERE clause

Example query: `SELECT ExternalId FROM vvelocity_dev__ContactsList__x WHERE ExternalId = '59f0f91a51280e0012c90584';`

Corresponding JSON input:

```
{"Input": [{"ExternalId": "59f0f91a51280e0012c90584"}], "Context": "Query"}
```

Upsert: Update a row (if ExternalId is specified) or create a row (if ExternalId is omitted).

Example query (UPDATE): `INSERT INTO myExternalObject (Name, Mobile...) VALUES ("Davidov Spruth", "415-607-9865")`

Corresponding JSON input: `{"Input": [{"Name": "Davidov Spruth", "Mobile": "415-607-9865", "CreateDate": "2017-10-25T20:50:34.188Z", "Work": "415-232-6365"}]}`

Delete: Delete specified rows.

Example query: `DELETE FROM myExternalObject WHERE ExternalId = '59f0f91a51280e0012c90584';`

Corresponding JSON input:

```
{"Input": [{"ExternalId": "59f0f91a51280e0012c90584"}], "Context": "Query"}
```

To access the external data to handle the query, define HTTP actions that call the REST endpoints that perform the required operations. If the results returned from the endpoint require further processing, create and call Data Mapper Transforms that perform the required transformations.

Query Result Format

To return results from the Integration Procedure to the query, structure the output JSON as follows:

SELECT queries: Return an array of nodes with names that correspond to the table columns defined for the external object. For example:

```
[ {
    "Work": "415-232-6365",
    "Mobile": "415-607-9865",
    "Email": "davs@gmail.com",
    "CreateDate": "2017-10-25T20:50:34.188Z",
```

```
        "Name": "Davidov Spruth",
        "ExternalId": "59f0f91a51280e0012c90584",
        "ExternalLookup": "59f0f91a51280e0012c90584",
        "AccountIndirectLookup": "123"
    },
    {
        "Email": "mw@gmail.com",
        "CreateDate": "2017-10-25T21:49:41.633Z",
        "Name": "Mike Wormwood",
        "ExternalId": "59f106f5e928870012f2b0a0",
        "ExternalLookup": "59f0f91a51280e0012c90584",
        "AccountIndirectLookup": "123"
    },
    {
        "Work": "33-555-1212",
        "Mobile": "33-879-0610",
        "CreateDate": "2017-10-25T20:59:18.657Z",
        "Name": "Tom Sor",
        "ExternalId": "59f0fb26e928870012f2b09e",
        "ExternalLookup": "59f0f91a51280e0012c90584",
        "AccountIndirectLookup": "123"
    }
]
```

UPDATE and DELETE queries: Return the external Id of the object affected in a node named "ExternalID"; for example:

```
{
    "ExternalId": "59efa135ba8e960012a3e8a5"
}
```

If the query does not succeed, return a node named "errorMessage" containing details about the problem that occurred.

Implement an External Object in an Integration Procedure

To access external data, you can define an Integration Procedure that implements a Salesforce External Object.

See Also

- [Define External Objects](#)
- [External Object Relationships](#)

Implement an External Object in an Integration Procedure

To access external data, you can define an Integration Procedure that implements a Salesforce External Object.

 **Note** This feature isn't available with the Omnistudio standard designer.

1. Create an Integration Procedure. For Type, specify `VlocityExternalObject`. For Subtype, specify the desired name for the external object.
2. In the Procedure Configuration section, define table and column settings. These are the same settings you can configure on the Salesforce External Object tab. For detailed information, refer to the Salesforce documentation: [Define External Objects](#) and [External Object Relationships](#).
3. Select **Activate**.
4. In Salesforce, go to the External Data Source tab and navigate to the `VlocityExternalObject` data source. Select **Validate and Sync**.

If the operation succeeds, your new external object is now listed on the page.

Work with Data and Lists

Use Integration Procedures to read, write, and transform complex, hierarchical data from Salesforce objects and other sources. Use actions to work with data, and blocks to process it.

Use a Data Mapper Extract Action with an Omnistudio Data Mapper Extract to access complex, hierarchical data that you can use with other actions. For example, if you create a Data Mapper Extract for the Contact object where `AccountId = id`, you can then map fields in the object to output fields (for example `contact:FirstName` to `contact:firstName`). To access the results, create a Data Mapper Extract Action and specify the name of the Data Mapper. You can then use the extracted list in a Loop Block or a List Block, addressing it by the name of the Data Mapper Extract Action and the output object (`DMExtractAddresses:contact`, for example).

Iterate over a List by Using a Loop Block

A Loop Block iterates over the items in a list, enabling the Actions within it to repeat for each item. In the Loop List parameter, you put a JSON node containing a list. The Loop Block processes one item in the list at a time. For example, suppose the list looks like this:

```
{  
    "Products": {  
        "Ids": [  
            {  
                "Id": 1  
            },  
            {  
                "Id": 2  
            }  
        ]  
    }  
}
```

```
{  
    "Id": 2  
}  
]  
}  
}
```

Each iteration of the Loop Block receives a single element, nested in the structure of the list:

```
{  
    "Products": {  
        "Ids": {  
            "Id": 1  
        }  
    }  
}
```

Actions within the Loop Block have access to one element at a time.

- Note** For a Loop Block that calls an Omnistudio Data Mapper Transform, a list with more than 1000 items degrades performance. You can use a [paging strategy](#) to break up long lists.
- Note** When the Omnistudio standard runtime is disabled and RollbackIPChanges is set to true, Loop Blocks in an Integration Procedure can result in loops executing within other loops. To resolve this, ensure that you have an Integration Procedure containing a loop that invokes a second Integration Procedure with an inner loop, RollbackIPChanges is set to false, or Omnistudio standard runtime is enabled.
- Note** When you enable the Include All Actions in Response setting for an Integration Procedure, a Loop Block in the Integration Procedure can't process an array of records. This occurs because the `stepResultFinal` array exceeds a size of 1 and results in an error: `Cannot Write Array to Root`

You can use a Loop Block to search for multiple records in a Salesforce object. Let's say you wanted to search for Contacts with the name "Miller" or Torres. You can put a Data Mapper Extract Action in a Loop Block, then pass in a list of names in the Input Parameters.

Consider the following JSON:

```
{  
    "names": [  
        {  
            "Name": "Miller"  
        }  
    ]  
}
```

```
    },
    {
        "Name": "Torres"
    }
]
```

To search for these names in a Loop Block:

1. Create a Data Mapper Extract that selects records with the Filter `Contact.LastName LIKE Name`.
2. Create a Data Mapper Extract Action in the Loop Block with the Data Source `names:Name` and the Filter Value `Name`. Set the Data Mapper Name to the name of the Data Mapper Extract.

As the Loop Block executes, the Data Mapper Extract Action uses each `Name` from the `names` list in the input JSON to search `Contact.LastName`.

Merge Lists by Using a List Action

The List Action merges multiple lists by matching values of specified list item JSON nodes. A basic merge matches node names exactly. An advanced merge matches nodes that have different names or reside at different levels in the incoming lists.

The inputs to a List Action must each be in list format even if the list contains only one item. Omnistudio Data Mapper Extracts that return a list with one item often convert it to a single object. To convert a single object back into a list, you can use the first example in [Omnistudio Data Mapper Transform Examples](#).

For more list processing options, see [List Input for Omnistudio Data Mappers](#) and the AVG, FILTER, IF, LIST, LISTMERGE, LISTMERGEPRIMARY, LISTSIZE, MAX, MIN, SORTBY, and SUM functions in the [Function Reference](#).

To match nodes that have different names or reside at different levels in the incoming lists, click Advanced Merge and specify settings for the nodes to be matched, as follows:

- List Key: Enter the name of the JSON list node where the node to be matched resides.
- Matching Path: Enter the path within the list to the node to be matched.
- Matching Group: Specify the same number for all nodes you want to match.

For example, this figure shows how to match first and last names from two incoming lists in which the nodes have different names.

Advanced Merge 

Advanced Merge Map

MERGE MAP 			
List Key  	Matching Path  	Matching Group 	
DRExtractAddresses:contact	firstName	1	
DRExtractBirthdates:contact	FN	1	
DRExtractAddresses:contact	lastName	2	
DRExtractBirthdates:contact	LN	2	

Concatenate List Items

This Integration Procedure example concatenates values from an array into a single comma-separated string by using a Set Values Action inside a Loop Block, then trimming and returning the output.

Create Contacts by Using a Loop Block

Use a Loop Block to iterate over a list provided in the input JSON, adding a Contact object for each element in the list.

Set Values in a List by Using Conditions

In this example, an Integration Procedure adds a dynamic value to each item in a list using conditions. Although dynamic in the list, the values are hard-coded in the Integration Procedure. Use this approach if you have a specific set of possible values.

Add Values to List Items by Using VALUELOOKUP

In this example, an Integration Procedure adds a dynamic value to each item in a list using the VALUELOOKUP function. The values aren't hard-coded in the Integration Procedure. Use this approach if you don't know the set of possible values in the input.

Match Records by Using Advanced Merge

Advanced Merge matches nodes with different names or at different levels in incoming lists. For example, instead of using a primary key that exists in both lists, you can match records based on first and last name.

List Action Examples

Use these Integration Procedure examples to explore the capabilities of List Actions. Each example builds on the previous example.

See Also

[Create an Omnistudio Data Mapper Extract](#)

[List Input for Omnistudio Data Mappers](#)

Concatenate List Items

This Integration Procedure example concatenates values from an array into a single comma-separated string by using a Set Values Action inside a Loop Block, then trimming and returning the output.

For this example, use this simple JSON input:

```
{  
  "Items": [
```

```
{  
    "Item": "First"  
,  
{  
    "Item": "Second"  
,  
{  
    "Item": "Third"  
}  
]  
}
```

You create a Loop Block, with `Items` as the Loop List. Inside the Loop Block, you use a Set Values Action to add the value of each `Item` to a string called `ConcatString`. Finally, you trim the extra characters from `ConcatString` by using a Set Values action, then use a Response Action to pass `ConcatString` to the output.

Use the following steps to configure the Integration Procedure.

1. Create a Loop Block with `Items` in the Loop List field.
2. Inside the Loop Block, create a Set Values Action to concatenate the current item to `ConcatString`.
3. In the Set Values Action's Element Value Map, add one element, `Concat`, that gets the current value of `ConcatString` and adds Item.
 - Element Name: `Concat`
 - Value: `=%ConcatString% + ", " + %Items:Item%`
4. Use the Response JSON to put `Concat` into `ConcatString` for the next round.
 - Response JSON Path: `Concat`
 - Response JSON Node: `ConcatString`
5. After the Loop Block, the strings are concatenated, but there's an extra comma and space at the beginning of `ConcatString` from the first execution of the loop. You can trim the output by adding a Set Values component after the Loop Block.
 - Element Name: `TrimOutput`
 - Element Value Map: `ConcatString`
 - Element Name: `TrimmedString`
 - Value: `=SUBSTRING(ConcatString, 2)`
6. Finally, send the resulting `TrimmedString` from `TrimOutput` to the output by using a Response Action.
 - Send JSON Path: `TrimOutput:TrimmedString`
 - Send JSON Node: `Output`

The output looks like this.

```
{
```

```
"Output": "First, Second, Third"  
}
```

Create Contacts by Using a Loop Block

Use a Loop Block to iterate over a list provided in the input JSON, adding a Contact object for each element in the list.

For this example, assume the input JSON contains an Account Id and an array of first and last names, like the snippet below.

```
{  
    "AccountId": "0016100001BKL4uAAH",  
    "Contacts": [  
        {  
            "Name": {  
                "First": "John",  
                "Last": "Doe"  
            }  
        },  
        {  
            "Name": {  
                "First": "June",  
                "Last": "Doe"  
            }  
        }  
    ]  
}
```

To create a contact, you can use a Data Mapper Load. When the script invokes the Data Mapper Load, the Data JSON is sent as input, so the Data Mapper Load can refer directly to the nodes in the input JSON.

After you create the Data Mapper Load, create a Data Mapper Post Action with properties similar to the properties in the table.

Data Mapper Post Action Properties

Property	Description
Data Mapper Interface	The name of the Data Mapper Load
Additional Input	<ul style="list-style-type: none">• Key: <code>AccountId</code>

Property	Description
	<ul style="list-style-type: none">• Value: <code>%AccountId%</code>• Key: <code>Name</code>• Value: <code>%Contacts:Name%</code>

This passes the first and last name of each contact, along with the AccountId, to the Data Mapper Load on each execution. You can also include more fields, such as phone numbers and email addresses.

Finally, you can create a Loop block to execute the Data Mapper Post Action for every contact. Use the name of the list of contacts for the Loop List parameter. For example, from the input JSON above, the Loop List would be `Contacts`. You can use the Additional Output parameter to pass along the execution output from each iteration of the Data Mapper Post Action. For example, if you named the Data Mapper Post Action `CreateContact` then for the Additional Output key and value you can use `CreateContact` and `%CreateContact%`, respectively.

Data Mapper Post Action output, which can be viewed in the Debug log on the Preview tab, allows other steps in the Integration Procedure to access and use the IDs and other details of the processed sObjects.

 **Note** Using an Omnistudio Data Mapper Post action in a Loop Block can sometimes reach the limit of 150 DML statements. See [Execution Governors and Limits](#).

Set Values in a List by Using Conditions

In this example, an Integration Procedure adds a dynamic value to each item in a list using conditions. Although dynamic in the list, the values are hard-coded in the Integration Procedure. Use this approach if you have a specific set of possible values.

Here's some sample input, with `uservalue` values initially set to `null`. The `uservalue` nodes don't have to be included in the input, but it's easier to compare input to output if they are.

```
{  
  "Labels": {  
    "OfferType": "Remediation",  
    "BillingCycle": "Immediate"  
  },  
  "productAttributes": {  
    "records": [  
      {  
        "record": {  
          "uservalue": null,  
          "label": "BillingCycle"  
        }  
      }  
    ]  
  }  
}
```

```
        }
    },
    {
        "record": {
            "uservalue": null,
            "label": "OfferType"
        }
    }
]
}
```

The Integration procedure contains a Loop Block that iterates over the elements in `productAttributes:records`. In the Loop Block are two Set Values components. Each one uses a Response JSON Node to set `uservalue` to a value it retrieves from the list of `Labels`, and each one uses an Execution Conditional Formula to execute only on a particular `label` in the record.

The output is similar to this example, with `uservalue` values set according to the `label` value in each list item.

```
{
  "Labels": {
    "OfferType": "Remediation",
    "BillingCycle": "Immediate"
  },
  "productAttributes": {
    "records": [
      {
        "record": {
          "uservalue": "Immediate",
          "label": "BillingCycle"
        }
      },
      {
        "record": {
          "uservalue": "Remediation",
          "label": "OfferType"
        }
      }
    ]
  }
}
```

Here's a list of the components and their configurations.

- A Loop Block named `LoopBlock1`, configured with these settings:
 - Loop List: `productAttributes:records`
 - Additional Loop Output:
 - Key: `productAttributes`
 - Value: `%productAttributes%`
- A Set Values component within `LoopBlock1`, named `IfOfferType`, configured with these settings:
 - Element Name: `IfOfferType`
 - Element Value Map:
 - Element Name: `LabelValue`
 - Value: `=%Labels:OfferType%`
 - Response JSON Path: `LabelValue`
 - Response JSON Node: `productAttributes:records:record:uservalue`
 - Execution Conditional Formula: `productAttributes:records:record:label == "OfferType"`
- Another Set Values component within `LoopBlock1`, named `IfBillingCycle`, configured with these settings:
 - Element Name: `IfBillingCycle`
 - Element Value Map:
 - Element Name: `LabelValue2`
 - Value: `=%Labels:BillingCycle%`
 - Response JSON Path: `LabelValue2`
 - Response JSON Node: `productAttributes:records:record:uservalue`
 - Execution Conditional Formula: `productAttributes:records:record:label == "BillingCycle"`
- A Set Values component after the Loop Block, named `AssembleOutput`, configured with these settings:
 - Element Name: `AssembleOutput`
 - Element Value Map:
 - Element Name: `productAttributes:records`
 - Value: `%LoopBlock1:productAttributes:records%`
 - Element Value Map:
 - Element Name: `Labels`
 - Value: `%Labels%`
- A Response Action, named `ResponseAction1`, configured with this setting:
 - Send JSON Path: `AssembleOutput`

Add Values to List Items by Using VALUELOOKUP

In this example, an Integration Procedure adds a dynamic value to each item in a list using the `VALUELOOKUP` function. The values aren't hard-coded in the Integration Procedure. Use this approach if you don't know the set of possible values in the input.

Here's some sample input, with `uservalue` values initially set to `null`. The `uservalue` nodes don't

have to be included in the input, but it's easier to compare input to output if they are.

```
{  
  "Labels": {  
    "OfferType": "Remediation",  
    "BillingCycle": "Immediate"  
  },  
  "productAttributes": {  
    "records": [  
      {  
        "record": {  
          "uservalue": null,  
          "label": "BillingCycle"  
        }  
      },  
      {  
        "record": {  
          "uservalue": null,  
          "label": "OfferType"  
        }  
      }  
    ]  
  }  
}
```

The `LabelValue` formula in the Loop Block's Set Values component looks under `Labels` for the node with the name specified in the `label` value of the current list item, then returns the value of that node.

The output looks like this, with `uservalue` values set according to the `label` value in each list item.

```
{  
  "Labels": {  
    "OfferType": "Remediation",  
    "BillingCycle": "Immediate"  
  },  
  "productAttributes": {  
    "records": [  
      {  
        "record": {  
          "uservalue": "Immediate",  
          "label": "BillingCycle"  
        }  
      },  
      {  
        "record": {  
          "uservalue": "Remediation",  
          "label": "OfferType"  
        }  
      }  
    ]  
  }  
}
```

```

    "record": {
        "uservalue": "Remediation",
        "label": "OfferType"
    }
}
]
}
}
}

```

Here's a list of the components and their configurations.

- A Loop Block named LoopBlock1, configured with these settings:

Property
Loop List
Additional Loop Output

- A Set Values component within the Loop Block, configured with these settings:

Property	Description
Element Name	GetValueForListItem
Element Value Map	<ul style="list-style-type: none"> - Element Name: LabelValue - Value: <pre>=VALUELOOKUP(Labels, productAttributes:records:record:label)</pre>
Response JSON Path	LabelValue
Response JSON Node	productAttributes:records:record:uservalue

- A Set Values component after the Loop Block, configured with these settings:

Property	Description
Element Name	AssembleOutput
Element Value Map	<ul style="list-style-type: none"> - Element Name: productAttributes - Value: %LoopBlock1:productAttributes%
Element Value Map	<ul style="list-style-type: none"> - Element Name: Labels

Property	Description
	- Value: %Labels%

- A Response Action below the last Set Values component, with its **Send JSON Path** property set to `AssembleOutput`.

Match Records by Using Advanced Merge

Advanced Merge matches nodes with different names or at different levels in incoming lists. For example, instead of using a primary key that exists in both lists, you can match records based on first and last name.

To use Advanced Merge, check the **Advanced Merge** box and creat an Advanced Merge Map table. The example table provides the paths and names of fields to match, using groups to indicate which fields to compare.

List Key	Matching Path	Matching Group
CopyBirthdates:contact	firstName	1
DRExtractBirthdates:contact	firstName	1
CopyBirthdates:contact	lastName	2
DRExtractBirthdates:contact	lastName	2

The two `firstName` fields are compared because they are both in group 1, and the two `lastName` fields are compared because they are both in group 2. For a record to be merged, the `firstName` and `lastName` fields must match.

List Action Examples

Use these Integration Procedure examples to explore the capabilities of List Actions. Each example builds on the previous example.

1. [Create a Basic List Merge Example](#).
2. [Create an Advanced List Merge Example](#).
3. [Create a List Merge Example with Dynamic Output Fields](#).

[Create a Basic List Merge Example](#)

In this example, the first Integration Procedure retrieves all Contacts with the same AccountId in two lists, one containing addresses, the other birthdates. It then merges the two lists into one.

Create an Advanced List Merge Example

The next Integration Procedure is a modified version of the previous example. Instead of matching list items using Id fields, it uses the Advanced Merge feature to match first and last names.

Create a List Merge Example with Dynamic Output Fields

The final Integration Procedure is a modified version of the previous example. It uses **Dynamic Output Fields** to eliminate the duplicate `FN` and `LN` JSON nodes from the output.

Create a Basic List Merge Example

In this example, the first Integration Procedure retrieves all Contacts with the same AccountId in two lists, one containing addresses, the other birthdates. It then merges the two lists into one.

If you're using the designer on a managed package, you can download a data pack of this example from [here](#).

The Integration Procedure has these components:

1. An Omnistudio Data Mapper Extract Action component named DRExtractAddresses
 2. A Data Mapper Extract Action component named DRExtractBirthdates
 3. A List Action named ListAction1
 4. A Response Action named ResponseAction1
1. Create the Data Mapper Extract that the DRExtractAddresses component calls. Name it GetAddresses and configure it with these settings:

Extract	A Contact extract step set to contact <code>AccountId</code> = <code>id</code>
Output	<p>These mappings:</p> <ul style="list-style-type: none">• <code>contact:Id</code> to <code>contact:id</code>• <code>contact:FirstName</code> to <code>contact:firstName</code>• <code>contact:LastName</code> to <code>contact:lastName</code>• <code>contact:MailingStreet</code> to <code>contact:street</code>• <code>contact:MailingCity</code> to <code>contact:city</code>• <code>contact:MailingState</code> to <code>contact:state</code>

If you aren't sure how to create a Data Mapper Extract, see the examples in [Omnistudio Data Mapper Extract Examples](#).

2. Create the Data Mapper Extract that the DRExtractBirthdates component calls:
 - a. Clone the GetAddresses Data Mapper and name the clone GetBirthdates.
 - b. Go to the Create Mapping page, and delete the city, state, and street mappings. If you're using the designer on a managed package, go to the Output tab, and delete the city, state, and street

mappings.

- c. Add a mapping from `contact:Birthdate` to `contact:birthday`.
3. From the App Launcher, find and select **Integration Procedures**.
4. Click **New**.
5. Enter a unique name, type, and subtype for your Integration Procedure.
6. Enter a description for your Integration Procedure.
7. Save your changes. The Integration Procedure canvas opens.
8. Add the first Data Mapper Extract Action component:
 - To open the Elements panel, click **+**, select **Data Mapper Extract**, click **...**, and select **Details**.
 - If you're using the designer on a managed package, open the Available Components panel, and drag **Data Mapper Extract Action** to the Structure panel.
9. Enter `DRExtractAddresses` as the element name, and `GetAddresses` as the Data Mapper name.
10. Add the second Data Mapper Extract Action component below the first Data Mapper Extract Action component.
11. Enter `DRExtractBirthdates` as the element name, and `GetBirthdates` as the Data Mapper name.
12. Add a List Action component into the Structure panel below the second Data Mapper Extract Action component and configure it with these settings:

Merge Lists Order	<ul style="list-style-type: none">• <code>DRExtractAddresses:contact</code>• <code>DRExtractBirthdates:contact</code>
Merge Fields	<code>id</code>
Sort By	<ul style="list-style-type: none">• <code>lastName</code>• <code>firstName</code>

13. Add a Response Action component into the Structure panel below other action components.
14. Enter `ListAction1` in the Send JSON Path field, and `contactMerge` in the Send JSON Node field.
15. On the Preview tab, in the Input Parameters pane, click **Add Key-Value Pair**.
16. Enter `id` as the key and an Account Id from your org as the Value.
17. Click **Execute**.

The output is similar to this example:

```
{  
  "contactMerge": [  
    {  
      "birthdate": "1975-10-03",  
      "street": "300 Broadway",  
      "state": "FL",  
      "city": "Orlando",  
      "lastName": "Jones",  
      "id": "0036100001E5xrWAAR",  
      "firstName": "Cathy"  
    },  
    {  
      "birthdate": "1976-10-04",  
      "street": "123 Main Street",  
      "state": "CA",  
      "city": "San Francisco",  
      "lastName": "Smith",  
      "id": "0036100002E5xrWAAR",  
      "firstName": "John"  
    }  
  ]  
}
```

```
        "street": "400 Washington Blvd",
        "state": "AZ",
        "city": "Phoenix",
        "lastName": "Jones",
        "id": "0036100001E5xrvAAB",
        "firstName": "Doug"
    },
    {
        "birthdate": "1973-10-01",
        "street": "100 Main Street",
        "state": "CA",
        "city": "San Francisco",
        "lastName": "Smith",
        "id": "0036100001E5xqnAAB",
        "firstName": "Albert"
    },
    {
        "birthdate": "1974-10-02",
        "street": "200 Second Street",
        "state": "OR",
        "city": "Portland",
        "lastName": "Smith",
        "id": "0036100001E5xr2AAB",
        "firstName": "Ben"
    }
]
}
```

What's next: [Create an Advanced List Merge Example](#).

Create an Advanced List Merge Example

The next Integration Procedure is a modified version of the previous example. Instead of matching list items using Id fields, it uses the Advanced Merge feature to match first and last names.

If you're using the designer on a managed package, you can download a data pack of this example from [here](#).

1. [Create a Basic List Merge Example](#).
2. Clone the GetBirthdates Omnistudio Data Mapper and name it GetBirthdates2. On the Output tab, change the **Output JSON Path** values from `contact:firstName` to `contact:FN` and from `contact:lastName` to `contact:LN`.
3. In the Integration Procedure, in the DRExtractBirthdates component, change the **Data Mapper Interface** to `GetBirthdates2`.

4. In the Integration Procedure, in the List Action, check **Advanced Merge**, and add these rows to the Advanced Merge Map table:

List Key	Matching Path	Matching Group
DRExtractAddresses:cont ct	firstName	1
DRExtractBirthdates:cont act	FN	1
DRExtractAddresses:cont ct	lastName	2
DRExtractBirthdates:cont act	LN	2

5. On the Preview tab, click **Execute**. The output is similar to this example:

```
{
  "contactMerge": [
    {
      "LN": "Jones",
      "FN": "Cathy",
      "birthdate": "1975-10-03",
      "street": "300 Broadway",
      "state": "FL",
      "city": "Orlando",
      "lastName": "Jones",
      "id": "0036100001E5xrWAAR",
      "firstName": "Cathy"
    },
    {
      "LN": "Jones",
      "FN": "Doug",
      "birthdate": "1976-10-04",
      "street": "400 Washington Blvd",
      "state": "AZ",
      "city": "Phoenix",
      "lastName": "Jones",
      "id": "0036100001E5xrvAAB",
      "firstName": "Doug"
    }
  ]
}
```

```
{  
    "LN": "Smith",  
    "FN": "Albert",  
    "birthdate": "1973-10-01",  
    "street": "100 Main Street",  
    "state": "CA",  
    "city": "San Francisco",  
    "lastName": "Smith",  
    "id": "0036100001E5xqnAAB",  
    "firstName": "Albert"  
,  
    {  
        "LN": "Smith",  
        "FN": "Ben",  
        "birthdate": "1974-10-02",  
        "street": "200 Second Street",  
        "state": "OR",  
        "city": "Portland",  
        "lastName": "Smith",  
        "id": "0036100001E5xr2AAB",  
        "firstName": "Ben"  
    }  
}  
}
```

What's next: [Create a List Merge Example with Dynamic Output Fields](#).

Create a List Merge Example with Dynamic Output Fields

The final Integration Procedure is a modified version of the previous example. It uses **Dynamic Output Fields** to eliminate the duplicate `FN` and `LN` JSON nodes from the output.

If you're using the designer on a managed package, you can download a data pack of this example from [here](#).

1. [Create an Advanced List Merge Example](#).
2. In the List Action, give the **Dynamic Output Fields** setting a value of `include`.
3. Go to the Preview tab.
4. Click **Edit as JSON**. If you're using the designer on a managed package, click **Preview**, and then click **Edit as JSON**.
5. Click **Add New Key-Value Pair**. Enter `include` as the key, and `birthdate,street,state,city,lastName,firstName,id` as the value. Only commas, and not spaces, separate the node names.
6. Click **Execute**. The output is similar to this example:

```
{  
  "contactMerge": [  
    {  
      "birthdate": "1975-10-03",  
      "street": "300 Broadway",  
      "state": "FL",  
      "city": "Orlando",  
      "lastName": "Jones",  
      "id": "0036100001E5xrWAAR",  
      "firstName": "Cathy"  
    },  
    {  
      "birthdate": "1976-10-04",  
      "street": "400 Washington Blvd",  
      "state": "AZ",  
      "city": "Phoenix",  
      "lastName": "Jones",  
      "id": "0036100001E5xrvAAB",  
      "firstName": "Doug"  
    },  
    {  
      "birthdate": "1973-10-01",  
      "street": "100 Main Street",  
      "state": "CA",  
      "city": "San Francisco",  
      "lastName": "Smith",  
      "id": "0036100001E5xqnAAB",  
      "firstName": "Albert"  
    },  
    {  
      "birthdate": "1974-10-02",  
      "street": "200 Second Street",  
      "state": "OR",  
      "city": "Portland",  
      "lastName": "Smith",  
      "id": "0036100001E5xr2AAB",  
      "firstName": "Ben"  
    }  
  ]  
}
```

Define Execution Logic

Define conditional flow in Integration Procedures by using the Execution Conditional Formula field on an action or block, or by using a Conditional Block to compose a series of mutually exclusive conditions.

- Every action and block can have an Execution Conditional Formula
- Use a Conditional Block for more complex conditions

When you add a formula to the Execution Conditional Formula field on an action or block, the result of the formula determines whether the action or block is executed.

- If the formula returns TRUE, the action or block is executed.
- If the formula returns FALSE, the action or block isn't executed.
- If no formula is specified, the action or block is executed.

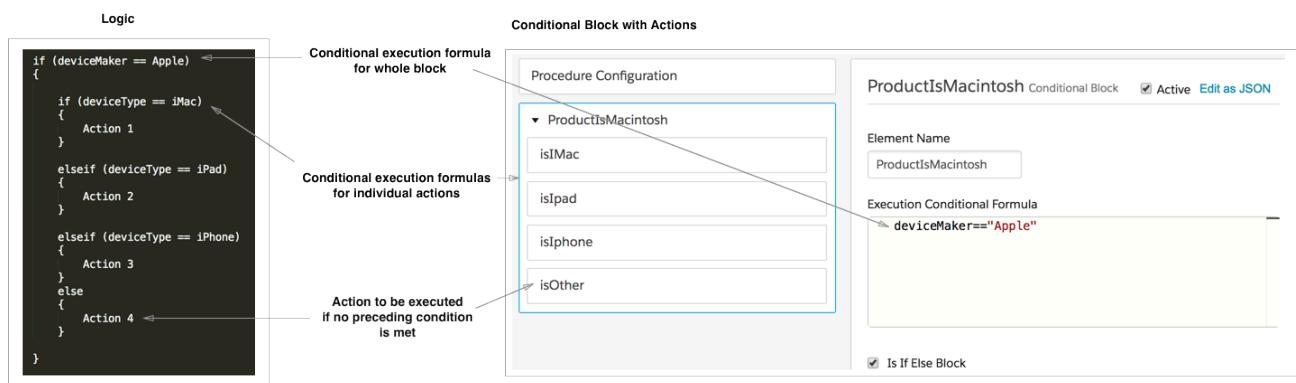
For example, to execute an action or block for records where the Contact First Name is blank, you might use the formula `%ContactInfo:FirstName% == ""`.

When you define an Execution Conditional Formula on a block, it affects all the actions and blocks inside the block. Elements in the block can have their own formulas. Any such action or block executes only if its own formula and the formula for the enclosing block are both true.

If you need to group actions and blocks behind a formula without performing a loop or merging a list, you can use the simplest block type: the Conditional Block. The Conditional Block's only job is to control the Integration Procedure flow.

If you need to define a group of mutually exclusive conditions, use a Conditional Block and check the **Is If Else** field. When the Conditional Block executes, the first action or block whose formula evaluates to True executes. No other actions or blocks in the Conditional Block execute. The execution of the Integration Procedure jumps to the first element after the Conditional Block. To configure a step that is executed if none of the preceding steps execute, leave the Execution Conditional Formula of the last element in the Conditional Block blank. To execute multiple actions for a condition, you can nest Conditional Blocks.

This figure illustrates the use of this logic to perform a different action, depending on the type of device. The Conditional Block itself executes only when the device being processed comes from Apple, and the block contains actions that execute only for a specific type of device.



See Also

[Omnistudio Formulas and Functions](#)

Send Email from an Integration Procedure

To send email from an Integration Procedure, use an Email Action. You can either specify all the email field values or use a Salesforce email template.

Create an Email Action Example with Specified Fields

In this example, an Integration Procedure accepts `email`, `ccemail`, `subject`, and `message` input parameters and uses them to compose and send an email.

Create an Email Action with Template

An Integration Procedure uses an Email Template and includes the email that's sent in the recipient's Activities list. No Email Templates exist by default, so this example includes steps for creating one.

Create an Email Action Example with Specified Fields

In this example, an Integration Procedure accepts `email`, `ccemail`, `subject`, and `message` input parameters and uses them to compose and send an email.

If you're using the designer on a managed package, you can download a [data pack example](#).

Note

- Importing Omnistudio for Managed Packages data packs into Omnistudio orgs is supported, and conversion is automatic.
- Importing Omnistudio DataPacks into Omnistudio for Managed Packages orgs isn't supported.

To build this Integration Procedure:

- From the App Launcher, find and select **Omnistudio Integration Procedures**.
- Click **New**.
- Enter a unique name, type, and subtype for your Integration Procedure.
- Enter a description for your Integration Procedure.

5. Save your changes. The Integration Procedure canvas opens.
6. Add an Email Action component.
 - To open the Elements panel, click +, select **Email**, click •••, and select **Details**.
 - If you're using the designer on a managed package, open the Available Components panel, and drag Email Action to the Structure panel.
7. In the Email Action Properties tab:
 - a. Deselect **Use Template**.
 - b. Add these properties:

Property	Value
TO EMAIL ADDRESS LIST	Click Add Recipient and enter <code>%email%</code> .
CC EMAIL ADDRESS LIST	Click Add Recipient and enter <code>%ccemail1%</code> . Click Add Recipient again and enter <code>%ccemail2%</code> .
Email Subject	Enter <code>%subject%</code> .
Email Body	Enter <code>%message%</code> .

8. To enter values for the input parameters, open the Email Action Preview tab.
Under Input Parameters in the Preview tab, toggle between **Edit as Params** and **Edit as JSON** to specify the input parameters either in fields or as JSON.
 - To use **Edit as Params**, click **Add New Key/Value Pair** five times and add these input parameters in the Key and Value fields:

Key	Value
email	<code>your_email_address</code>
ccemail1	<code>first_ccemail_address</code>
ccemail2	<code>second_ccemail_address</code>
subject	Test Email
message	This email is a test.

- To use **Edit as JSON**, add these input parameters as key-value pairs in a JSON object:

```
{
  "email": "your_email_address",
  "ccemail1": "first_ccemail_address",
  "ccemail2": "second_ccemail_address",
  "subject": "Test Email",
```

```
    "message": "This email is a test."  
}
```

9. Click **Execute**. In a few minutes, the test email is sent from your org to the specified recipients.

Create an Email Action with Template

An Integration Procedure uses an Email Template and includes the email that's sent in the recipient's Activities list. No Email Templates exist by default, so this example includes steps for creating one.

If you're using the designer on a managed package, you can download a [data pack example](#).

To build this example:

1. [Create the Email Template](#).
2. [Select an Email Recipient](#).
3. [Create the Integration Procedure with the Email Action](#).
4. [Test the Integration Procedure with the Email Action](#).

[Create the Email Template](#)

Create the email template that the Email Action example uses.

[Select an Email Recipient](#)

Select the ID of the Contact to whom the email in the Email Action example is to be sent.

[Create the Integration Procedure with the Email Action](#)

After you create the email template and select the recipient, you can create an Integration Procedure that sends an email to a Contact.

[Test the Integration Procedure with the Email Action](#)

After you've created the Integration Procedure with the Email Action, your final task is to test it.

Create the Email Template

Create the email template that the Email Action example uses.

1. Go to the Email Templates tab, and click **New Email Template**.
2. In the **Email Template Name** field, type *Simple Email Template*.
3. Enter details in the **Subject** and **HTML Value** fields.

For example, enter the **Subject** as *Hello* and the **HTML Value** as *Just saying hello*.

4. Save your changes. The Email Template's page opens.
5. While on the Email Template's page, copy the Id from the URL.

An example of an ID is `00X4N000000aCE5UAM`.

Select an Email Recipient

Select the ID of the Contact to whom the email in the Email Action example is to be sent.

1. Go to the Contacts tab.
2. Find a Contact that has an email address, or add an email address for a Contact.
3. While on the Contact's page, copy the ID from the URL.

An example of an ID is **0036100000423z8AAA**.

Create the Integration Procedure with the Email Action

After you create the email template and select the recipient, you can create an Integration Procedure that sends an email to a Contact.

1. From the App Launcher, find and select **Omnistudio Integration Procedures**.
2. Click **New**.
3. Enter a unique name, type, and subtype for your Integration Procedure.
4. Enter a description for your Integration Procedure.
5. Save your changes. The Integration Procedure canvas opens.
6. Add an Email Action component:
 - To open the Elements panel, click **+**, select **Email**, click **...**, and select **Details**.
 - If you're using the designer on a managed package, open the Available Components panel, and drag **Email Action** to the Structure panel.
7. Select **Use an email template**. If you're using the designer on a managed package, select **Use Template**.
8. Copy and paste the Email Template ID into the Email Template field. A dropdown list appears with an item.
An example item is **Simple_Email_Template_1580943234381**.
9. Select the list item. The list item replaces the ID in the field.
10. In the Email Target Object Id field, type **%contact%**.
11. Select **Save As Activity**.

Test the Integration Procedure with the Email Action

After you've created the Integration Procedure with the Email Action, your final task is to test it.

1. Go to the Preview tab.
2. Click **Edit as JSON**. If you're using the designer on a managed package, click **Preview** and then click **Edit as JSON**.
3. In the Input Parameters panel, add **contact** as the key and the Contact Id that you copied previously as the value.
4. Click **Execute**. This step sends the email.
5. Go to the Contacts tab. Select the Contact to whom you sent the email to open their page.

If the Contact record page includes an Activities component, click **Refresh** to see the email in the list

of activities.

6. If the Contact record page doesn't include an Activities component, add a component:
 - a. If you're using Salesforce Classic, click **Switch to Lightning Experience**.
 - b. Click . If you're using the designer on a managed package, click  and then click **Edit Page**.
 - c. Add an Activities component:
 - To open the Elements panel, click +, select **Activities**, click •••, and select **Details**.
 - If you're using the designer on a managed package, open the Available Components panel and drag **Activities** to the Structure panel.
 - d. Click **Save**, then click **Back** to return to the Contact's page. The Activities component appears on the Contact's page and lists the email you sent.

Make an HTTP Call from an Integration Procedure

This Integration Procedure example retrieves the astronomy picture of the day from NASA using NASA's publicly available REST API.

To build this example:

1. [Add NASA as a Remote Site](#).
2. [Get an API Key from NASA](#).
3. [Create the Integration Procedure with the HTTP Action](#).

Add NASA as a Remote Site

To invoke NASA REST APIs from your org, you must add NASA as a Remote Site.

Get an API Key from NASA

You must specify an API key every time you invoke a NASA REST API. You can get this key from NASA.

Create the Integration Procedure with the HTTP Action

After you configure NASA as a Remote Site and get a NASA API key, you can create and run the example Integration Procedure, which retrieves the astronomy picture of the day.

Add NASA as a Remote Site

To invoke NASA REST APIs from your org, you must add NASA as a Remote Site.

1. From Setup, in the Quick Find box, type *remote*, then click **Remote Site Settings**.
2. Click **New Remote Site**.
3. For the **Remote Site Name**, enter *NASA*.
4. For the **Remote Site URL**, enter *https://api.nasa.gov*.
5. Save your changes.

What's next: [Get an API Key from NASA](#).

Get an API Key from NASA

You must specify an API key every time you invoke a NASA REST API. You can get this key from NASA.

1. Go to <https://api.nasa.gov>.
2. Enter your **First Name**, **Last Name**, and **Email**, then click **Signup**.
3. Copy the response and save it to a text file. The response is similar to this example:

```
Your API key for me@example.com is:  
YbbMNWeWX2BqxoPKXEiWWcKMgN1UHhHXggWG5Xbt
```

You can start using this key to make web service requests. Simply pass your key in the URL when making a web request. Here's an example:

```
https://api.nasa.gov/planetary/apod?api_key=YbbMNWeWX2BqxoPKXEiWWcKMgN1UHhHX  
ggWG5Xbt
```

For additional support, please contact us. When contacting us, please tell us what API you're accessing and provide the following account details so we can quickly find you:

```
Account Email: me@example.com  
Account ID: b4345628-22cd-4a1d-b610-3d9ec5ba95fd
```

You need your API key to run the Integration Procedure.

4. (Optional) To see the types of data you can retrieve from NASA, click **Browse APIs**.

What's next: [Create the Integration Procedure with the HTTP Action](#).

Create the Integration Procedure with the HTTP Action

After you configure NASA as a Remote Site and get a NASA API key, you can create and run the example Integration Procedure, which retrieves the astronomy picture of the day.

1. Add an HTTP Action component.
2. Configure the HTTP Action component with these settings:

Property	Value
HTTP Path	https://api.nasa.gov/planetary/apod
HTTP Method	GET

Property	Value
REST OPTIONS: Add Parameter	<ul style="list-style-type: none"> Key: <code>api_key</code> Value: <code>%ApiKey%</code>

3. Add a Response Action component into the Structure panel below the HTTP Action, and configure it with these settings:

Property	Value
Send JSON Path	<code>HTTPAction1</code>
Send JSON Node	<code>PictureInfo</code>

4. Edit the input JSON, adding the key `ApiKey` and your API key as the value.

5. Click **Execute**. The output is similar to this example:

```
{
  "PictureInfo": {
    "url": "https://apod.nasa.gov/apod/image/2006/Eclipse-under-bamboos1024c.jpg",
    "title": "Eclipse under the Bamboo",
    "service_version": "v1",
    "media_type": "image",
    "hdurl": "https://apod.nasa.gov/apod/image/2006/Eclipse-under-bamboos.jpg",
    "explanation": "Want to watch a solar eclipse safely? Try looking down instead of up, though you might discover you have a plethora of images to choose from. For example, during the June 21st solar eclipse this confusing display appeared under a shady bamboo grove in Pune, India. Small gaps between close knit leaves on the tall plants effectively created a network of randomly placed pinholes. Each one projected a separate image of the eclipsed Sun. The snapshot was taken close to the time of maximum eclipse in Pune when the Moon covered about 60 percent of the Sun's diameter. But an annular eclipse, the Moon in silhouette completely surrounded by a bright solar disk at maximum, could be seen along a narrow path where the Moon's dark shadow crossed central Africa, south Asia, and China",
    "date": "2020-06-26",
    "copyright": "Somak Raychaudhury"
  }
}
```

6. (Optional) Paste the `url` or `hdurl` value into your browser and view the picture.

Call an Apex Class or an Invocable Action from an Integration Procedure

To call an Apex class and method, or an invocable action, use a Remote Action and pass in invocation options and data.

If the data that a remote action returns isn't in `Map<String, Object>` format, sometimes an Integration Procedure or Omnistudio Data Mapper can't process it. Because the returned data isn't an sObject, you can't use it to update Product2 objects without reserializing. You can convert the data in one of these ways:

- In the remote action, under **Remote Options**, add a **Key** named `reserialize` with a Value of `true`.
- In a subsequent Set Values step, use the RESERIALIZE function on the remote action's output. For information about related functions such as DESERIALIZE and SERIALIZE, see the [Supported Data Mapper and Integration Procedure Functions](#). For information about reserializing map objects as inputs, see [Create a Remote Action Apex Class Example](#).

An error message that begins with `Invalid conversion from runtime type String` often indicates a need to reserialize data.

Make sure that you include the ID in the SOQL query. That field is required because the Apex class retains the queries and serializes data according to what's executed.

If you receive an error that says the Apex class can't be loaded, add the Apex class namespace in the Remote Class field in the remote action like this: `{yournamespace}.ApexClassName`. For example, `omnistudio.PrefillOrders` where `omnistudio` is the namespace and `PrefillOrders` is the Apex class.

 **Note** Users who use Remote Actions in Integration Procedures must have access to the object and record they're invoking through the action. If they receive an access error, they must be granted object and record access at the profile, permission set, or permission set group level.

To ensure that a Remote Action error causes a rollback if **Rollback on Error** is checked in the Procedure Configuration, set the `errorCode` in the class output map.

 **Note** Community Plus users must log in before they can use an Integration Procedure Remote Action that sends emails.

[Create a Remote Action Example for an Invocable Action](#)

This example Integration Procedure uses a Remote Action to call the `emailSimple` Invocable Action, which is present in every Salesforce org. Inputs to the Invocable Action are specified in the Integration Procedure as Additional Input.

[Create a Remote Action Example for an Invocable Action with an Input Key](#)

An Integration Procedure uses a Remote Action to call the emailSimple Invocable Action, which is present in every Salesforce org. Inputs to the Invocable Action are specified in the Integration Procedure in a Remote Option named InvocableInputKey.

Create a Remote Action Apex Class Example

This example Integration Procedure uses a Remote Action to call an Apex class and demonstrate how to reserialize data.

Create a Remote Action Example for an Invocable Action

This example Integration Procedure uses a Remote Action to call the emailSimple Invocable Action, which is present in every Salesforce org. Inputs to the Invocable Action are specified in the Integration Procedure as Additional Input.

1. Create a new Integration Procedure.
2. Add a Remote Action component with these settings:

Property	Value
Remote Class	DefaultInvocableAction
Remote Method	emailSimple

3. Under Additional Input, add these key-value pairs:

Key	Value
emailAddresses	%SendTo%
emailSubject	%Subject%
emailBody	%Message%

4. Add a Response Action into the Structure panel and select **Return entire JSON output**. If you're using the designer on a managed package, select **Return Full Data JSON**.
5. Go to the Preview tab.
6. Under Input Parameters, add these key-value pairs:

Key	Value
SendTo	(your email)

Key	Value
Subject	Invocable Test
Message	This is a test.

7. Click **Execute**.

If you set up the Integration Procedure correctly, an email titled **[RD] Invocable Test [sent via RD-PRD mail server]** arrives in your Inbox.

Create a Remote Action Example for an Invocable Action with an Input Key

An Integration Procedure uses a Remote Action to call the emailSimple Invocable Action, which is present in every Salesforce org. Inputs to the Invocable Action are specified in the Integration Procedure in a Remote Option named InvocableInputKey.

1. Create a new Integration Procedure.
2. Add a Remote Action component with these settings.

Property	Value
Remote Class	DefaultInvocableAction
Remote Method	emailSimple
Remote Options, Key	InvocableInputKey
	InvocableActionType
Remote Options, Value	Input
	decisionTableAction

3. Add a Response Action into the Structure panel and select **Return entire JSON output**. If you're using the designer on a managed package, select **Return Full Data JSON**.
4. On the Preview tab, click **Edit as JSON**.
5. In the Input Parameters panel, provide input with this structure, substituting your email:

```
{
  "Input": {
    "emailAddresses": "jdoe@example.com",
    "emailSubject": "Invocable Test",
```

```
        "emailBody": "This is a test."  
    }  
}
```

6. Click **Execute**.

If you set up the Integration Procedure correctly, an email titled **[RD] Invocable Test [sent via RD-PRD mail server]** arrives in your Inbox.

Create a Remote Action Apex Class Example

This example Integration Procedure uses a Remote Action to call an Apex class and demonstrate how to reserialize data.

If you're using the designer on a managed package, you can download a data pack of this example from [here](#). The data pack doesn't include the Apex class.

Importing Omnistudio for Managed Packages data packs into Omnistudio orgs is supported, and conversion is automatic. But importing Omnistudio data packs into Omnistudio for Managed Packages orgs isn't supported.

To build this example:

1. [Create the TestReserializeApex Class](#).
2. [Create the Integration Procedure with the Remote Action](#).
3. [Test the Integration Procedure with the Remote Action](#).

[Create the TestReserializeApex Class](#)

Before you can create the Integration Procedure example, you must create the *TestReserializeApex* Apex class that it calls.

[Create the Integration Procedure with the Remote Action](#)

An Integration Procedure retrieves and reserializes the value of a `name` variable from the *TestReserializeApex* Apex class.

[Test the Integration Procedure with the Remote Action](#)

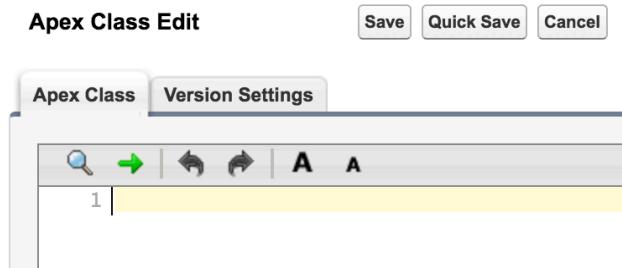
After you have created the example Integration Procedure with the Remote Action, the final task is to test it.

Create the TestReserializeApex Class

Before you can create the Integration Procedure example, you must create the *TestReserializeApex* Apex class that it calls.

1. From Setup, in the **Quick Find** box, type `apex`.
2. Click **Apex Classes**.
3. Click **New**.

Apex Class



- Enter the following Apex code in the **Apex Class** tab:

```
@JsonAccess(serializable='always')
global with sharing class TestReserializeApex implements Callable {
    public Object call(String action, Map<String, Object> args) {
        Map<String, Object> input = (Map<String, Object>)args.get('input');
        Map<String, Object> output = (Map<String, Object>)args.get('output');
        Map<String, Object> options = (Map<String, Object>)args.get('options');
        return invokeMethod(action, input, output, options);
    }
    private Object invokeMethod(String methodName, Map<String, Object> input,
        Map<String, Object> output, Map<String, Object> options) {
        return output.put('result', new TestReserialize());
    }
    @JsonAccess(serializable='always')
    global class TestReserialize {
        global String name = 'Dave Smith';
    }
}
```

Also note the use of the [JsonAccess annotation](#) on the class and method.

- Click **Save**.

What's next: [Create the Integration Procedure with the Remote Action](#).

Create the Integration Procedure with the Remote Action

An Integration Procedure retrieves and reserializes the value of a `name` variable from the `TestReserializeApex` Apex class.

The Integration Procedure has these components:

- A Remote Action, named ReserializeApex
- A Set Values component, named ReserializeApexFunction

- A Set Values component, named MyName
- A Response Action, named ResponseAction1

To build this Integration Procedure:

1. Create a new Integration Procedure.
2. Add a Remote Action component with these settings.

Property	Value
Element Name	ReserializeApex
Remote Class	TestReserializeApex
Remote Method	random
Remote Options, Key	reserialize
Remote Options, Value	false

A **Remote Method** value is required, but it can be anything, because this Integration Procedure doesn't invoke a method.

A step in the test of this Integration Procedure will ask you to change the **Remote Options Value**.

3. Add a Set Values component into the Structure panel and configure it with these settings:

Property	Value
Element Name	ReserializeApexFunction
Element Value Map, Element Name	result
Element Value Map, Value	=RESERIALIZE(%ReserializeApex%)
Response JSON Path	result
Response JSON Node	ReserializeApex

4. Add another Set Values component into the Structure panel and configure it with these settings:

Property	Value
Element Name	MyName
Element Value Map, Element Name	Full Name
Element Value Map, Value	%ReserializeApex:result:name%

5. Add a Response Action component into the Structure panel and select **Return entire JSON output**. If you're using the designer on a managed package, select **Return Full Data JSON**.

What's next: [Test the Integration Procedure with the Remote Action](#).

Test the Integration Procedure with the Remote Action

After you have created the example Integration Procedure with the Remote Action, the final task is to test it.

1. Go to the Preview tab and click **Execute**. The output should look like this:

```
{
  "response": {},
  "vlcchainableStepDepth": 0,
  "ResponseAction1Status": true,
  "MyName": {
    "Full Name": "Dave Smith"
  },
  "MyNameStatus": true,
  "ReserializeApexFunctionStatus": true,
  "ReserializeApex": {
    "result": {
      "name": "Dave Smith"
    },
    "errorCode": "INVOKE-200",
    "error": "OK"
  },
  "ReserializeApexStatus": true,
  "options": {
    "queueableChainable": false,
    "ignoreCache": true,
    "resetCache": false,
    "chainable": false
  }
}
```

```
}
```

Note that this Integration Procedure successfully passes the value of the `name` variable in the Apex class to its `Full Name` node.

2. Go to the Properties tab, select the ReserializeApexFunction component, and click **Active** to remove the check. This turns off reserialization.
3. Go to the Preview tab and click **Execute** again. The output should look like this:

```
{
  "response": {},
  "vlcchainableStepDepth": 0,
  "ResponseAction1Status": true,
  "MyName": {
    "Full Name": ""
  },
  "MyNameStatus": true,
  "ReserializeApex": {
    "result": {
      "name": "Dave Smith"
    },
    "errorCode": "INVOKE-200",
    "error": "OK"
  },
  "ReserializeApexStatus": true,
  "options": {
    "queueableChainable": false,
    "ignoreCache": true,
    "resetCache": false,
    "chainable": false
  }
}
```

Note that the `Full Name` node has no value.

4. Go to the Properties tab, select the ReserializeApex component, and change the **Remote Options Value** to `true`. This enables reserialization in a different way.
5. Go to the Preview tab and click **Execute** a third time. The output should look like this:

```
{
  "response": {},
  "vlcchainableStepDepth": 0,
  "ResponseAction1Status": true,
  "MyName": {
    "Full Name": "Dave Smith"
  }
}
```

```
        },
        "MyNameStatus": true,
        "ReserializeApex": {
            "error": "OK",
            "errorCode": "INVOKE-200",
            "result": {
                "name": "Dave Smith"
            }
        },
        "ReserializeApexStatus": true,
        "options": {
            "queueableChainable": false,
            "ignoreCache": true,
            "resetCache": false,
            "chainable": false
        }
    }
}
```

Note that the `Full Name` node has a value again.

Improve Performance by Using Caching

Omnistudio offers multiple caching options to improve performance and minimize unnecessary data processing in Integration Procedures. The two primary user-configurable caching types are Org Cache and Session Cache, which are available in Integration Procedure designer. These caching mechanisms are part of the Salesforce Platform Cache and are implemented through Scale Cache in Omnistudio with standard runtime. Each serves a distinct purpose and comes with specific configuration considerations.

Omnistudio also offers Metadata Cache, which is for internal use only. It's not configurable by users. It stores Omnistudio component metadata or definitions, and not the actual data. You can configure the data to remain in the cache (time to live) for a maximum of 48 to 72 hours.

Omnistudio caches two different types of content:

- **Metadata:** The definition that includes the structure and configuration of Integration Procedures and Data Mappers, such as steps, mappings, and formulas. This is stored in Metadata Cache and is used to improve load times.
- **Data:** The response of a Data Mapper or an Integration Procedure that is cached in the Session Cache or Org Cache. Every time, when the same request input is entered or provided, the cached data is returned.

Types of data cache in Integration Procedures include:

- **Org Cache:** The cached data is shared among all users in the same org. The org cache is recommended only if the data is non-sensitive or not user-specific. For example, picklists, configurations, or object

data without any access restrictions. You can configure the data to remain in the cache (time to live) for a maximum of 48 hours.

 **Important** Use org cache only for non-sensitive data.

- Session Cache: It stores user-specific data and is limited to active user sessions. It is safer for personalized data and helps reduce redundant processing for data that varies by user profile or permissions. You can configure the data to remain in the cache (time to live) for a maximum of 8 hours.

You can use caching with Integration Procedures in three ways:

- You can cache metadata for the entire Integration Procedure.
- You can cache the response of the entire Integration Procedure, called top-level data. See [Cache for Top-Level Integration Procedure Data](#).
- You can cache the result of a specific set of steps by placing the steps inside a Cache Block. See [Enhance Performance by Using Cache Blocks](#).

Use Cache Blocks if some parts of the Integration Procedure update data, or if you need different cached data to expire at different times. For example, current weather data changes more frequently than user session data.

You can also perform a record-level security check for cached data. See [Security for Omnistudio Data Mappers and Integration Procedures](#).

Metadata Cache for Integration Procedures

To disable metadata caching for an Integration Procedure, go to the Procedure Configuration and check the **Disable Definition Cache** checkbox.

 **Tip** To test the performance benefit of metadata caching, execute the Integration Procedure in the Preview tab with **Disable Definition Cache** checked and then unchecked. Compare the Browser, Server, and Apex CPU values.

Methods to Clear Metadata from the Integration Procedure Cache

You can clear Integration Procedure metadata from the cache. You can also clear all cached data for an Integration Procedure, including session cache data, org cache data, and metadata.

1. Go to the Developer Console.
2. Click the user menu and select **Developer Console** from the menu.
3. Select **Debug | Open Execute Anonymous Window**.
4. To clear metadata cache for an Integration Procedure, in the Apex code window, execute this code:

```
ConnectApi.IntegrationProcedureCacheInputRepresentation finalInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation();
ConnectApi.IntegrationProcedureCacheInputRepresentation apexInput = new ConnectApi.Int
```

```
egrationProcedureCacheInputData();
apexInput.ipKey = ipKey;

List l = new List();
l.add(apexInput);
finalInput.ipInput = l;
finalInput.cacheStorageType = ConnectApi.CacheStorageType.Metadata;
ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.O
mniDesignerConnect.ClearIntegrationProcedureCache(finalInput);
```

Here, *ipkey* specifies the Integration Procedure to invoke in the Type_Subtype format.



Note Starting with Summer '25, replace the

`namespace.IntegrationProcedureService.clearMetadataCache('Type_Subtype')`
method with the Connect API to clear Integration Procedure metadata from the cache.

5. To clear all cached data for an Integration Procedure, including session cache data, org cache data, and metadata, in the Apex code window, execute this code:

```
ConnectApi.IntegrationProcedureCacheInputRepresentation finalInput = new Con
nectApi.IntegrationProcedureCacheInputRepresentation();
ConnectApi.IntegrationProcedureCacheInputData apexInput = new ConnectApi.Int
egrationProcedureCacheInputData();
apexInput.ipKey = ipKey;
List l = new List();
l.add(apexInput);
finalInput.ipInput = l;
finalInput.cacheStorageType = ConnectApi.CacheStorageType.All;
ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.O
mniDesignerConnect.ClearIntegrationProcedureCache(finalInput);
```



Note Starting with Summer '25, replace the

`omnistudio.IntegrationProcedureService.clearAllCache('Type_Subtype')`
method with the Connect API to clear all cached data from an Integration Procedure.

Cache for Top-Level Integration Procedure Data

Using a cache to store frequently accessed, infrequently updated Integration Procedure data saves round trips to the database and improves performance. You can cache all the data for an Integration Procedure, as described here, or you can use a Cache Block to cache only part of it.

Configure Top-Level Caching for an Integration Procedure

To configure top-level caching for an Integration Procedure, go to the Procedure Configuration and set the **Salesforce Platform Cache Type** and **Time To Live In Minutes** properties.

Create a Top-Level Caching Example

An Integration Procedure accepts a list of first or last names and retrieves Contacts having those

names. Top-level caching to the VlocityAPIResponse partition improves Contact retrieval performance.

Turn Off the Scale Cache

Turn off the Scale Cache globally when working with sensitive or user-specific data. This prevents such data from being cached and ensures real-time access control and data accuracy.

Connect APIs for Cache Management in Integration Procedures

See this table for a summary of Connect APIs to call Integration Procedures from Apex classes.

Starting Summer '25, replace the existing methods in the *IntegrationProcedureService* Apex classes with the Connect APIs.

Enhance Performance by Using Cache Blocks

Use Cache Blocks if some parts of the Integration Procedure update data, or if you need different cached data to expire at different times. For example, current weather data changes more frequently than user session data.

Cache for Top-Level Integration Procedure Data

Using a cache to store frequently accessed, infrequently updated Integration Procedure data saves round trips to the database and improves performance. You can cache all the data for an Integration Procedure, as described here, or you can use a Cache Block to cache only part of it.

To configure top-level caching for an Integration Procedure, go to the Procedure Configuration and set the **Salesforce Platform Cache Type** and **Time To Live In Minutes** properties. See [Configure Top-Level Caching for an Integration Procedure](#).

 **Note** If an Integration Procedure that has top-level caching enabled fails, its data isn't cached.

Options in Preview for Top-Level Caching

When you test an Integration Procedure that uses top-level caching in the Preview tab, you can use two caching settings in the Options JSON section. These settings control top-level data and have no effect on the metadata cache.

- `ignoreCache` – Doesn't clear or save data to the cache. The default value is `true`. Use this setting to test Integration Procedure steps without the possible interference of caching effects.
- `resetCache` – Forces data to be saved to the cache. The default value is `false`. Use this setting as part of testing caching itself.

 **Note** To test caching, be sure to set `ignoreCache` to `false`. See [Create a Top-Level Caching Example](#).

You can pass `ignoreCache` and `resetCache` as parameters when you invoke an Integration Procedure that uses caching using a REST API. For example, you can include `?resetCache=true` in the URL to force caching. See [Integration Procedure Invocation Using POST](#).

Top-Level Caching JSON Nodes and REST Headers

If top-level caching is configured and the Integration Procedure is active, the Integration Procedure JSON can include the following nodes under the root node:

- `vlcCacheKey` – Key for any data stored in the cache
- `vlcCacheResult` – Included and set to true if data is retrieved from the cache
- `vlcCacheEnabled` – Included and set to false if the `ignoreCache` setting disables caching
- `vlcCacheException` – Any caching errors

These nodes are returned as headers if you invoke an Integration Procedure that uses top-level caching using a REST API. See [Integration Procedure Invocation Using POST](#).

Methods for Clearing Top-Level Data

To clear top-level Integration Procedure data from the cache, execute these Connect APIs in the Developer Console.

-  **Note** Starting with Summer '25, replace the `clearSessionCache`, `clearOrgCache`, and `clearAllCache` methods with the Connect APIs to clear top-level Integration Procedure data from the cache.

- Clear session cache:

Starting with Summer '25, replace the

`IntegrationProcedureService.clearSessionCache('Type_Subtype', new Map<String, Object>{'key' => 'value'})` method with this API:

```
ConnectApi.IntegrationProcedureCacheInputRepresentation finalInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation();
ConnectApi.IntegrationProcedureCacheInputData apexInput = new ConnectApi.IntegrationProcedureCacheInputData();
apexInput.ipKey = ipKey;
apexInput.inputData = JSON.serialize(inputData);
List l = new List();
l.add(apexInput);
finalInput.ipInput = l;
finalInput.cacheStorageType = ConnectApi.CacheStorageType.Session;
ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(finalInput);
```

ipKey: Specifies the Integration Procedure to invoke in Type_Subtype format.

inputData: Specifies the input in the Map<string,object> format.

- Clear session cache with `vlcCacheKey`:

Starting with Summer '25, replace the

`IntegrationProcedureService.clearSessionCache('vlcCacheKey')` method with this API:

```
ConnectApi.IntegrationProcedureCacheInputRepresentation apexInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation();
List cacheKeyList = new List();
cacheKeyList.add(vlcCacheKey);
apexInput.cacheKeys= cacheKeyList;
apexInput.cacheStorageType = ConnectApi.CacheStorageType.Session;
ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(apexInput);
```

- Clear org cache:

Starting with Summer '25, replace the

```
IntegrationProcedureService.clearOrgCache('Type_Subtype', new Map<String, Object>{'key' => 'value'})
```

method with this API:

```
ConnectApi.IntegrationProcedureCacheInputRepresentation finalInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation();
ConnectApi.IntegrationProcedureCacheInputData apexInput = new ConnectApi.IntegrationProcedureCacheInputData();
apexInput.ipKey = ipKey;
apexInput.inputData = JSON.serialize(inputData);
List<ConnectApi.IntegrationProcedureCacheInputData> l = new List<ConnectApi.IntegrationProcedureCacheInputData>();
l.add(apexInput);
finalInput.ipInput = l;
finalInput.cacheStorageType = ConnectApi.CacheStorageType.Org;
ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(finalInput);
```

- Clear org cache with `vlcCacheKey`:

Starting with Summer '25, replace the

```
IntegrationProcedureService.clearOrgCache('vlcCacheKey')
```

method with this API:

```
ConnectApi.IntegrationProcedureCacheInputRepresentation apexInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation();
List<String> cacheKeyList = new List<String>();
cacheKeyList.add(vlcCacheKey);
apexInput.cacheKeys= cacheKeyList;
apexInput.cacheStorageType = ConnectApi.CacheStorageType.Org;
ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(apexInput);
```

- Clear all cached data for an Integration Procedure, including session cache data, org cache data, and metadata:

Starting with Summer '25, replace the

```
IntegrationProcedureService.clearAllCache('Type_Subtype')
```

method with this API:

```
ConnectApi.IntegrationProcedureCacheInputRepresentation finalInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation();
ConnectApi.IntegrationProcedureCacheInputData apexInput = new ConnectApi.IntegrationProcedureCacheInputData();
apexInput.ipKey = ipKey;
List<ConnectApi.IntegrationProcedureCacheInputData> l = new List<ConnectApi.IntegrationProcedureCacheInputData>();
l.add(apexInput);
finalInput.ipInput = l;
finalInput.cacheStorageType = ConnectApi.CacheStorageType.All;
ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(finalInput);
```

Configure Top-Level Caching for an Integration Procedure

To configure top-level caching for an Integration Procedure, go to the Procedure Configuration and set the **Salesforce Platform Cache Type** and **Time To Live In Minutes** properties.

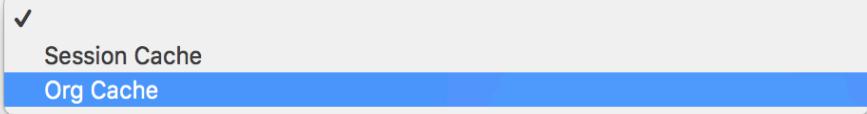
1. Go to the Integration Procedure configuration page.
2. In the Cache Configuration section, set the **Salesforce Platform Cache Type** to one of these:
 - Blank – For no caching
 - **Session Cache** – For data related to users and their login sessions
 - **Org Cache** – For all other types of data

Important Use Org Cache only for non-sensitive data.

▼ CACHE CONFIGURATION

Disable Definition Cache

Salesforce Platform Cache Type



Time To Live In Minutes ⓘ

5

Behind this drop-down is the IsCacheable__c picklist.

3. Specify a value for **Time To Live In Minutes**. This setting determines how long data remains in the cache. The minimum value is 5. Default and maximum values depend on the cache type:
 - **Session Cache** – The default value is 5. The maximum value is 480, equivalent to 8 hours. The cache is cleared when the user's session expires regardless of this value.
 - **Org Cache** – The default value is 1440, equivalent to 24 hours. The maximum value is 2880,

equivalent to 48 hours.

Top-level caching overrides Cache Block caching for the duration of the top-level **Time To Live In Minutes** value. Top-level and Cache Block data is cleared when the Integration Procedure is deactivated regardless of this value.

4. Click **Activate Version**. Top-level caching occurs only if the Integration Procedure is active.

Create a Top-Level Caching Example

An Integration Procedure accepts a list of first or last names and retrieves Contacts having those names. Top-level caching to the VlocityAPIResponse partition improves Contact retrieval performance.

This Integration Procedure also includes a Loop Block; see [Work with Data and Lists](#).

To download a DataPack of this example for Omnistudio, click [here](#).

This Integration Procedure has these components:

- A Loop Block, named LoopBlock1
- An Omnistudio Data Mapper Extract Action within the Loop Block, named ExtractContact
- A Response Action, named ResponseAction

To build this Integration Procedure:

1. From the Omnistudio Integration Procedures tab, click **New**.
2. Provide an **Integration Procedure Name**, a **Type**, and a **SubType**, and click **Save**.
3. In the Procedure Configuration, set the Salesforce Platform Cache Type to **Org Cache**.
4. Drag a Loop Block component into the Structure panel. Give the Loop Block the following settings:

Property	Value
Loop List	names
Additional Loop Output	<code>ExtractContact</code> set to <code>%ExtractContact%</code>

5. Create the Data Mapper Extract that the Data Mapper Extract Action calls, name it ExtractContactName, and give it the following settings:

Tab	Settings
Extract	A Contact extract step set to <code>Contact Name LIKE name</code>

Tab	Settings
Output	A mapping from <code>Contact:Id</code> to <code>Contact:Id</code> A mapping from <code>Contact:Name</code> to <code>Contact:Name</code>

If you aren't sure how to create a Data Mapper Extract, see the examples in [Omnistudio Data Mapper Extract Examples](#).

- Drag a Data Mapper Extract Action inside the Loop Block and give it the following settings:

Property	Value
Element Name	ExtractContact
Data Mapper Interface	ExtractContactName
Data Source	names:name
Filter Value	name

- Create the Response Action below the Loop Block and give it the following settings:

Property	Value
Send JSON Path	LoopBlock1:ExtractContact
Response JSON Node	Contact

- In the Procedure Configuration, click **Activate Version**.

- On the Preview Tab, click **Edit as JSON** and provide input with the following structure:

```
{
  "names": [
    {
      "name": "Miller"
    },
    {
      "name": "Torres"
    }
]
```

```
}
```

To demonstrate performance most effectively, specify common names or many names.

10. Click **Execute** and note the resulting Browser, Server, and Apex CPU values.
11. Open the Options pane and change the `ignoreCache` value to `false`.
12. Click **Execute** again. This step populates the cache, so the resulting Browser, Server, and Apex CPU values are likely to be similar to the previous values.
13. Click **Execute** a third time. This step uses the cached values, so the resulting Browser, Server, and Apex CPU values are likely to be noticeably less than the previous values. In addition, `Cached Response: true` appears after the performance metrics.

Turn Off the Scale Cache

Turn off the Scale Cache globally when working with sensitive or user-specific data. This prevents such data from being cached and ensures real-time access control and data accuracy.

To turn off the Scale Cache that Omnistudio Data Mappers and Integration Procedures use, go to Setup and add the **TurnOffScaleCache** setting to the Omni Interaction Configuration.

1. Go to Setup.
2. In the **Quick Find** box, type `omni`, then select **Omni Interaction Configuration**.
3. Click **New Omni Interaction Configuration**.
4. In the **Label** field, type `TurnOffScaleCache`.
5. In the **Value** field, type `true`.
6. Click **Save**.

Connect APIs for Cache Management in Integration Procedures

See this table for a summary of Connect APIs to call Integration Procedures from Apex classes. Starting Summer '25, replace the existing methods in the `IntegrationProcedureService` Apex classes with the Connect APIs.

Purpose	Connect API	Existing Method (Replace with Connect API)
Clear org cache for an Integration Procedure.	<pre>ConnectApi.IntegrationProcedureCacheInputRepresentation finalInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation();</pre>	<pre>IntegrationProcedureService.clearOrgCache(ipKey, inputData);</pre>

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre data-bbox="621 291 997 1438"> ConnectApi.Integration ProcedureCacheInputDat a apexInput = new Conn ectApi.IntegrationProc edureCacheInputData(); apexInput.ipKey = <i>ipKe y</i>; apexInput.inputData = JSON.serialize(<i>inputDa ta</i>); List<ConnectApi.Integr ationProcedureCacheInp utData> l = new List<C onnectApi.IntegrationP rocedureCacheInputDat a>(); l.add(apexInput); finalInput.ipInput = l; finalInput.cacheStorag eType = ConnectApi.Cac heStorageType.Org; ConnectApi.Integration ProcedureCacheOutputRe presentation test = Co nnectApi.OmniDesignerC onnect.ClearIntegratio nProcedureCache(finalI nput); </pre> <p data-bbox="616 1495 1008 1611"><i>ipKey</i>: Specifies the Integration Procedure to invoke in Type_Subtype format.</p> <p data-bbox="616 1643 1013 1717"><i>inputData</i>: Specifies the input in the Map<string,object> format.</p> <p data-bbox="616 1755 731 1786">Example:</p> <div data-bbox="633 1845 992 1877" style="border: 1px solid #ccc; padding: 5px; width: fit-content;">ConnectApi.Integration</div>	

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre data-bbox="633 291 998 1759"> ProcedureCacheInputRepresentation finalInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation(); ConnectApi.IntegrationProcedureCacheInputData apexInput = new ConnectApi.IntegrationProcedureCacheInputData(); apexInput.ipKey = 'LastNames_Cached'; Map<String, Object> map = new Map<String, Object>'ContactLastName' => 'Smith'; apexInput.inputData = JSON.serialize(map); List<ConnectApi.IntegrationProcedureCacheInputData> l = new List<ConnectApi.IntegrationProcedureCacheInputData>(); l.add(apexInput); finalInput.ipInput = l; finalInput.cacheStorageType = ConnectApi.CacheStorageType.Org; ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(finalInput); </pre>	
Clear org cache with vlcCacheKeys	ConnectApi.Integration	IntegrationProcedureSe

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre data-bbox="633 291 997 1153"> ProcedureCacheInputRepresentation apexInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation(); List<String> cacheKeyList = new List<String>(); cacheKeyList.add(cacheKey); apexInput.cacheKeys= cacheKeyList; apexInput.cacheStorageType = ConnectApi.CacheStorageType.Org; ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(apexInput); </pre>	<pre data-bbox="1062 291 1416 361"> rvice.clearOrgCache(cacheKey); </pre>

Example:

```

ConnectApi.IntegrationProcedureCacheInputRepresentation apexInput
= new ConnectApi.IntegrationProcedureCacheInputRepresentation();
List<String> cacheKeyList = new List<String>();
cacheKeyList.add('2032076016a1745801061oc');
apexInput.cacheKeys= cacheKeyList;
apexInput.cacheStorageType = ConnectApi.CacheStorageType.Org;

```

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre data-bbox="633 291 997 597"> eStorageType.Org; ConnectApi.Integration ProcedureCacheOutputRe presentation test = Co nnectApi.OmniDesignerC onnect.ClearIntegratio nProcedureCache(apexIn put); </pre>	
Clear session cache for an Integration Procedure	<pre data-bbox="633 682 997 1888"> ConnectApi.Integration ProcedureCacheInputRep resentation finalInput = new ConnectApi.Integ rationProcedureCacheIn putRepresentation(); ConnectApi.Integration ProcedureCacheInputDat a apexInput = new Conn ectApi.IntegrationProc edureCacheInputData(); apexInput.ipKey = ipKe y; apexInput.inputData = JSON.serialize(inputDa ta); List<ConnectApi.Integr ationProcedureCacheInp utData> l = new List<C onnectApi.IntegrationP rocedureCacheInputDat a>(); l.add(apexInput); finalInput.ipInput = l; finalInput.cacheStorag eType = ConnectApi.Cac heStorageType.Session; ConnectApi.Integration ProcedureCacheOutputRe </pre>	<pre data-bbox="1057 682 1421 804"> IntegrationProcedureSe rvice.clearSessionCach e(ipKey, inputData); </pre>

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre>presentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(finalInput);</pre> <p>Example:</p> <pre>ConnectApi.IntegrationProcedureCacheInputRepresentation finalInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation(); ConnectApi.IntegrationProcedureCacheInputData apexInput = new ConnectApi.IntegrationProcedureCacheInputData(); apexInput.ipKey = 'LastNames_Cached'; Map<String, Object> map = new Map<String, Object>{'ContactLastName' => 'Smith'}; apexInput.inputData = JSON.serialize(map); List<ConnectApi.IntegrationProcedureCacheInputData> l = new List<ConnectApi.IntegrationProcedureCacheInputData>(); l.add(apexInput); finalInput.ipInput = l; finalInput.cacheStorageType = ConnectApi.CacheStorageType.Session;</pre>	

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre data-bbox="633 291 997 508">ProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(finalInput);</pre>	
Clear session cache with vlcCacheKeys	<pre data-bbox="633 614 997 1516">ConnectApi.IntegrationProcedureCacheInputRepresentation apexInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation(); List<String> cacheKeyList = new List<String>(); cacheKeyList.add(cacheKey); apexInput.cacheKeys= cacheKeyList; apexInput.cacheStorageType = ConnectApi.CacheStorageType.Session; ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(apexInput);</pre> <p data-bbox="616 1579 731 1613">Example:</p> <pre data-bbox="633 1670 997 1896">ConnectApi.IntegrationProcedureCacheInputRepresentation apexInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation();</pre>	<pre data-bbox="1057 614 1421 720">IntegrationProcedureService.clearSessionCache(cacheKey);</pre>

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre data-bbox="633 291 997 956"> List<String> cacheKeyList = new List<String>(); cacheKeyList.add('2032076016a1745801061oc'); apexInput.cacheKeys= cacheKeyList; apexInput.cacheStorageType = ConnectApi.CacheStorageType.Session; ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(apexInput); </pre>	
Clear all cached data for an Integration Procedure	<pre data-bbox="633 1051 997 1886"> ConnectApi.IntegrationProcedureCacheInputRepresentation finalInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation(); ConnectApi.IntegrationProcedureCacheInputData apexInput = new ConnectApi.IntegrationProcedureCacheInputData(); apexInput.ipKey = ipKey; List<ConnectApi.IntegrationProcedureCacheInputData> l = new List<ConnectApi.IntegrationProcedureCacheInputData>(); l.add(apexInput); finalInput.ipInput = </pre>	<pre data-bbox="1062 1051 1426 1157"> IntegrationProcedureService.clearAllCache(ipKey); </pre>

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre data-bbox="621 276 1008 713"> l; finalInput.cacheStorageType = ConnectApi.CacheStorageType.All; ConnectApi.IntegrationProcedureCacheOutputRepresentation test = ConnectApi.OmniDesignerConnect.ClearIntegrationProcedureCache(finalInput); </pre> <p data-bbox="616 777 731 811">Example:</p> <pre data-bbox="621 868 1008 1898"> ConnectApi.IntegrationProcedureCacheInputRepresentation finalInput = new ConnectApi.IntegrationProcedureCacheInputRepresentation(); ConnectApi.IntegrationProcedureCacheInputData apexInput = new ConnectApi.IntegrationProcedureCacheInputData(); apexInput.ipKey = 'LastNames_Cached'; List<ConnectApi.IntegrationProcedureCacheInputData> l = new List<ConnectApi.IntegrationProcedureCacheInputData>(); l.add(apexInput); finalInput.ipInput = l; finalInput.cacheStorageType = ConnectApi.CacheStorageType.All; ConnectApi.Integration </pre>	

Purpose	Connect API	Existing Method (Replace with Connect API)
	<pre data-bbox="633 291 997 508">ProcedureCacheOutputRe presentation test = Co nnectApi.OmniDesignerC onnect.ClearIntegratio nProcedureCache(finalI nput);</pre>	
Clear metadata cache for an Integration Procedure	<pre data-bbox="633 614 997 1881">ConnectApi.Integration ProcedureCacheInputRep resentation finalInput = new ConnectApi.Integ rationProcedureCacheIn putRepresentation(); ConnectApi.Integration ProcedureCacheInputDat a apexInput = new Conn ectApi.IntegrationProc edureCacheInputData(); apexInput.ipKey = ipKe y; List<ConnectApi.Integr ationProcedureCacheInp utData> l = new List<C onnectApi.IntegrationP rocedureCacheInputDat a>(); l.add(apexInput); finalInput.ipInput = l; finalInput.cacheStorag eType = ConnectApi.Cac heStorageType.Metadata; ConnectApi.Integration ProcedureCacheOutputRe presentation test = Co nnectApi.OmniDesignerC onnect.ClearIntegratio nProcedureCache(finalI nput);</pre>	<pre data-bbox="1057 614 1421 720">IntegrationProcedureSe rvice.clearMetadataCac he(ipKey);</pre>

Purpose	Connect API	Existing Method (Replace with Connect API)
	<p>Example:</p> <pre data-bbox="616 361 1013 1706"> ConnectApi.Integration ProcedureCacheInputRep resentation finalInput = new ConnectApi.Integ rationProcedureCacheIn putRepresentation(); ConnectApi.Integration ProcedureCacheInputDat a apexInput = new Conn ectApi.IntegrationProc edureCacheInputData(); apexInput.ipKey = 'Las tNames_Cached'; List<ConnectApi.Integr ationProcedureCacheInp utData> l = new List<C onnectApi.IntegrationP rocedureCacheInputDat a>(); l.add(apexInput); finalInput.ipInput = l; finalInput.cacheStorag eType = ConnectApi.Cac heStorageType.Metadat a; ConnectApi.Integration ProcedureCacheOutputRe presentation test = Co nnectApi.OmniDesignerC onnect.ClearIntegratio nProcedureCache(finalI nput);</pre>	

Enhance Performance by Using Cache Blocks

Use Cache Blocks if some parts of the Integration Procedure update data, or if you need different cached

data to expire at different times. For example, current weather data changes more frequently than user session data.

It's important to understand how top-level Integration Procedure caching interacts with Cache Blocks. See [Improving Data Mapper Performance with Caching](#).

A Cache Block saves the output of the steps within it to a session or org cache in the VlocityAPIResponse cache partition for quick retrieval.

Test Options for Cache-Block Caching in Preview

When you test an Integration Procedure that includes a Cache Block in the Preview tab, you can use two caching settings in the Options JSON section. These settings also affect top-level caching but have no effect on the metadata cache.

- `ignoreCache` – Doesn't clear or save data to the cache. The default value is `true`. Use this setting to test Integration Procedure steps without the possible interference of caching effects.
- `resetCache` – Forces data to be saved to the cache. The default value is `false`. Use this setting as part of testing caching itself.

 **Note** To test caching, be sure to set `ignoreCache` to `false`. See [Create a Cache Block Example](#).

You can pass `ignoreCache` and `resetCache` as parameters when you invoke an Integration Procedure that uses caching using a REST API. For example, you can include `?resetCache=true` in the URL to force caching. See [Integration Procedure Invocation Using POST](#).

Cache Block JSON Nodes

An active Integration Procedure that uses a Cache Block can include the following JSON nodes in its Debug Log output, which is visible on the Preview tab:

- `vlcCacheKey` – Key for any data stored in the cache
- `vlcCacheResult` – Included and set to true if data is retrieved from the cache
- `vlcCacheEnabled` – Included and set to false if the `ignoreCache` setting disables caching
- `vlcCacheException` – Any caching errors

These nodes are under the `Info` node for the Cache Block. For example, if the Cache Block is named CacheBlock1, these nodes are under the `CacheBlock1Info` node.

Apex Methods to Clear Cache Block Data

If you must clear Cache Block data from the cache, follow the instructions in [Fix the DataPack Import or Export Error: No Configuration Found](#), but execute one of these lines of code instead:

```
IntegrationProcedureService.clearSessionCache('Type_Subtype', 'blockName', new Map<String, Object>{'key' => 'value'});  
  
IntegrationProcedureService.clearOrgCache('Type_Subtype', 'blockName', new Map<String, Object>{'key' => 'value'});  
  
IntegrationProcedureService.clearSessionCache('vlcCacheKey');  
  
IntegrationProcedureService.clearOrgCache('vlcCacheKey');
```

You can clear all cached data for an Integration Procedure, including session cache data, org cache data, and metadata. Follow the instructions in [Fix the DataPack Import or Export Error: No Configuration Found](#), but execute this line of code instead, specifying the Integration Procedure's Type and Subtype:

```
IntegrationProcedureService.clearAllCache('Type_Subtype');
```

For example, execute this code if:

- You want to clear a Cache Block key in the Org Cache
- The Type_Subtype parameter for the Integration Procedure is LastNames_Cached
- The Cache Block is named CacheContacts
- The cache key you want to clear is ContactLastName
- The key's value is Smith

```
IntegrationProcedureService.clearOrgCache('LastNames_Cached', 'CacheContacts', new Map<String, Object>{'ContactLastName' => 'Smith'});
```

The following example clears the session cache using a vlcCacheKey value:

```
IntegrationProcedureService.clearSessionCache('2032076016a1745801061oc');
```

Create a Cache Block Example

In this example, an Integration Procedure accepts a list of first or last names and retrieves Contacts having those names. A Cache Block improves Contact retrieval performance.

Create a Cache Block Example

In this example, an Integration Procedure accepts a list of first or last names and retrieves Contacts having those names. A Cache Block improves Contact retrieval performance.

In this example, the Integration Procedure has these components:

- A Cache Block, named CacheBlock1

- A Loop Block within the Cache Block, named LoopBlock1
- An Omnistudio Data Mapper Extract Action within the Loop Block, named ExtractContact
- A Response Action, named ResponseAction

To build this Integration Procedure:

1. Create an Integration Procedure.
2. Add a Cache Block component:
 - To open the Elements panel, click +, select **Cache Block**, click •••, and select **Details**.
 - If you're using the designer on a managed package, open the Available Components panel, and drag **Cache Block** to the Structure panel.
3. Configure the Cache Block component with these settings:

Property	Value
Salesforce Platform Cache Type	Org Cache
Time To Live In Minutes	5
Cache Keys	<ul style="list-style-type: none"> • Key: contactId • Value: %ExtractContact:Contact:Id% • Key: contactName • Value: %ExtractContact:Contact:Name%
Cache Block Output	<ul style="list-style-type: none"> • Key: LoopBlock1 • Value: %LoopBlock1%

4. Add a Loop Block inside the Cache Block and configure it with these settings:

Property	Value
Loop List	names
Additional Loop Output	<ul style="list-style-type: none"> • Key: ExtractContact • Value: %ExtractContact%

5. Create a Data Mapper Extract that the Data Mapper Extract Action calls, and configure it with these settings:

Tab	Settings
Extract	A Contact extract step set to Contact Name LIKE name

Tab	Settings
Output	A mapping from <code>Contact:Id</code> to <code>Contact:Id</code> A mapping from <code>Contact:Name</code> to <code>Contact:Name</code>

To create a Data Mapper Extract, see the examples in [Omnistudio Data Mapper Extract Examples](#).

- Add a Data Mapper Extract Action inside the Loop Block, and configure it with these settings:

Property	Value
Element Name	ExtractContact
Data Mapper Interface	ExtractContactName
Data Source	names:name
Filter Value	name

- Add a Response Action below the Loop Block, and configure it with these settings:

Property	Value
Send JSON Path	CacheBlock1

- Click **Edit as JSON**. If you're using the designer on a managed package, click **Preview**, and then click **Edit as JSON**.
- Copy this JSON data and paste it into the Input Parameters panel:

```
{
  "names": [
    {
      "name": "Miller"
    },
    {
      "name": "Torres"
    }
  ]
}
```

To demonstrate performance most effectively, specify common and/or many names.

10. Click **Execute** and note the resulting Browser, Server, and Apex CPU values.
11. Open the Options pane and change the `ignoreCache` value to `false`.
12. Click **Execute**. This step populates the cache, so the resulting Browser, Server, and Apex CPU values should be similar to the previous values.
13. Click **Execute**. This step uses the cached values, so the resulting Browser, Server, and Apex CPU values should be noticeably less than the previous values.
14. To see the `vlcCacheKey` and `vlcCacheResult` nodes, open the Debug Log and scroll to the `CacheBlock1Info` node.

Handle Errors in Integration Procedures

You can configure the conditions for success or failure of an Integration Procedure action or group of actions.

In each step, you can set the following properties:

- Failure Conditional Formula: Specify a formula that evaluates to TRUE if the step has failed to execute successfully. See the second example in [Handle Errors by Using a Try-Catch Block](#).
-  **Note** An action that returns a list cannot use returned data in its Failure Conditional Formula.
- Fail On Step Error: Enable this option to terminate the Integration Procedure if the conditional formula determines that the step has failed.

To capture detailed information about failed Integration Procedure steps, configure error logging. See [Enable Error Logging for Integration Procedures](#).

To configure conditions and behavior for the success and failure of a group of actions, you can use a Try-Catch Block. See [Handle Errors by Using a Try-Catch Block](#).

To add debugging information to the Data JSON, use Response Actions. For details, see [Response Action for Integration Procedures](#).

HTTP actions add details about the results of the call to the Data JSON `ElementNameInfo` node as follows:

- Response headers, for example, `Content-Type`
- Status of the response
- Status code, for example, 200

[Enable Error Logging for Integration Procedures](#)

Enable error logging to capture detailed information about failed Integration Procedure steps.

Omnistudio writes all Integration Procedure errors to the OmniComponentErrorLog sObject records.

[Handle Errors by Using a Try-Catch Block](#)

Handle errors in Integration Procedures by using Try-Catch Blocks. A Try-Catch Block lets you "try" running the steps inside it and then "catch" the error if a step fails.

Enable Error Logging for Integration Procedures

Enable error logging to capture detailed information about failed Integration Procedure steps. Omnistudio writes all Integration Procedure errors to the `OmniComponentErrorLog` sObject records.

1. To enable error logging for Integration Procedures, create an Omni Interaction Configuration setting.
 - a. From Setup, enter Omni Interaction Configuration in the Quick Find box, and then select `Omni Interaction Configuration`.
 - b. Click **New Omni Interaction Configuration**.
 - c. For label and name, enter `ErrorLoggingEnabled` without space and without changing the case.
 - d. Enter `true` as the value.
 - e. Save your changes.

The error details and response for a failed step are sent to the Input field on the `OmniComponentErrorLog` sObject record. The error message is sent to the ErrorMessage field. You can also send custom error messages to the ErrorMessage field on the `OmniComponentErrorLog` sObject.

2. View detailed information about failed Integration Procedure steps in the `OmniComponentErrorLog` sObject records:
 - a. From Setup, find Users, and then select **Profiles**.
 - b. Select a profile.
 - c. In Tab Settings, set Omni Component Error Logs to **Default On**.
 - d. From the App Launcher, select **Omni Component Error Logs**.

Handle Errors by Using a Try-Catch Block

Handle errors in Integration Procedures by using Try-Catch Blocks. A Try-Catch Block lets you "try" running the steps inside it and then "catch" the error if a step fails.

To set up a Try-Catch Block:

1. Drag a Try-Catch Block into the Structure panel and make sure its **Fail on Block Error** checkbox is checked.
2. Configure the "catch" behavior – what the Try-Catch Block will do if a failure occurs. You can choose one or both of these options:
 - Under **Failure Response**, specify a key-value pair to return as the response. The value can be a formula. In Spring '20 and later releases, the value can include merge fields.
 - Under **Custom Failure Response**, specify a Remote Class and a Remote Method to execute an Apex class. The Apex class must implement `Callable`.
3. Drag substeps into the Try-Catch Block, and make sure the **Fail on Step Error** checkbox is checked for each step that must trigger the "catch" behavior if it fails.

You can optionally specify a **Failure Condition Formula**, which evaluates to TRUE if a specific step has failed to execute successfully.

Create a Try-Catch Block Example

An Integration Procedure creates and deletes a Contact with the specified LastName and returns an error message if the LastName is blank.

Create a Try-Catch Block Example with a Formula

An Integration Procedure finds Contacts with the specified Name and returns an error message if none are found. Because returning no records normally isn't considered a failure, the Omnistudio Data Mapper Extract Action within the Try-Catch Block includes a Failure Condition Formula.

Create a Try-Catch Block Example

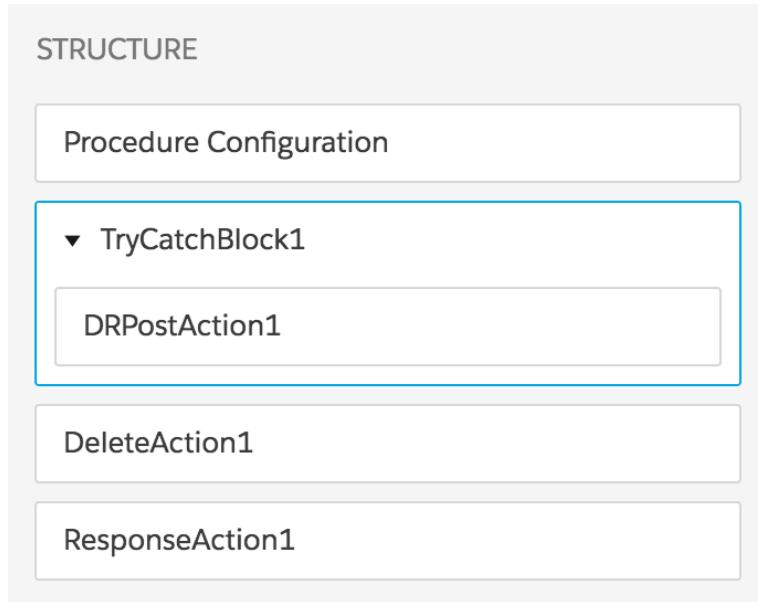
An Integration Procedure creates and deletes a Contact with the specified LastName and returns an error message if the LastName is blank.

To download a DataPack of this example for Omnistudio, click [here](#).

The Integration Procedure has these components:

- A Try-Catch Block, named TryCatchBlock1
- An Omnistudio Data Mapper Post Action, named DRPostAction1
- A Delete Action, named DeleteAction1
- A Response Action, named ResponseAction1

The Structure panel looks like this:



To build this Integration Procedure:

1. From the Omnistudio Integration Procedures tab, click **New**.
2. Provide an **Integration Procedure Name**, a **Type**, and a **SubType**, and click **Save**.
3. Drag a Try-Catch Block into the Structure panel and give it the following settings:

- a. Under Failure Response, click **Add Key/Value Pair**. Set the Key to `failureResponse` and the Value to `You must provide a last name.`.
 - b. Make sure the **Fail on Block Error** checkbox is checked.
4. Create the Data Mapper Load that the Data Mapper Post Action component calls:
- a. Give it a **Data Mapper Interface Name** of `CreateContact` and an **Interface Type of Load**.
 - b. On the Objects tab, click **Add Object** and select **Contact**.
 - c. On the Fields tab, click the + icon and enter `LastName` as both the **Input JSON Path** and the **Domain Object Field**.
- If you aren't sure how to create a Data Mapper Load, see the examples in [Omnistudio Data Mapper Load Examples](#).
5. Drag a Data Mapper Post Action component into the Try-Catch Block and give it the following settings:
- a. Set the **Element Name** to `DRPostAction1`.
 - b. Set the **Data Mapper Interface** to `CreateContact`.
 - c. Make sure the **Fail on Step Error** checkbox is checked.
6. Drag a Delete Action component after the Try-Catch Block. Under Delete SObject, set the **Type** to **Contact** and the **Path To Id** to `%DRPostAction1:Contact_1:Id%`.
7. Drag a Response Action into the Structure panel as the last component and check the **Return Full Data JSON** checkbox.
8. Go to the Preview tab and test the Integration Procedure:
- a. Under Input Parameters, click **Add New Key/Value Pair**.
 - b. Set the Key to `LastName` and the Value to any name you like.
 - c. Click **Execute**.

The output should look something like this:

```
{  
  "response": {},  
  "ResponseAction1Status": true,  
  "DeleteAction1": [  
    {  
      "errors": [],  
      "success": true,  
      "id": "0034N000001rNgqqQAC"  
    }  
  ]}
```

```
],
  "DeleteAction1Status": true,
  "TryCatchBlock1": null,
  "TryCatchBlock1Status": true,
  "DRPostAction1": {
    "ActualTime": 626,
    "CpuTime": 345,
    "Contact_1": [
      {
        "UpsertSuccess": true,
        "Id": "0034N00001rNgqqQAC",
        "LastName": "Aristotle"
      }
    ],
    "error": "OK",
    "responseType": "SObject"
  },
  "DRPostAction1Status": true,
  "options": {
    "queueableChainable": false,
    "ignoreCache": true,
    "resetCache": false,
    "chainable": false
  },
  "LastName": "Aristotle"
}
```

9. Make the Value blank and click **Execute** again. The output should look something like this:

```
{
  "result": {
    "failureResponse": "You must provide a last name."
  },
  "success": false
}
```

Create a Try-Catch Block Example with a Formula

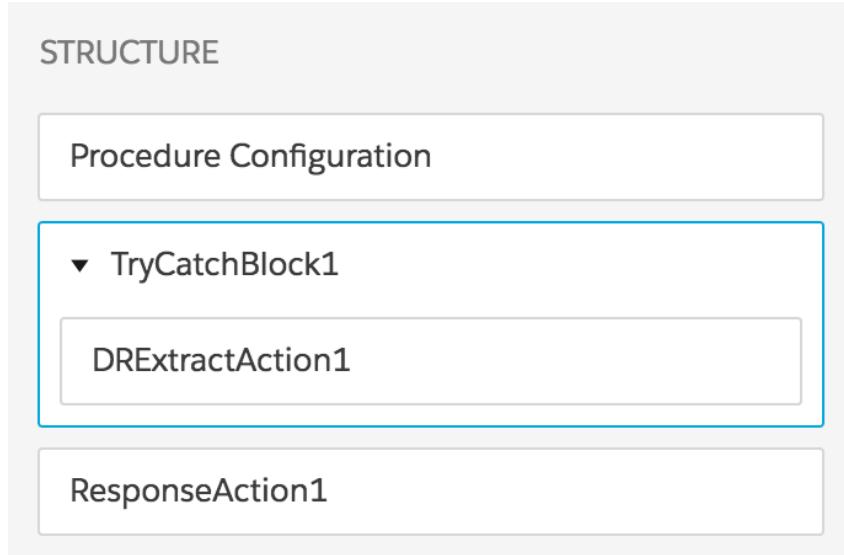
An Integration Procedure finds Contacts with the specified Name and returns an error message if none are found. Because returning no records normally isn't considered a failure, the Omnistudio Data Mapper Extract Action within the Try-Catch Block includes a Failure Condition Formula.

To download a DataPack of this example for Omnistudio, click [here](#).

The Integration Procedure has these components:

- A Try-Catch Block, named TryCatchBlock1
- A Data Mapper Extract Action, named DRExtractAction1
- A Response Action, named ResponseAction1

The Structure panel looks like this:



To build this Integration Procedure:

1. Create the Data Mapper Extract that the Integration Procedure calls:
 - a. From the Omnistudio Data Mapper tab, click **New**.
 - b. Enter a **Data Mapper Interface Name** of `GetContactName` and an **Interface Type** of **Extract**, and click **Save**.
 - c. On the Extract tab, click **Add Extract Step** and select **Contact**. Specify the path and filter `Contact Name LIKE Name`.
 - d. On the Output tab, click the + icon and enter `Contact:Name` as the **Extract JSON Path** and `ContactName` as the **Output JSON Path**.
- If you aren't sure how to create a Data Mapper Extract, see the examples in [Omnistudio Data Mapper Extract Examples](#).
2. From the Omnistudio Integration Procedures tab, click **New**.
3. Provide an **Integration Procedure Name**, a **Type**, and a **SubType**, and click **Save**.
4. Drag a Try-Catch Block into the Structure panel and give it the following settings:
 - a. Under Failure Response, click **Add Key/Value Pair**. Set the Key to `failureResponse` and the Value to `Name %Name% not found`.
 - b. Make sure the **Fail on Block Error** checkbox is checked.

5. Drag a Data Mapper Extract Action component within the Try-Catch Block and give it the following settings:
 - a. Set the **Element Name** to `DRExtractAction1`.
 - b. Set the **Data Mapper Interface** to `GetContactName`.
 - c. Make sure the **Fail on Step Error** checkbox is checked.
 - d. Specify this **Failure Condition Formula**:

```
ISBLANK(ContactName)
```

6. Drag a Response Action after the Try-Catch Block and check the **Return Full Data JSON** box.
7. Go to the Preview tab and test the Integration Procedure:

- a. Under Input Parameters, click **Add New Key/Value Pair**.
- b. Set the Key to `Name` and the Value to any first or last name you like.
- c. Click **Execute**.

If at least one Contact with that Name is found, the output looks something like this:

```
{
  "Name": "Leanne",
  "TryCatchBlock1": null,
  "DRExtractAction1Status": true,
  "TryCatchBlock1Status": true,
  "ResponseAction1Status": true,
  "options": {
    "vlcFilesMap": null,
    "forceQueueable": false,
    "mockHttpResponse": null,
    "vlcApexResponse": true,
    "ParentInteractionToken": null,
    "useFuture": false,
    "isTestProcedure": false,
    "useQueueable": false,
    "disableMetadataCache": false,
    "resetCache": false,
    "vlcIPData": null,
    "OmniAnalyticsTrackingDebug": false,
    "ignoreCache": true,
    "isDebug": true,
    "queueableChainable": false,
    "useContinuation": false,
```

```
"chainable": false,  
"shouldCommit": false,  
"vlcTestSuiteUniqueKey": null,  
"vlcTestUniqueKey": null,  
"vlcCacheKey": null,  
"useHttpCalloutMock": false,  
"continuationStepResult": null,  
"useQueueableApexRemoting": false  
},  
"DRExtractAction1": {  
    "ContactName": "Leanne Tomlin"  
},  
"response": {}  
}
```

8. Change the Value to an uncommon name (or make it blank) and click **Execute** again. The output looks something like this:

```
{  
    "success": false,  
    "result": {  
        "error": [],  
        "failureResponse": "Name Quincy not found.",  
        "success": false,  
        "result": []  
    }  
}
```

Integration Procedure Best Practices

To maximize the benefits of Integration Procedures, follow the best practices whenever possible.

- Use Integration Procedures for all data calls to Salesforce.
- To avoid SOQL limit errors, don't put too many data calls in the same Integration Procedure.
- To ensure data security and maintain compliance with Salesforce encryption access controls, always check that a user has the **View Encrypted Data** permission before displaying or processing decrypted values of encrypted fields.
- To maximize reuse and performance, use a modular approach to implement a small group of related functions in each Integration Procedure.
- Use a [Response Action for Integration Procedures](#) to trim the data and only return what is needed. For specific trimming strategies, see [Manipulate JSON with the Send/Response Transformations Properties](#).
- To allow an Integration Procedure to exit early under appropriate conditions, use multiple Response Actions with different Execution Conditional Formulas.

- Use caching to store frequently accessed, infrequently updated data. See [Improving Data Mapper Performance with Caching](#).
- Avoid using an empty Loop Block in an Integration Procedure without any action elements inside it. Loop Blocks iterate over a data array and execute defined actions for each item. Without any action elements inside, the Loop Block performs no meaningful work and may indicate a design flaw. Additionally, unnecessary loops can negatively impact performance and maintainability of your Integration Procedures. To ensure efficiency and clarity, always include relevant action elements within the Loop Block.
- To run data operations asynchronously, call Integration Procedures using these settings:
 - Use Future – Use when the calling Omniscript or Integration Procedure doesn't need a response and completion time isn't critical.
 - Invoke Mode: Fire and Forget – Use Invoke Mode instead of Use Future when the calling Omniscript must invoke the Integration Procedure immediately.
 - Invoke Mode: Non-Blocking – Use to run the Integration Procedure immediately while continuing the user interaction of the calling Omniscript. A response is returned when the Integration Procedure is complete.



Note When an Omniscript invokes an Integration Procedure asynchronously, the Integration Procedure continues running if the user goes to the next step, but not if the user navigates away from the Omniscript.

For more information about these settings, see [Common Action Element Properties](#), [Retrieve Data with an Integration Procedure in an Omniscript](#), and [Integration Procedure Action for Integration Procedures](#).

To determine whether an Omnistudio Data Mapper or an Integration Procedure is best for your use case, see [When to Use Omnistudio Data Mappers and Integration Procedures](#).

Integration Procedures on Agentforce

Integration Procedures in Omnistudio are powerful tools used to automate data integration and transfer between systems. They enable the creation of complex workflows, data transformations, and business logic execution, all within a single, unified environment. Agentforce is Salesforce's agent-focused workspace, designed to streamline customer service operations by providing a centralized platform for case management, customer interaction, and system integration.

With the new Agentforce configuration for Integration Procedures, you can now bring these automation capabilities directly into the Agentforce environment. This integration simplifies workflows, automates data movement, and enhances efficiency—allowing agents to access real-time data and execute backend processes without leaving their workspace.

Industries that manage a high volume of cases, such as healthcare, insurance, and utilities can benefit from improved case management through features like smart routing, instant data access, and automated routine tasks. These enhancements lead to greater agent productivity, faster issue resolution, and a better customer experience.

Enabling Integration Procedures within Agentforce enhances agent productivity and simplifies service workflows by bringing powerful automation tools directly into the agent workspace.

Integration Procedures that are configured with Agentforce offer these benefits:

- Reusability: Leverage existing Integration Procedures without additional development.
- Real-time data access: Retrieve information instantly from multiple systems to support faster service delivery.
- Process automation: Automate routine tasks and integrate with flows, decisioning, and AI tools.
- Streamlined workflows: Centralize logic and data interactions in one environment to reduce system complexity.

Points to consider while working with Integration Procedures that are configured with Agentforce:

- Execution of user-created vs. Standard Integration Procedure - When you create an Integration Procedure version from a standard Integration Procedure version, Agentforce prioritizes the user-created Integration Procedure during execution. This means that even if both the standard and user-created Integration Procedure versions exist and are active, Agentforce always invokes the user-created Integration Procedure.

For example: When you create a user-created Integration Procedure (V2) from an existing standard version (V1) and activate V2 while V1 remains active, V2 is invoked at runtime. This occurs even though both versions are active, as the agent runtime always gives precedence to the user-created version.

- Behavior on Salesforce Flow: On Flow, both the user-created and standard versions of an Integration Procedure are listed as available actions. However, regardless of which version you choose, Flow always runs the user-created version because it has higher priority.

For example: Both `StandardAction` and `UserCreatedAction` versions of an Integration Procedure appear as available in Flow. Even if you select `StandardAction` while building a flow, the `UserCreatedAction` version is invoked at runtime, as it takes precedence.

- Apex class changes on Flow: When you update the input or output configuration of an Integration Procedure, the related Apex classes are automatically updated.

 **Note** On Flow, these changes might not take effect instantly when using Integration Procedure as an invocable action, because Flow may cache older versions.

Integrate Integration Procedures with Agentforce

To make Integration Procedures available as a new action type within Agent Actions, start by selecting the desired Integration Procedure. Define its input and output data in JSON format, then verify the data. This process automatically converts the Integration Procedure Data JSON into a format compatible with Agentforce. Then, create an Agent Action, configure the Agent and Topic, and activate the embedded Integration Procedure within Agentforce. After it's set up, the Integration Procedure can be triggered during agent-customer interactions to drive real-time automation and data access.

Integrate Integration Procedures with Agentforce

To make Integration Procedures available as a new action type within Agent Actions, start by selecting

the desired Integration Procedure. Define its input and output data in JSON format, then verify the data. This process automatically converts the Integration Procedure Data JSON into a format compatible with Agentforce. Then, create an Agent Action, configure the Agent and Topic, and activate the embedded Integration Procedure within Agentforce. After it's set up, the Integration Procedure can be triggered during agent-customer interactions to drive real-time automation and data access.

Before you begin:

- Ensure that you've the Omnistudio permissions. See [Omnistudio Permission Sets](#).
- Enable the Omnistudio Metadata in your org. See [Enable Omnistudio Metadata API Support](#).

Prepare an Integration Procedure for Agentforce Integration

Before using an Integration Procedure within Agentforce, it's important to note that the data format expected by Agentforce is different from the standard Integration Procedure format. To ensure compatibility, you must prepare the Integration Procedure accordingly.

1. From the Integration Procedure list view page, select and open the Integration Procedure that you want to integrate with Agentforce.
2. Click **Preview**.
3. Click **Configure Agentforce**.
4. Define the input and output data for the Integration Procedure in JSON format, and then click **Verify Data**.

 **Note** When defining the input and output JSON, only alphanumeric characters (A-Z, a-z, 0-9) are allowed for the key in the key-value pair. No special characters are permitted.

For example, enter this data that includes a user's income details in JSON format to calculate the base tax slab and surcharge based on the income slab.

Input Data JSON:

```
{  
    "taxpayerId": "TAXP123456",  
    "financialYear": "2024-2025",  
    "totalIncome": 1800000  
}
```

Output JSON Response:

```
{  
    "taxpayerId": "TAXP123456",  
    "financialYear": "2024-2025",  
    "baseTax": 315000,  
    "surcharge": 0,  
    "totalTaxPayable": 315000
```

{}

When you click **Verify Data**, the data structure is updated automatically.

5. Click **Next**.
6. Click **Save**.
7. Activate the Integration Procedure.

Create Agent Actions

Agent Actions act as connectors between Agentforce and the Integration Procedure.

1. From Setup, search and open Agentforce Assets, click **Actions**, and then click **New Agent Action**.



Note Dynamic Apex classes may be auto-generated even before configuring Integration Procedures as Agent Actions. These Apex classes have no functional impact and can be safely ignored.

2. Select **Integration Procedure** as the reference action type.
3. Select the desired Integration Procedure as the reference action.
4. Enter an Agent action label.
5. Enter an Action API name.
6. Click **Next**.
7. Provide input instructions:
 - a. Select **Require input**.
 - b. Select **Collect data from user**.
8. Provide output instructions:
 - a. Select **Select Show in conversation**.
9. Click **Finish**.

Configure an Agent

Create an Agent to manage conversation flows and execute Agent Actions. Add a topic and associate it with the Agent Action. Topics define specific intents or tasks the Agent handles.

For information about how to configure an Agent, see [Configure Your Agent](#).

Invoke an Integration Procedure

You can invoke Integration Procedures from other Omnistudio tools such as Omniscripts and Cards. You can also invoke Integration Procedures from Apex classes, batch jobs, REST APIs, or Salesforce flows.

[Integration Procedure Invocation from Apex](#)

Starting with Summer' 25, use the

`ConnectAPI.OmniDesignerConnect.integrationProcedureExecute(ipName, apexInput)` Connect API for Integration Procedure calls from the `IntegrationProcedureService` Apex class. This Connect API removes the dependency on the managed package and provides up to 21% better performance for Integration Procedure calls from an Apex class compared to the previous method.

Integration Procedure Unit Testing from Apex

You can set up a test class in Apex and use it to call and test an Integration Procedure. In the core runtime, HTTP mocking isn't possible, and real HTTP calls occur during test execution.

Integration Procedure Invocation from REST APIs

With a REST API, you can use either a GET or a POST call to invoke an Integration Procedure and retrieve the result. The difference between the two is that a GET call can't pass data with a JSON request body.

Integration Procedure Invocation from Salesforce Flow

To call an Integration Procedure from a Salesforce Flow, create a Salesforce Flow component by defining a class using the Salesforce Developer Console, as shown in the following example. Note that the class must have the `@InvocableMethod` annotation.

Batch Jobs for Integration Procedures and Vlocity Open Interfaces

The Vlocity batch framework lets you run an Integration Procedure or `VlocityOpenInterface` on a schedule. Typical use cases are a recurrent billing cycle or a scan for insurance policies that come up for renewal in the next 60 days. When a job is run, it sends its input to an Integration Procedure or Apex method that processes the records.

Integration Procedure Invocation from Apex

Starting with Summer' 25, use the `ConnectAPI.OmniDesignerConnect.integrationProcedureExecute(ipName, apexInput)` Connect API for Integration Procedure calls from the `IntegrationProcedureService` Apex class. This Connect API removes the dependency on the managed package and provides up to 21% better performance for Integration Procedure calls from an Apex class compared to the previous method.

 **Important** The performance figures in this document are provided for informational purposes only and are based on internal validation and testing under specific conditions. Actual results may vary depending on your component design, production environment, and other factors. These figures provide general guidance and aren't a guarantee of performance.

Invoke an Integration Procedure and return its data

Invokes an Integration Procedure and returns its data. Ignores Sharing Rules and Custom Permissions, which ensures that the Integration Procedure is private. See Sharing Rules and Custom Permissions in Salesforce Help.

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre> String ipName = 'FindContacts_LastNames'; Map<String, Object> inputMap = new Map<String, Object>(); inputMap.put('ContactLastName', 'Smith'); Map<String, Object> optionsMap = new Map<String, Object>(); optionsMap.put('isDebug', false); ConnectAPI.IntegrationProcedureServiceRunInputRepresentation apexInput = new ConnectAPI.IntegrationProcedureServiceRunInputRepresentation(); List<String> stringList = new List<String>(); String serializedInputMap = JSON.serialize(inputMap); stringList.add(serializedInputMap); apexInput.input = stringList; ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation options = new ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation(); for (String key: optionsMap.keySet()) { if (key == 'isDebug') { options.isDebugEnabled = (Boolean)optionsMap.get(key); } else if (key == 'chainable') { options.isChainable = (Boolean)optionsMap.get(key); } else if (key == 'resetCache') { options.resetCache = (Boolean)optionsMap.get(key); } else if (key == 'ignoreCache') { } } </pre>	<pre> String ipName = 'FindContacts_LastNames'; Map<String, Object> inputMap = new Map<String, Object>(); inputMap.put('ContactLastName', 'Smith'); Map<String, Object> optionsMap = new Map<String, Object>(); optionsMap.put('isDebug', false); Object ipOutput = devopsimpkg11.IntegrationProcedureService.runIntegrationService(ipName, inputMap, optionsMap); </pre>

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre> options.ignoreCache = (Boolean)optionsMap.get(key); } else if (key == 'queueableChai nable') { options.queueableChainable = (Boolean)optionsMap.get(key); } else if (key == 'useQueueableA pexRemoting') { options.useQueueableApexRemo ting = (Boolean)optionsMap.get(key); } else if (key == 'vlcApexRespon se') { options.vlcApexResponse = (B oolean)optionsMap.get(key); } else if (key == 'useFuture') { options.useFuture = (Boolea n)optionsMap.get(key); } else if (key == 'useQueueabl e') { options.useQueueable = (Bool ean)optionsMap.get(key); } else if (key == 'vlcIPData') { options.vlcIPData = (Strin g)optionsMap.get(key); } else if (key == 'vlcStatus') { options.vlcStatus = (Strin g)optionsMap.get(key); } else if (key == 'vlcMessage') { options.vlcMessage = (Strin g)optionsMap.get(key); } } options.shouldSendLegacyResponse = tr ue; apexInput.options = options; ConnectAPI.IntegrationProcedureServiceRun OutputRepresentation output = ConnectAP I.OmniDesignerConnect.integrationProcedur </pre>	

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre data-bbox="204 285 791 760"> eExecute(ipName, apexInput); List<String> responseList = output.response; String currentSerResponse = responseList[0]; Map<String, Object> currentResponseObj = (Map<String, Object>) JSON.deserializeUntyped(currentSerResponse); Object ipOutput = currentResponseObj.get('result'); </pre>	

Invoke an Integration Procedure and return its data ignoring Sharing Rules and Custom Permissions

Invokes an Integration Procedure and returns its data. Ignores sharing rules and custom permissions, which ensures that the Integration Procedure is private. See [Sharing Rules](#) and [Custom Permissions](#) in Salesforce Help.

Connect API for IP Invocation	Existing Method (Replace with Connect API)
<p>Users must have appropriate permissions to execute the Integration Procedure.</p> <pre data-bbox="204 1436 791 1826"> String ipName = 'FindContacts_LastNames'; Map<String, Object> inputMap = new Map<String, Object>(); inputMap.put('ContactLastName', 'Smith'); Map<String, Object> optionsMap = new Map<String, Object>(); optionsMap.put('isDebug', false); </pre>	<pre data-bbox="840 1288 1428 1858"> String ipName = 'FindContacts_LastNames'; Map<String, Object> inputMap = new Map<String, Object>(); inputMap.put('ContactLastName', 'Smith'); Map<String, Object> optionsMap = new Map<String, Object>(); optionsMap.put('isDebug', false); Object ipOutput = devopsim pkg11.IntegrationProcedureService.runIntegrationProcedureServiceFromApex(ipName, inputMap, optionsMap); </pre>

Connect API for IP Invocation	Existing Method (Replace with Connect API)
<pre> ConnectAPI.IntegrationProcedureServiceRunInputRepresentation apexInput = new ConnectAPI.IntegrationProceduresServiceRunInputRepresentation(); List<String> stringList = new List<String>(); String serializedInputMap = JSON.serialize(inputMap); stringList.add(serializedInputMap); apexInput.input = stringList; ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation options = new ConnectAPI.IntegrationProceduresServiceRunOptionsRepresentation(); for (String key: optionsMap.keySet()) { if (key == 'isDebug') { options.isDebugEnabled = (Boolean)optionsMap.get(key); } else if (key == 'chainable') { options.isChainable = (Boolean)optionsMap.get(key); } else if (key == 'resetCache') { options.resetCache = (Boolean)optionsMap.get(key); } else if (key == 'ignoreCache') { options.ignoreCache = (Boolean)optionsMap.get(key); } else if (key == 'queueableChainable') { options.queueableChainable = (Boolean)optionsMap.get(key); } else if (key == 'useQueueableApexRemoting') { options.useQueueableApexRemoting = (Boolean)optionsMap.get(key); } else if (key == 'vlcApexResponse') { </pre>	

Connect API for IP Invocation	Existing Method (Replace with Connect API)
<pre> options.vlcApexResponse = (B oolean)optionsMap.get(key); } else if (key == 'useFuture') { options.useFuture = (Boolea n)optionsMap.get(key); } else if (key == 'useQueueabl e') { options.useQueueable = (Bool ean)optionsMap.get(key); } else if (key == 'vlcIPData') { options.vlcIPData = (Strin g)optionsMap.get(key); } else if (key == 'vlcStatus') { options.vlcStatus = (Strin g)optionsMap.get(key); } else if (key == 'vlcMessage') { options.vlcMessage = (Strin g)optionsMap.get(key); } } options.shouldSendLegacyResponse = tr ue; apexInput.options = options; ConnectAPI.IntegrationProcedureRu nOutputRepresentation output = ConnectAP I.OmniDesignerConnect.integrationProcedur eExecute(ipName, apexInput); List<String> responseList = output.r esponse; String currentSerResponse = response List[0]; Map<String, Object> currentResponseO bj = (Map<String, Object>) JSON.deser ializeUntyped(currentSerResponse); Object ipOutput = currentResponseObj.ge t('result'); </pre>	

Invoke an Integration Procedure with a @future annotation applied

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre> String ipName = 'FindContacts_LastNames'; String inputMap = '{"ContactLastName": "Smith"}'; String optionsMap = '{"isDebug": false}'; ConnectAPI.IntegrationProcedureServiceRunInputRepresentation apexInput = new ConnectAPI.IntegrationProcedureServiceRunInputRepresentation(); List<String> stringList = new List<String>(); stringList.add(inputMap); apexInput.input = stringList; Map<String, Object> optionsMapDeserialized = (Map<String, Object>) JSON.deserializeUntyped(optionsMap); ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation options = new ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation(); for (String key: optionsMapDeserialized.keySet()) { if (key == 'isDebug') { options.isDebugEnabled = (Boolean) optionsMapDeserialized.get(key); } else if (key == 'chainable') { options.isChainable = (Boolean) optionsMapDeserialized.get(key); } else if (key == 'resetCache') { options.resetCache = (Boolean) optionsMapDeserialized.get(key); } } </pre>	<pre> String ipName = 'FindContacts_LastNames'; String inputMap = '{"ContactLastName": "Smith"}'; String optionsMap = '{"isDebug": false}'; devopsimpkg11.IntegrationProcedureService.runIntegrationServiceFuture(ipName, inputMap, optionsMap); </pre>

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre> } else if (key == 'ignoreCache') { options.ignoreCache = (Boolean)optionsMapDeserialized.get(key); } else if (key == 'queueableChai nable') { options.queueableChainable = (Boolean)optionsMapDeserialized.ge t(key); } else if (key == 'useQueueableA pexRemoting') { options.useQueueableApexRemo ting = (Boolean)optionsMapDeserializ ed.get(key); } else if (key == 'vlcApexRespon se') { options.vlcApexResponse = (B oolean)optionsMapDeserialized.get(ke y); } else if (key == 'useFuture') { options.useFuture = (Boolea n)optionsMapDeserialized.get(key); } else if (key == 'useQueueabl e') { options.useQueueable = (Bool ean)optionsMapDeserialized.get(key); } else if (key == 'vlcIPData') { options.vlcIPData = (Strin g)optionsMapDeserialized.get(key); } else if (key == 'vlcStatus') { options.vlcStatus = (Strin g)optionsMapDeserialized.get(key); } else if (key == 'vlcMessage') { options.vlcMessage = (Strin g)optionsMapDeserialized.get(key); } } options.shouldSendLegacyResponse = tr </pre>	

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre> ue; options.useFuture = true; apexInput.options = options; ConnectAPI.OmniDesignerConnect.integrati onProcedureExecute(ipName, apexInput); </pre>	

Invoke an Integration Procedure as a Queueable Apex Job for Asynchronous Processing

Returns the ID of the Queueable job. See [Queueable Apex](#) in Salesforce Help.

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre> String ipName = 'FindContacts_LastNa mes'; Map<String, Object> inputMap = new M ap<String, Object>(); inputMap.put('ContactLastName', 'Smi th'); Map<String, Object> optionsMap = new Map<String, Object>(); optionsMap.put('isDebug', false); ConnectAPI.IntegrationProcedureServ ceRunInputRepresentation apexInput = new ConnectAPI.IntegrationProcedureS erviceRunInputRepresentation(); List<String> stringList = new List<S tring>(); String serializedInputMap = JSON.serialize(in putMap); stringList.add(serializedInputMap); apexInput.input = stringList; </pre>	<pre> String ipName = 'FindContacts_LastNa mes'; Map inputMap = new Map(); inputMap.put('ContactLastName', 'Smi th'); Map optionsMap = new Map(); optionsMap.put('isDebug', false); Id ipOutput = devopsimpkg11.Integrati onProcedureService.runIntegrationPro cedureQueueable(ipName, inputMap, op tionsMap); </pre>

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre>ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation options = new ConnectAPI.IntegrationProcedureserviceRunOptionsRepresentation(); for (String key: optionsMap.keySet()) { if (key == 'isDebug') { options.isDebug = (Boolean)optionsMap.get(key); } else if (key == 'chainable') { options.chainable = (Boolean)optionsMap.get(key); } else if (key == 'resetCache') { options.resetCache = (Boolean)optionsMap.get(key); } else if (key == 'ignoreCache') { options.ignoreCache = (Boolean)optionsMap.get(key); } else if (key == 'queueableChainable') { options.queueableChainable = (Boolean)optionsMap.get(key); } else if (key == 'useQueueableApexRemoting') { options.useQueueableApexRemoting = (Boolean)optionsMap.get(key); } else if (key == 'vlcApexResponse') { options.vlcApexResponse = (Boolean)optionsMap.get(key); } else if (key == 'useFuture') { options.useFuture = (Boolean)optionsMap.get(key); } else if (key == 'useQueueable') { options.useQueueable = (Boolean)optionsMap.get(key); } }</pre>	

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre> } else if (key == 'vlcIPData') { options.vlcIPData = (String)optionsMap.get(key); } else if (key == 'vlcStatus') { options.vlcStatus = (String)optionsMap.get(key); } else if (key == 'vlcMessage') { options.vlcMessage = (String)optionsMap.get(key); } options.shouldSendLegacyResponse = true; options.useQueueable = true; apexInput.options = options; ConnectAPI.IntegrationProcedureServiceRun OutputRepresentation output = ConnectAP I.OmniDesignerConnect.integrationProcedur eExecute(ipName, apexInput); List<String> responseList = output.r esponse; String currentSerResponse = response List[0]; Map<String, Object> currentResponseO bj = (Map<String, Object>) JSON.deser ializeUntyped(currentSerResponse); Object currentResult = currentRespon seObj.get('result'); Map<String, Object> mapCurrentResult = (Map<String, Object>)currentResul t; Object ipResultObj = mapCurrentResul t.get('IPResult'); Map<String, Object> IpResultMap = (M ap<String, Object>)ipResultObj; </pre>	

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre data-bbox="204 285 791 359">Id ipOutput = (Id)IpResultMap.get('queueableJobId');</pre>	

Invokes an Integration Procedure passing in a map of input parameters and receiving the output in another map

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre data-bbox="204 781 791 1795"> String ipName = 'FindContacts_LastNames'; Map<String, Object> inputMap = new Map<String, Object>(); inputMap.put('ContactLastName', 'Smith'); Map<String, Object> outputMap = new Map<String, Object>(); Map<String, Object> optionsMap = new Map<String, Object>(); optionsMap.put('isDebug', false); ConnectAPI.IntegrationProcedureServiceRunInputRepresentation apexInput = new ConnectAPI.IntegrationProcedureServiceRunInputRepresentation(); List<String> stringList = new List<String>(); String serializedInputMap = JSON.serialize(inputMap); stringList.add(serializedInputMap); apexInput.input = stringList; ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation options = </pre>	<pre data-bbox="848 781 1436 1752"> String ipName = 'FindContacts_LastNames'; Map<String, Object> inputMap = new Map<String, Object>(); inputMap.put('ContactLastName', 'Smith'); Map<String, Object> outputMap = new Map<String, Object>(); Map<String, Object> optionsMap = new Map<String, Object>(); optionsMap.put('isDebug', false); devopsimpkg11.IntegrationProcedureService ipService = new devopsimpkg11.IntegrationProcedureService(); ipService.invokeMethod(ipName, inputMap, outputMap, optionsMap); // Returns Boolean Object ipOutput = outputMap.get('IPResult'); </pre>

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre>new ConnectAPI.IntegrationProcedures serviceRunOptionsRepresentation(); for (String key: optionsMap.keySet()) { if (key == 'isDebug') { options.isDebug = (Boolean)optionsMap.get(key); } else if (key == 'chainable') { options.chainable = (Boolean)optionsMap.get(key); } else if (key == 'resetCache') { options.resetCache = (Boolean)optionsMap.get(key); } else if (key == 'ignoreCache') { options.ignoreCache = (Boolean)optionsMap.get(key); } else if (key == 'queueableChai nable') { options.queueableChainable = (Boolean)optionsMap.get(key); } else if (key == 'useQueueableA pexRemoting') { options.useQueueableApexRemo ting = (Boolean)optionsMap.get(key); } else if (key == 'vlcApexRespon se') { options.vlcApexResponse = (B oolean)optionsMap.get(key); } else if (key == 'useFuture') { options.useFuture = (Boolea n)optionsMap.get(key); } else if (key == 'useQueueabl e') { options.useQueueable = (Bool ean)optionsMap.get(key); } else if (key == 'vlcIPData') { options.vlcIPData = (Strin g)optionsMap.get(key);</pre>	

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre> } else if (key == 'vlcStatus') { options.vlcStatus = (String)optionsMap.get(key); } else if (key == 'vlcMessage') { options.vlcMessage = (String)optionsMap.get(key); } } options.shouldSendLegacyResponse = true; apexInput.options = options; ConnectAPI.IntegrationProcedureServiceRun OutputRepresentation output = ConnectAP I.OmniDesignerConnect.integrationProcedur eExecute(ipName, apexInput); List<String> responseList = output.r esponse; String currentSerResponse = response List[0]; Map<String, Object> currentResponseO bj = (Map<String, Object>) JSON.deser ializeUntyped(currentSerResponse); Object ipOutput = currentResponseOb j.get('result'); if (options.useQueueable == true) { Map<String, Object> mapCurrentRe sult = (Map<String, Object>) ipOutpu t; Object ipResultObj = mapCurrentR esult.get('IPResult'); ipOutput = ipResultObj; } else if (options.useFuture == true) {</pre>	

Connect API for Integration Procedure Invocation	Existing Method (Replace with Connect API)
<pre>ipOutput = new Map<String, Object>(); } outputMap.put('IPResult', ipOutput);</pre>	

Integration Procedure Call Examples

You can invoke an Integration Procedure from Apex code in two ways using the `IntegrationProcedureService` class.

Example 1

```
class IntegrationProcedureService {
/*
procedureAPIName - the Type_SubType of Procedure
input - The payload / initial data in the Map<String, Object> format for the Procedure
*/
String ipName = 'FindContacts_LastNames';
Map<String, Object> inputMap = new Map<String, Object>();
inputMap.put('ContactLastName', 'Smith');

Map<String, Object> optionsMap = new Map<String, Object>();
optionsMap.put('isDebug', false);

ConnectAPI.IntegrationProcedureServiceRunInputRepresentation apexInput = new ConnectAPI.IntegrationProcedureServiceRunInputRepresentation();
List<String> stringList = new List<String>();
String serializedInputMap = JSON.serialize(inputMap);
stringList.add(serializedInputMap);
apexInput.input = stringList;

ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation options = new ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation();
for (String key: optionsMap.keySet()){
if (key == 'isDebug') {
```

```

        options.isDebug = (Boolean)optionsMap.get(key);
    } else if (key == 'chainable') {
        options.chainable = (Boolean)optionsMap.get(key);
    } else if (key == 'resetCache') {
        options.resetCache = (Boolean)optionsMap.get(key);
    } else if (key == 'ignoreCache') {
        options.ignoreCache = (Boolean)optionsMap.get(key);
    } else if (key == 'queueableChainable') {
        options.queueableChainable = (Boolean)optionsMap.get(key);
    } else if (key == 'useQueueableApexRemoting') {
        options.useQueueableApexRemoting = (Boolean)optionsMap.get(key);
    } else if (key == 'vlcApexResponse') {
        options.vlcApexResponse = (Boolean)optionsMap.get(key);
    } else if (key == 'useFuture') {
        options.useFuture = (Boolean)optionsMap.get(key);
    } else if (key == 'useQueueable') {
        options.useQueueable = (Boolean)optionsMap.get(key);
    } else if (key == 'vlcIPData') {
        options.vlcIPData = (String)optionsMap.get(key);
    } else if (key == 'vlcStatus') {
        options.vlcStatus = (String)optionsMap.get(key);
    } else if (key == 'vlcMessage') {
        options.vlcMessage = (String)optionsMap.get(key);
    }
}
options.shouldSendLegacyResponse = true;
apexInput.options = options;

ConnectAPI.IntegrationProcedureServiceRunOutputRepresentation output = Connect
API.OmniDesignerConnect.integrationProcedureExecute(ipName, apexInput);

List<String> responseList = output.response;
String currentSerResponse = responseList[0];
Map<String, Object> currentResponseObj = (Map<String, Object>) JSON.deserialize
Untyped(currentSerResponse);

Object ipOutput = currentResponseObj.get('result');

```

Example 2

```

/* Initialize variables */
String procedureName = 'Type_SubType';
Map<String, Object> ipInput = new Map<String, Object> ();
Map<String, Object> ipOutput = new Map<String, Object> ();

```

```
Map<String, Object> ipOptions = new Map<String, Object> () ;

/* Populating input map for an Integration Procedure.
Follow whatever structure your VIP expects */
String orderId = '8010000000abcd';
ipInput.put('orderId', orderId);

/* Call the Integration Procedure via runIntegrationService,
and save the output to ipOutput */
ipOutput = (Map<String, Object>)

String ipName = 'FindContacts_LastNames';
Map<String, Object> inputMap = new Map<String, Object>();
inputMap.put('ContactLastName', 'Smith');

Map<String, Object> optionsMap = new Map<String, Object>();
optionsMap.put('isDebug', false);

ConnectAPI.IntegrationProcedureServiceRunInputRepresentation apexInput = new ConnectAPI.IntegrationProcedureServiceRunInputRepresentation();
List<String> stringList = new List<String>();
String serializedInputMap = JSON.serialize(inputMap);
stringList.add(serializedInputMap);
apexInput.input = stringList;

ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation options = new ConnectAPI.IntegrationProcedureServiceRunOptionsRepresentation();
for (String key: optionsMap.keySet()){
    if (key == 'isDebug') {
        options.isDebug = (Boolean)optionsMap.get(key);
    } else if (key == 'chainable') {
        options.chainable = (Boolean)optionsMap.get(key);
    } else if (key == 'resetCache') {
        options.resetCache = (Boolean)optionsMap.get(key);
    } else if (key == 'ignoreCache') {
        options.ignoreCache = (Boolean)optionsMap.get(key);
    } else if (key == 'queueableChainable') {
        options.queueableChainable = (Boolean)optionsMap.get(key);
    } else if (key == 'useQueueableApexRemoting') {
        options.useQueueableApexRemoting = (Boolean)optionsMap.get(key);
    } else if (key == 'vlcApexResponse') {
        options.vlcApexResponse = (Boolean)optionsMap.get(key);
    } else if (key == 'useFuture') {
        options.useFuture = (Boolean)optionsMap.get(key);
```

```

} else if (key == 'useQueueable') {
    options.useQueueable = (Boolean)optionsMap.get(key);
} else if (key == 'vlcIPData') {
    options.vlcIPData = (String)optionsMap.get(key);
} else if (key == 'vlcStatus') {
    options.vlcStatus = (String)optionsMap.get(key);
} else if (key == 'vlcMessage') {
    options.vlcMessage = (String)optionsMap.get(key);
}
}

options.shouldSendLegacyResponse = true;
apexInput.options = options;

ConnectAPI.IntegrationProcedureServiceRunOutputRepresentation output = ConnectAPI.OmniD
esignerConnect.integrationProcedureExecute(ipName, apexInput);

List<String> responseList = output.response;
String currentSerResponse = responseList[0];
Map<String, Object> currentResponseObj = (Map<String, Object>) JSON.deserialize
Untyped(currentSerResponse);

Object ipOutput = currentResponseObj.get('result');

System.debug('IP Output: ' + ipOutput);

```

Integration Procedure Unit Testing from Apex

You can set up a test class in Apex and use it to call and test an Integration Procedure. In the core runtime, HTTP mocking isn't possible, and real HTTP calls occur during test execution.

The following is an example unit test of an Integration Procedure:

```

@isTest(seeAllData=true)
global with sharing class TestVlocityIntergationProcedure
{
    static testMethod void testVip()
    {
        Map<String, Object> inputMap = new Map<String, Object>();
        inputMap.put('AccountName', 'Vlocity');

        ConnectAPI.IntegrationProcedureServiceRunInputRepresentation apexInput
        = new ConnectAPI.IntegrationProcedureServiceRunInputRepresentation();

        ConnectAPI.IntegrationProcedureServiceRunOutputRepresentation output =

```

```

ConnectAPI.OmniDesignerConnect.integrationProcedureExecute('Type_Subtype', apexInput);
    List<String> stringList = new List<String>();
    String serializedInputMap = JSON.serialize(inputMap);
    stringList.add(serializedInputMap);
    apexInput.input = stringList;

    ConnectAPI.IntegrationProcedureServiceRunOutputRepresentation output =
ConnectAPI.OmniDesignerConnect.integrationProcedureExecute('Type_Subtype', apexInput);
    List<String> responseList = output.response;
    String currentSerResponse = responseList[0];
    Map<String, Object> currentResponseObj = (Map<String, Object>) JSON.deserializeUntyped(currentSerResponse);

    Object response = currentResponseObj.get('result');
    System.assertEquals('joe@vlocity.com', response.get('ContactEmail'));
}
}
}

```

Note these features of the example:

- You must use Salesforce's **seeAllData** property on the `@isTest` annotation. This ensures that the unit test sees the Vlocity SObject data in the org.
- Use the `ConnectAPI.OmniDesignerConnect.integrationProcedureExecute()` Connect API to run the Integration Procedure.
- The namespace is `vlocity_cmt`, `vlocity_ins`, or `vlocity_ps`.
- The Type_Subtype parameter specifies the Integration Procedure to test.

See Also

[Integration Procedure Invocation from Apex](#)

Integration Procedure Invocation from REST APIs

With a REST API, you can use either a GET or a POST call to invoke an Integration Procedure and retrieve the result. The difference between the two is that a GET call can't pass data with a JSON request body.

You invoke a REST call with a URL formatted like this:

```
/services/apexrest/{namespace}/v1/integrationprocedure/{Type}_{SubType}/
```

The namespace is typically `omnistudio`. Specify the Type and SubType of the Integration Procedure.

You use parameters to pass data to an Integration Procedure in three ways:

- As inline values in the URL path (GET or POST)
- As query parameter=value pairs append to the URL (GET or POST)
- By specifying a JSON request body (POST only)

 **Note** When using an Integration Procedure with a REST API, follow these guidelines:

- In JSON notation, curly braces are literal. They delimit JSON objects.
- In REST URL notation, curly braces surround variable values that you must replace. Don't include the braces in an actual request.
- Treat all parts of the REST URL as case-sensitive.

You can also set Integration Procedure options such as `chainable` or `queueableChainable` as query parameters with a REST call:

```
/services/apexrest/vvelocity_ins/v1/integrationprocedure/Create_Cases/?queueableChainable=true
```

For an example of a REST call to an Integration Procedure, see [Invoke a Chainable Integration Procedure with REST Calls](#).

Integration Procedure Invocation Using GET

To request and receive JSON data results from an Integration Procedure, issue a GET call. With a GET call, you can specify inline URL path parameters and append query parameters to the URL. If you must pass data with a JSON request body, use a POST call instead.

Integration Procedure Invocation Using POST

To send JSON data to and receive results from an Integration Procedure, issue a POST call. With a POST call, you can specify inline URL path parameters, append query parameters to the URL, and include a JSON request body.

Integration Procedure Invocation Using GET

To request and receive JSON data results from an Integration Procedure, issue a GET call. With a GET call, you can specify inline URL path parameters and append query parameters to the URL. If you must pass data with a JSON request body, use a POST call instead.

In this example, an Integration Procedure that creates cases requires a Contact name and returns the Id of the newly created case. The Contact name is appended to the URL as a query parameter. The value `%20` is the URL-encoded space between the first and last names.

- GET URL:

```
/services/apexrest/vvelocity_ins/v1/integrationprocedure/Create_Cases/?Contact=Dennis%20Reynolds
```

- Result:

```
{  
  "Case": {  
    "Id": "0036100001HDn3QAAT"  
  }  
}
```

In this second example, the syntax of the URL looks like this:

```
/services/apexrest/{namespace}/v1/integrationprocedure/{Type}_{SubType}/{inlinevalue1}/{inlinevalue2}/?{Param1}={Value1}
```

You can call the URL with these values:

```
/services/apexrest/vlocity_cmt/v1/integrationprocedure/IP_Rest/Apple/Phones/?product=iPhoneX
```

The GET call sends this input to the Integration Procedure. Values passed in the URL path are added under the `options` node of the JSON, with keys named `PathN`.

```
{  
  "options": {  
    "Path1": "Apple",  
    "Path2": "Phones",  
    "product": "iPhoneX",  
    "isDebug": "true"  
  }  
}
```

Integration Procedure Invocation Using POST

To send JSON data to and receive results from an Integration Procedure, issue a POST call. With a POST call, you can specify inline URL path parameters, append query parameters to the URL, and include a JSON request body.

In this example, an Integration Procedure that creates cases requires a Contact name and returns the Id of the newly created case. The Contact name is specified in a JSON request body.

- POST URL:

```
/services/apexrest/vlocity_ins/v1/integrationprocedure/Create_Cases/
```

- POST JSON data:

```
{
    "Contact": "Dennis Reynolds"
}
```

- Result:

```
{
    "Case": {
        "Id": "0036100001HDn3QAAT"
    }
}
```

In this second example, the syntax of a URL looks like this:

```
/services/apexrest/{namespace}/v1/integrationprocedure/{Type}_{SubType}/{inlin
evalue1}/{inlinevalue2}/?{Param1}={Value1}
```

You can call the URL with these values:

```
/services/apexrest/vlocity_cmt/v1/integrationprocedure/IP_Rest/Apple/Phones/?p
roduct=iPhoneX
```

The POST call sends this input to the Integration Procedure. Values passed in the URL path are added under the `options` node of the JSON, with keys named `PathN`.

```
{
    "options": {
        "Path1": "Apple",
        "Path2": "Phones",
        "product": "iPhoneX",
        "isDebug": "true"
    }
}
```

Integration Procedure Invocation from Salesforce Flow

To call an Integration Procedure from a Salesforce Flow, create a Salesforce Flow component by defining a class using the Salesforce Developer Console, as shown in the following example. Note that the class must have the `@InvocableMethod` annotation.

When calling `IntegrationProcedureService.runIntegrationService`, be sure to replace the

namespace with `omnistudio`, `vlocity_cmt`, `vlocity_ins`, or `vlocity_ps`.

```
global with sharing class IntegrationProcedureInvocable {
    @InvocableMethod(label = 'Integration Procedure')
    global static List < IntegrationProcedureOutput > runIntegrationServiceInvocable(List < IntegrationProcedureInput > input) {
        System.debug(LogLevel.Error, JSON.serialize(input));
        IntegrationProcedureOutput result = new IntegrationProcedureOutput();
        Map<String, Object> ipInput = new Map<String, Object> ();
        Map<String, Object> ipOptions = new Map<String, Object> ();
        result.output = JSON.serialize(
            namespace.IntegrationProcedureService.runIntegrationService(
                input[0].procedureAPIName,
                ipInput,
                ipOptions));
        System.debug(LogLevel.Error, JSON.serialize(result));
        return new List < IntegrationProcedureOutput >
        {
            result
        };
    }
    global class IntegrationProcedureInput
    {
        @InvocableVariable(label = 'Procedure Name') global String procedureAPIName;
        @InvocableVariable(label = 'Input') global String input;
    }
    global class IntegrationProcedureOutput
    {
        @InvocableVariable(label = 'Output') global String output;
    }
}
```

After defining the class, you can use the resulting flow component in the Salesforce Flow Designer to call Integration Procedures. Drag the component from the list to the flow and set its properties as follows.

General Settings: Set **Name** and **Unique Name** to descriptive names for the instance of the component in the flow.

Input Settings

- **Procedure Name:** Specify the Integration Procedure to be run using this format: **Type_SubType** (note the underscore).
- **Input:** For each input variable required by the Integration Procedure, choose **Variable** and specify the name of the input variable using the following format: `{ !variableName }`

Output Settings: For each variable that is returned by the Integration Procedure, choose **Variable** and specify the name of the output variable using the following format: `{!variableName}`

Batch Jobs for Integration Procedures and Vlocity Open Interfaces

The Vlocity batch framework lets you run an Integration Procedure or `VlocityOpenInterface` on a schedule. Typical use cases are a recurrent billing cycle or a scan for insurance policies that come up for renewal in the next 60 days. When a job is run, it sends its input to an Integration Procedure or Apex method that processes the records.

-  **Note** Batch Actions and Scheduled Jobs are available in Omnistudio if the Omnistudio package was installed before the Salesforce Industries package was installed. See [OmniStudio-First Side-by-Side Package Installation Results](#).

The input can be one of the following:

- Null: The Integration Procedure or `VlocityOpenInterface` finds data using its Actions or methods.
- SOQL query: The result of a static query is the input.
- JSON data with merge fields: The Integration Procedure is called by a business process that has contextual data, such as an Omniscript, and is queued immediately rather than being scheduled.

To define the jobs to be run, you add records to the Vlocity Scheduled Job custom object. To launch scheduled jobs, you can:

- Run the Vlocity batch framework
- Launch jobs programmatically

The following sections describe these tasks in detail.

[Batch Framework Initialization](#)

The Vlocity batch framework launches the jobs that are defined in the Vlocity Scheduled Job custom object. To start the Vlocity batch framework, you must execute Apex code, which creates an instance of the framework and invokes its `start` method.

[Programmatic Job Invocation](#)

Use `VlocityBatchFramework` methods to start scheduled jobs with a Data Source Type of Query. You can start all jobs, multiple jobs, or a single job. You can also invoke an Integration Procedure or `VlocityOpenInterface` as a job.

Batch Framework Initialization

The Vlocity batch framework launches the jobs that are defined in the Vlocity Scheduled Job custom object. To start the Vlocity batch framework, you must execute Apex code, which creates an instance of the framework and invokes its `start` method.

```
VlocityBatchFramework batchFramework = new VlocityBatchFramework();  
batchFramework.startBatchScheduler('<frequency>');
```

These commands create a [CronTrigger](#) (scheduled job) with the name `VlocityScheduledJob`.

The **Frequency** parameter specifies how often the batch framework checks for jobs to launch. To ensure that jobs are executed on the desired schedule, specify a frequency greater than or equal to that of the highest-frequency job. For example, if you have a job that is scheduled to run hourly, specify [Hourly](#) or [15 Minutes](#) for the batch framework frequency. Note that the timing of job execution is controlled entirely by the batch framework, and jobs might not be executed at precise intervals.

```
VlocityBatchFramework batchFramework = new VlocityBatchFramework();  
batchFramework.startBatchScheduler('15 Minutes');
```

Programmatic Job Invocation

Use `VlocityBatchFramework` methods to start scheduled jobs with a Data Source Type of Query. You can start all jobs, multiple jobs, or a single job. You can also invoke an Integration Procedure or `VlocityOpenInterface` as a job.

To start all scheduled jobs with a Data Source Type of Query immediately, invoke the `VlocityBatchFramework.startScheduledJobs()` method using anonymous Apex, omitting any parameters, as follows:

```
VlocityBatchFramework.startScheduledJobs();
```

To start a single scheduled job with a Data Source Type of Query immediately, invoke the `VlocityBatchFramework.startScheduledJob()` method using anonymous Apex, specifying the ID for the job you want to run. For example:

```
VlocityBatchFramework.startScheduledJob('a3Zf4000000J81v');
```

To start multiple scheduled jobs with a Data Source Type of Query immediately, invoke the `VlocityBatchFramework.startScheduledJobs()` method using anonymous Apex, specifying a list of IDs for the jobs that you want to run. For example:

```
VlocityBatchFramework.startScheduledJobs(new List<Id>{'a3Zf4000000J81v',  
'a3Zf4000000IwIU'});
```

-  **Note** The batch framework doesn't support invocation of scheduled jobs with Data Source Type values of No Input, Data Input, or List Input.

The batch framework also enables you to programmatically invoke any Integration Procedure or *VlocityOpenInterface* as a job. The methods for starting batch jobs are as follows:

```
VlocityBatchFramework.startIntegrationProcedureBatch(String integrationProcedureKey, List<Object> input, new Map<String, Object> options);
VlocityBatchFramework.startOpenInterfaceBatch(String classMethod, List<Object> input, new Map<String, Object> options);
```

To start an Integration Procedure job from Apex, issue the following command:

```
VlocityBatchFramework.startIntegrationProcedureBatch({integration procedure key}, input list, options map);
```

For example:

```
VlocityBatchFramework.startIntegrationProcedureBatch('VlocityBatchFramework_Va
lidateAccountBillingStreet', accounts, new Map<String, Object>{'isInputAsList'
=> true, 'batchSize' => 20});
```

To start an asynchronous Apex job that runs the *invokeMethod* of a specified class, issue the following command:

```
VlocityBatchFramework.startOpenInterfaceBatch({ Class name . method name }, in
put list, options map);
```

For example:

```
VlocityBatchFramework.startOpenInterfaceBatch('VlocityBatchFrameworkTest0I4.up
dateStatus', accounts, new Map<String, Object>{'isInputAsList' => false, 'batc
hSize' => 20});
```

To chain scheduled jobs that run Integration Procedures, provide the IDs of the jobs and include a chainable option that is set to true:

```
startScheduledJobs(new Map<String, Object>{'vlocityScheduledJobIds' => new Li
st<String>{'JobId', 'JobId2'}, new Map<String, Object>{'chainable' => true}});
```

For example:

```
VlocityBatchFramework.startScheduledJobs(new Map<String, Object>{'vlocitySched
uledJobIds' => new List<String>{'a3Zf4000000T7A1', 'a3Zf4000000T7A6'}}, new Ma
```

```
p<String, Object>{'chainable' => true});
```

You can specify the following options in the options map when invoking a job programmatically. See above for details.

- `batchSize`
- `chainable`: Only jobs containing Integration Procedures are chainable.
- `isInputAsList`
- `jobQueue`

The following methods enable you to manage job execution programmatically:

- `getCurrentBatchJobs()` : Returns AsyncApexJobs that are not "Aborted", "Completed", or "Failed".
- `abortBatchJobs()` : Aborts all AsyncApexJobs named "VlocityBatchFramework".
- `getScheduledBatchJob()` : Returns a CronTrigger named "VlocityScheduledJob".
- `abortScheduledBatchJob()` : Tries to abort the CronTrigger named "VlocityScheduledJob".

Long-Running Integration Procedures

When invoking long-running Integration Procedures you can avoid hitting Salesforce governor limits by using chainable and queueable chainable settings or Apex Continuations, or by chaining on one or more specific long-running steps.

Settings for Long-Running Integration Procedures

You can use chainable and queueable chainable settings to avoid hitting Salesforce governor limits when invoking long-running Integration Procedures. You can also chain on one or more specific long-running steps.

Continuation in Long-Running Calls

To support long-running calls, Salesforce provides Apex Continuations. If your Omniscript calls a long-running Integration Procedure or Apex class, you can enable continuation. If your Flexcard calls a long-running Integration Procedure that has an HTTP action or a Remote Action, you can enable continuation.

Settings for Long-Running Integration Procedures

You can use chainable and queueable chainable settings to avoid hitting Salesforce governor limits when invoking long-running Integration Procedures. You can also chain on one or more specific long-running steps.

By default, all the actions in an Integration Procedure run in a single transaction. If the transaction exceeds a Salesforce governor limit, Salesforce ends the transaction and the Integration Procedure fails. You can't set a limit that exceeds the maximum imposed by Salesforce. For details about governor limits, see the relevant [Salesforce topic](#).

To mitigate this problem, use these settings:

- Enable chaining from the Omniscript or parent Integration Procedure action that calls the long-running Integration Procedure.
- Optionally, set limits that trigger chaining in the long-running Integration Procedure itself using the settings described here. These limits define the threshold points at which Integration Procedures start chaining.

When an Integration Procedure with chaining enabled exceeds a governor limit, step results are saved and the step continues in a separate transaction. Because the step runs in its own transaction, it doesn't contend for resources with other steps in the Integration Procedure, so it's less likely to hit governor limits.

Breaking the Integration Procedure execution into chunks and saving interim results can reduce performance. To maximize performance, steps are chained only when they hit governor limits. If no limits are exceeded, all steps in the Integration Procedure run in a single transaction.

You can circumvent Salesforce governor limits by allowing a chainable step to start a queueable job, which runs as an Async Apex Job. These jobs have higher CPU, SOQL, and Heap size limits than regular transactions. Starting a queueable job helps to ensure that governor limits aren't exceeded, but it can reduce performance.

For Developer Edition and Trial orgs, the maximum stack depth for chained jobs is 5, which means that you can chain jobs four times. This limit of 5 includes the initial parent queueable job. See [Queueable Apex](#) in Salesforce Help.

 **Note** To execute multiple transactions, you must call a chainable or queueable chainable Integration Procedure from a context that can act as a client. For example, you can invoke a chainable Integration Procedure from a REST API. You can't call a chainable or queueable chainable Integration Procedure from an Apex class.

Chainable and Queueable Chainable Settings

To configure limits below the Salesforce defaults, edit the **Chainable Configuration** and **Queueable Chainable Limits** settings in the Integration Procedure's **Procedure Configuration** section. To disable checking for a limit, leave it blank. If you disable checking for a limit and a step exceeds the Salesforce limit, the Integration Procedure fails.

Chainable settings are as follows:

- **Chainable Queries Limit:** Maximum number of SOQL queries that can be issued. The default maximum is 100.
- **Chainable DML Statements Limit:** Maximum number of Data Manipulation Language (DML) statements that can be issued. The default maximum is 150.
- **Chainable CPU Limit:** Maximum CPU time on the Salesforce servers. The default maximum is 10,000 milliseconds.

- **Chainable Heap Size Limit:** Memory is used to store data during transaction processing. The default maximum is 6 MB.
- **Chainable DML Rows Limit:** Maximum number of records that can be processed as a result of DML statements, Approval.process, or database.emptyRecycleBin. The default maximum is 10,000.
- **Chainable Query Rows Limit:** Maximum number of rows that a SOQL query is permitted to retrieve. The default maximum is 50,000.
- **Chainable SOSL Queries Limit:** Maximum number of SOSL queries that can be issued. The default maximum is 20.
- **Chainable Actual Time:** The number of seconds an Integration Procedure can run before chaining occurs to avoid reaching the [Salesforce Concurrent Request Limit](#). No default.

 **Note** **Chainable Queries Limit, Chainable Query Rows Limit, and Chainable SOSL Queries Limit** work within a managed package but not outside of it. For example, if a Remote Action runs queries outside the package, these queries don't count toward the limits.

Queueable Chainable settings are:

- **Queueable Chainable Heap Size Limit:** Memory is used to store data during transaction processing. The default maximum is 12 MB.
- **Queueable Chainable CPU Limit:** Maximum CPU time on the Salesforce servers. The default maximum is 60,000 milliseconds.
- **Queueable Chainable Queries Limit:** Maximum number of SOQL queries that can be issued. The default maximum is 200.

Queueable Chainable settings override their Chainable equivalents if both are set.

How to Call a Chainable or Queueable Chainable Integration Procedure

To enable chaining for an Integration Procedure that is called from an Omniscript, open the Integration Procedure action that calls the Integration Procedure and check the **Chainable** checkbox. To enable chainable steps to start queueable jobs, also check the **Queueable** checkbox.

If you're calling an Integration Procedure from a REST API, you can set the `chainable` or `queueableChainable` option to `true`.

A parent Integration Procedure can disable the Chainable settings of a child Integration Procedure that it calls.

The Chain on Step Setting

To enable chaining for a particular long-running step in an Integration Procedure, check its **Chain On Step** setting. When the Integration Procedure is called with chaining enabled, and a limit is reached, the step runs in its own transaction.

Chain On Step isn't required for chaining to occur. If no step has **Chain On Step** applied, chaining occurs

when the Integration Procedure's resource use approaches governor limits or a chainable or queueable chainable setting.

If your Integration Procedure has an Omnistudio Data Mapper Post action followed by an HTTP action, enable **Chain On Step** for the Data Mapper Post action to avoid this Salesforce error: You have uncommitted work pending. Commit or rollback before calling out.

Chainable and Queueable Chainable Settings in Preview

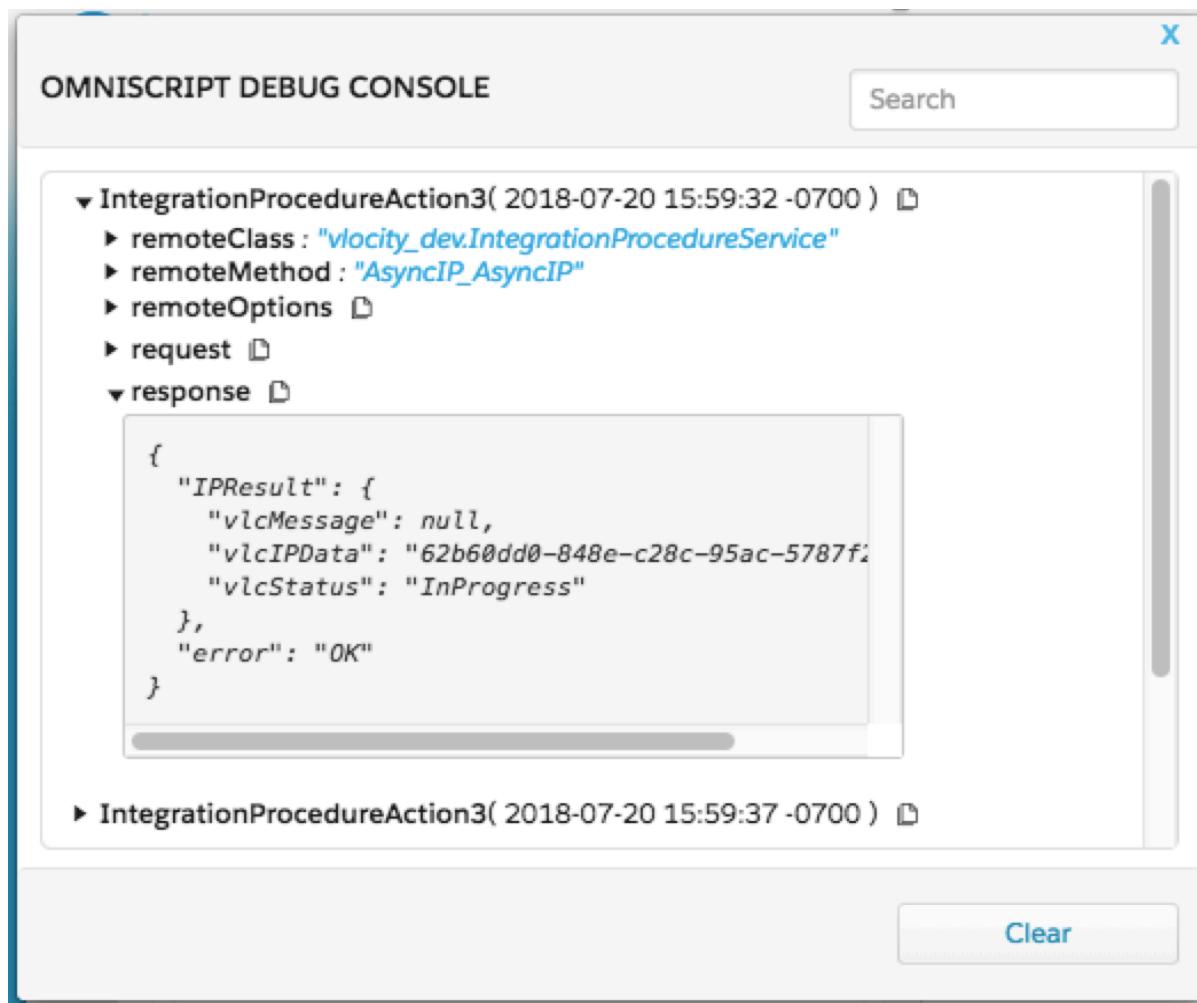
To enable the Chainable or Queueable Chainable settings when testing an Integration Procedure using the Preview tab, expand the Options pane and set one or more of these options to `true`:

- `chainable` : To enable the Chainable settings for testing.
- `queueableChainable` : To enable the Queueable Chainable settings for testing.
- `useQueueableApexRemoting` : To limit the amount of time the Apex CPU runs by starting a queueable job with no chaining.

The names of all options, for example `queueableChainable`, are case-sensitive in all contexts.

The Action Message Property in the Calling Omniscript

To view the progress of a chained Integration Procedure when debugging the OminScript that calls it, set the **Action Message** property of the chained steps in the Integration Procedure. The text you enter in the Action Message field can be viewed in the Debug pane of the Omniscript Designer.



JavaScript Code to Call Chainable Integration Procedures

Omniscripts and parent Integration Procedures that call chainable Integration Procedures handle the transactions automatically. However, if you call a chainable Integration Procedure from a [Lightning Web Component](#), you must include code that handles the intermediate responses.

To see what the intermediate responses look like, see [Invoke a Chainable Integration Procedure with REST Calls](#).

Here's a way to structure JavaScript code to handle these responses:

```
function runChainable(options) {  
    options.isDebugEnabled = vm.isDebugEnabled;  
    remoteActions.testIntegrationProcedure(ipId, inputData, options).then(function(response) {  
        responseHandler(response);  
    });  
}
```

```
function responseHandler(response) {  
    if (typeof(response) === 'string') {  
        response = JSON.parse(response);  
    }  
    if (response &&  
        response.vlcIPData &&  
        response.vlcStatus === 'InProgress') {  
        runChainable(response);  
    } else {  
        // handle IP response  
    }  
}
```

Invoke a Chainable Integration Procedure with REST Calls

A chainable Integration Procedure is split into multiple transactions to avoid hitting Salesforce governor limits. Using a REST API is the easiest way to see the partial responses each transaction returns before the Integration Procedure completes.

Invoke a Chainable Integration Procedure with REST Calls

A chainable Integration Procedure is split into multiple transactions to avoid hitting Salesforce governor limits. Using a REST API is the easiest way to see the partial responses each transaction returns before the Integration Procedure completes.

Before You Begin

- Determine the namespace that Integration Procedures in your org use; for example, `omnistudio`.
- Create this Integration Procedure example or import its DataPack: [Create a Try-Catch Block Example with a Formula](#).
- Get the session ID for your current login to your org. See [Set Up Authorization](#) in Salesforce's *REST API Developer Guide*.

Use the Integration Procedure with the REST API:

1. Open the Integration Procedure example in your org. If you imported the DataPack, it's listed under [Documentation/IPTryCatch2](#) or [Documentation/IPTryCatchF2](#).
2. In the Preview tab, try different Name input parameter values until you find one that returns more than one record but not many. An ideal Name value returns two names.
3. In the Procedure Configuration, click **Deactivate Version** if the Integration Procedure is active so you can edit it. Set the **Chainable Query Rows Limit** to a value low enough to cause chaining.
For example, if the Name value in the previous step returns two records, a **Chainable Query Rows Limit** value of 1 results in two transactions. If the Name value returns three records, a **Chainable Query Rows Limit** value of 2 results in three transactions.
4. Go to the DRExtractAction1 component and check its **Chain on Step** property.

5. Go back to the Procedure Configuration and click **Activate Version**.
6. In a terminal, enter a curl command of this form to call the REST API:

```
curl -X POST \
-H 'Authorization: Bearer session_id' \
-H 'Content-Type: application/json' \
-H 'chainable:true' \
--data-raw '{"Name":"Name"}' \
https://ap1.salesforce.com/services/apexrest/namespace/v1/integrationprocedure/Type_SubType
```

For example, if the Name is `Jones`, the namespace is `omnistudio`, the Type is `Documentation`, and the SubType is `IPTryCatchF2`, enter this curl command. The session ID, which is a long string of the form `00Dnn00000nnnn!ARUAQLt...QqodNY3`, is shown as `SessionId` for clarity.

```
curl -X POST \
-H 'Authorization: Bearer SessionId' \
-H 'Content-Type: application/json' \
-H 'chainable:true' \
--data-raw '{"Name":"Jones"}' \
https://MyDomainName.my.salesforce.com/services/apexrest/omnistudio/v1/integrationprocedure/Documentation_IPTryCatchF2
```



Tip You can assemble the curl command in a text editor and copy it into the terminal.

7. Examine the response, which contains the `vlcIPData`:

```
{  
    "vlcUseQueueableApexRemoting": false,  
    "vlcMessage": null,  
    "vlcIPData": "dc3e1986-2f7c-0871-e968-28eb63e11820",  
    "vlcStatus": "InProgress"  
}
```



Note Integration Procedure Preview handles chaining automatically, so you never see this type of response in the Preview.

8. Add a header for the `vlcIPData` to the curl command and enter it:

```
curl -X POST \
-H 'Authorization: Bearer SessionId' \
-H 'Content-Type: application/json' \
-H 'chainable:true' \
-H 'vlcIPData:dc3e1986-2f7c-0871-e968-28eb63e11820' \
--data-raw '{"Name":"Jones"}' \
https://MyDomainName.my.salesforce.com/services/apexrest/omnistudio/v1/integ
```

```
rationprocedure/Documentation_IPTryCatchF2
```

9. Examine the response. If it contains another `vlcIPData` value, repeat the previous step with the new `vlcIPData` value. Keep repeating with each new `vlcIPData` value until the response contains no such value.

This response is an example of what's returned when the Integration Procedure is complete.

```
{  
    "response": {},  
    "ResponseAction1Status": true,  
    "Name": "Jones",  
    "options": {  
        "Accept": "*/*",  
        "X-B3-SpanId": "ee09609f22d890e1",  
        "CipherSuite": "ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 256-bits",  
        "User-Agent": "curl/7.64.1",  
        "X-B3-Sampled": "0",  
        "Host": "MyDomainName.my.salesforce.com",  
        "X-B3-TraceId": "ee09609f22d890e1",  
        "chainable": "true",  
        "X-Salesforce-SIP": "42.42.42.42",  
        "Content-Type": "application/json"  
    },  
    "DRExtractAction1Status": true,  
    "DRExtractAction1": [  
        {  
            "ContactName": "John Jones"  
        },  
        {  
            "ContactName": "June Jones"  
        }  
    ],  
    "TryCatchBlock1Status": true,  
    "TryCatchBlock1": null  
}
```

Continuation in Long-Running Calls

To support long-running calls, Salesforce provides Apex Continuations. If your Omniscript calls a long-running Integration Procedure or Apex class, you can enable continuation. If your Flexcard calls a long-running Integration Procedure that has an HTTP action or a Remote Action, you can enable continuation.

-  **Note** Beginning Winter '23, Flexcards supports long-running calls from an Integration Procedure that have an HTTP Action or a Remote Action.

For Flexcards that call long-running Integration Procedures that have an HTTP Action or a Remote Action, enable continuation support by adding a new Options Map to the Integration Procedure data source. Enter `useContinuation` as the key and `true` as the value.

For Omniscripts that call long-running Integration Procedures, check the **Use Continuation** setting in the Integration Procedure Action.

For Omniscripts that call Apex classes using Remote Actions, you have two options:

- [Make a Long-Running Remote Call Using VlocityContinuationIntegration](#) – This approach is older, still supported but deprecated.
- [Make a Long-Running Remote Call Using Omnistudio.OmniContinuation](#) – This approach is recommended because it's namespace-independent.

[Make a Long-Running Remote Call Using VlocityContinuationIntegration](#)

To support long-running remote calls, Vlocity supports the use of the Salesforce Continuation object. The `VlocityOpenInterface2` interface and `VlocityContinuationIntegration` class support normal Remote Calls and Remote Calls that use the Continuation Object. For the new Apex classes, don't use `VlocityOpenInterface`.

[Make a Long-Running Remote Call Using Omnistudio.OmniContinuation](#)

To support long-running remote calls, Vlocity supports the use of the Salesforce Continuation object.

Make a Long-Running Remote Call Using VlocityContinuationIntegration

To support long-running remote calls, Vlocity supports the use of the Salesforce Continuation object. The `VlocityOpenInterface2` interface and `VlocityContinuationIntegration` class support normal Remote Calls and Remote Calls that use the Continuation Object. For the new Apex classes, don't use `VlocityOpenInterface`.

For more information on the SFDC Continuation object, see the following Salesforce documentation:

- [Process for Using Asynchronous Callouts](#)
- [Apex Continuations: Asynchronous Callouts from Visualforce Pages](#)

To make a remote call using the Salesforce Continuation object and extending the `VlocityContinuationIntegration` class:

1. Create an Apex class that extends the `VlocityContinuationIntegration` class.

This class implements `VlocityOpenInterface2`.

2. Implement the callback method in the `invokeMethod`. For more information, refer to the sample class at the end of this topic.
3. For the method that returns the Continuation Object, call the method `VlocitySetAsyncCallbackState`

before the return.

```
con.continuationMethod = 'customcallback'; // implemented in invokeMethod
VlocitySetAsyncCallbackState(con, con.addHttpRequest(req), options);
```

- For the last callback method in the chain that returns the response back to Omniscrypt, use the following code to get the state of the Continuation Object and system labels.

```
Object state = inputMap.get('vlcContinuationCallbackState');
Object labels = inputMap.get('vlcContinuationCallbackLabels');
```

- The 'state' Object returned from `inputMap.get('vlcContinuationCallbackState')` must be a `Map<String, Object>` with contents that can't be directly cast using `(CustomApexWrapperClass)inputMap.get('vlcContinuationCallbackState')`. Instead, the returned object must be serialized to JSON and then deserialized and cast to its correct class. For example:

```
CustomApexWrapperClass wrap = (CustomApexWrapperClass)JSON.deserialize(JSON.serialize(state), CustomApexWrapperClass.class);
```

This approach permits any data written to the `VlocitySetAsyncCallbackState` to be retained and used as its custom Apex class.

Sample Class VlocityContinuationIntegrationTest.cls

```
// Sample Apex class for making Remote Call in OmniScript
// (1) Create a custom Apex class which extends VlocityContinuationIntegration,
//      for a Vlocity managed package,
//      need to include the Namespace prefix - NS.VlocityContinuationIntegration
// (2) implement invokeMethod, return type is Object (a) in the Continuation
//      Object case, return Continuation Object
// (b) in the normal case, you can return Boolean
global with sharing class VlocityContinuationIntegrationTest extends Vlocity
ContinuationIntegration
{
    global override Object invokeMethod(String methodName, Map<String, Object> i
nputMap, Map<String, Object> outMap, Map<String, Object> options)
    {
        Boolean result = true;
        try
        {
            // the custom methods can have any customized signature, but
            // PLEASE MAKE SURE YOU ALWAYS PASS IN options
            if(methodName.equals('Continuation1'))
            {
```

```
// this returns Continuation Object
return Continuation1(10, inputMap, outMap, options);
}
else if(methodName.equals('Continuation2'))
{
    return Continuation2(8, options);
}
else if(methodName.equals('customcallback'))
{
    customcallback(inputMap, outMap, options);
}
// other methods to handle normal Remote Call
else
{
    result = false;
}
}
catch(System.Exception e)
{
    // System.log ...
    result = false;
}
return result;
}
// You can have other parameters as well, but make sure you always pass in Map<String, Object> options
private Object Continuation1(Integer count, Map<String, Object> inputMap, Map<String, Object> outMap, Map<String, Object> options)
{
    // THIS IS A SAMPLE TO CALL HEROKU NODE SERVICE
    // Make an XMLHttpRequest as we normally would
    // Remember to configure a Remote Site Setting for the service!
    String url = 'https://node-count.herokuapp.com/' + count;
    >HttpRequest req = new HttpRequest();
    req.setMethod('GET');
    req.setEndpoint(url);
    // Create a Continuation for the XMLHttpRequest
    Continuation con = new Continuation(60);
    //
    // Please set up callback method here
    // (1) Include this callback method in the above invokeMethod, refer to
    //      else if(methodName.equals('Continuation2'))
    // (2) methodName is CASE SENSITIVE
    con.continuationMethod = 'Continuation2';
```

```
// The following method MUST BE CALLED
// params:
// (1) first parameter = Continuation Object
// (2) second parameter = WHATEVER YOU WANT TO SET THE state parameter of
the CONTINUATION Object
//      https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/ap
excode/apex_continuation_process.htm
// (3) third parameter = options
// what it does:
// restructure the state property of the Continuation Object to be passed
to the next Callback
// con.state is a Map, which contains:
// (a) vlcContinuationCallbackState - the state you want to set, in this
example, con.addHttpRequest(req)
// (b) vlcContinuationCallbackLabels - labels, refer to
//      https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/ap
excode/apex_continuation_process.htm
VlocitySetAsyncCallbackState(con, con.addHttpRequest(req), options);
// Return it to the system for processing
return con;
}
// callback for the first Continuation
// Returns Continuation Object
// This is to show you how to serialize multiple long running remote calls
private Object Continuation2(Integer count, Map<String, Object> options)
{
    // EXAMPLE
    // Make an HTTPRequest as we normally would
    // Remember to configure a Remote Site Setting for the service!
    String url = 'https://node-count.herokuapp.com/' + count;
    HttpRequest req = new HttpRequest();
    req.setMethod('GET');
    req.setEndpoint(url);
    // Create a Continuation for the HTTPRequest
    Continuation con = new Continuation(60);
    // (1) This callback method should be included in the above invokeMethod,
refer to above
    //      else if(methodName.equals('customcallback'))
    con.continuationMethod = 'customcallback';
    // The following line has to be called
    VlocitySetAsyncCallbackState(con, con.addHttpRequest(req), options);
    // Return it to the system for processing
    return con;
}
```

```
// Last callback method
// This does NOT return Continuation Object
// This will return the response back to OmniScript, therefore need to set outMap
private Object customcallback(Map<String, Object> inputMap, Map<String, Object> outMap, Map<String, Object> options)
{
    // This is how you access the state and labels passed by the Continuation Object
    Object state = inputMap.get('vlcContinuationCallbackState');
    Object labels = inputMap.get('vlcContinuationCallbackLabels');
    // EXAMPLE
    HttpResponse response = Continuation.getResponse((String)state);
    Integer statusCode = response.getStatusCode();
    if (statusCode >= 2000)
    {
        // System.log ...
    }
    // This goes back to OmniScript
    outMap.put('continuousResp', response.getBody());
    return null;
}
```

Make a Long-Running Remote Call Using Omnistudio.OmniContinuation

To support long-running remote calls, Vlocity supports the use of the Salesforce Continuation object.

-  **Note** Beginning Winter '23, Flexcards supports long-running calls from an Integration Procedure that has an HTTP Action or a Remote Action.

For details about the SFDC Continuation object and the *Callable* interface, see the following Salesforce documentation:

- [Process for Using Asynchronous Callouts](#)
- [Apex Continuations: Asynchronous Callouts from Visualforce Pages](#)
- [Callable Interface](#)

To make a remote call using the *OmniContinuation* object and implementing the *Callable* interface:

1. Create an Apex class that implements the *Callable* interface.
2. Implement the callback method in the *invokeMethod*. See the `private Object invokeMethod` code in the example.
3. For the method that returns the *OmniContinuation* Object, call the *Omnistudio.OmniContinuation*

constructor.

```
OmniStudio.OmniContinuation con = new OmniStudio.OmniContinuation(120);  
con.ContinuationMethod = 'customCallback';
```

4. For the last callback method in the chain that returns the response back to the Flexcard or Omniscript, use the following code to get the state of the OmniContinuation Object and system labels.

```
Object state = inputMap.get('vlcContinuationCallbackState');  
Object labels = inputMap.get('vlcContinuationCallbackLabels');
```

5. Use the `Omnistudio.OmniContinuation.getResponse` method to retrieve the response.

Sample Class OmniContinuationTest.cls

```
global with sharing class RemoteActionCallableOmniContinuation implements Callable  
{  
    // Dispatch actual methods  
    public Object call(String action, Map<String, Object> args) {  
  
        Map<String, Object> input = (Map<String, Object>)args.get('input');  
        Map<String, Object> output = (Map<String, Object>)args.get('output');  
        Map<String, Object> options = (Map<String, Object>)args.get('options');  
  
        return invokeMethod(action, input, output, options);  
    }  
  
    private Object invokeMethod(String methodName, Map<String, Object> inputMap, Map<String, Object> outMap, Map<String, Object> options)  
    {  
        Boolean result = true;  
        try  
        {  
            // the custom methods can have any customized signature, but  
            // PLEASE MAKE SURE YOU ALWAYS PASS IN options  
            if(methodName.equals('populateElements'))  
            {  
                // this returns Continuation Object  
                return populateElements(5, inputMap, outMap, options);  
            }  
            else if(methodName.equals('continuation2'))  
            {  
                return continuation2(8, inputMap, options);  
            }  
        }  
    }  
}
```

```
        }

        else if(methodName.equals('customCallback'))
        {
            customCallback(inputMap, outMap, options);
        }
        else
        {
            result = false;
        }
    }

    catch(System.Exception e)
    {
        result = false;
    }

    //outMap.put('nothing', methodName);

    return result;
}

// You can have other parameters as well, but make sure you always pass
in Map<String, Object> options
private Object populateElements(Integer count, Map<String, Object> input
Map, Map<String, Object> outMap, Map<String, Object> options)
{
    // THIS IS A SAMPLE TO CALL HEROKU NODE SERVICE
    //

    // Make an XMLHttpRequest as we normally would
    // Remember to configure a Remote Site Setting for the service!
    String url = 'https://node-count.herokuapp.com/' + count;
    HttpRequest req = new HttpRequest();
    req.setMethod('GET');
    req.setEndpoint(url);

    // Create a Continuation for the XMLHttpRequest
    OmniStudio.OmniContinuation con = new OmniStudio.OmniContinuation(6
0);
    con.ContinuationMethod = 'continuation2';

    //

    // Create the state
    Map<String, Object> stateAsMap = new Map<String, Object>();
    String label = con.addHttpRequest(req);
    stateAsMap.put('label', label);
```

```
// Store the state
con.state = stateAsMap;

options.put('continuationState', label);
options.put('continuationCallback', 'continuation2');

// Return it to the system for processing
return con;
}

// callback for the first Continuation
// this again returns Continuation Object
// This is to show you how to serialize multiple long running remote cal
ls
private Object continuation2(Integer count, Map<String, Object> inputMa
p, Map<String, Object> options)
{
    String url = 'https://node-count.herokuapp.com/' + count;
    HttpRequest req = new HttpRequest();
    req.setMethod('GET');
    req.setEndpoint(url);

    // Create a OmniContinuationTest for the HttpRequest
    OmniStudio.OmniContinuation con = new OmniStudio.OmniContinuation(12
0);
    con.ContinuationMethod = 'customCallback';

    // Create the state
    Map<String, Object> stateAsMap = new Map<String, Object>();
    String label = con.addHttpRequest(req);
    stateAsMap.put('label', label);
    stateAsMap.put('populateElements-label', inputMap.get('vlcContinuati
onCallbackState'));

    // Save the state
    con.state = stateAsMap;

    options.put('continuationState', label);
    options.put('continuationCallback', 'customCallback');

    // Return it to the system for processing
    return con;
}
```

```
// Last callback method
// This does NOT return Continuation Object
// This will return the response back to the FlexCard or OmniScript, therefore need to set outMap
private Object customCallback(Map<String, Object> inputMap, Map<String, Object> outMap, Map<String, Object> options)
{
    // This is how you access the state and labels passed by the Continuation Object
    Object state = inputMap.get('vlcContinuationCallbackState');
    Object labels = inputMap.get('vlcContinuationCallbackLabels');

    Map<String, Object> stateAsMap = (Map<String, Object>) state;

    // EXAMPLE
    HttpResponse response = OmniStudio.OmniContinuation.getResponse((String)stateAsMap.get('label'));
    Integer statusCode = response.getStatusCode();

    if (statusCode >= 2000)
    {
        // System.log ...
    }

    // This goes back to the FlexCard or OmniScript
    outMap.put('responseFromCallableOmniContinuation', 'Response using OmniContinuationWrapperPtc (Using Callable): ' + response.getBody());

    return null;
}
}
```

Security for Omnistudio Data Mappers and Integration Procedures

You can control access to Data Mappers and Integration Procedures using settings that reference Sharing Settings and Sharing Sets or Profiles and Permission Sets.

- !** **Important** Guest Users, also called anonymous users, cannot access any records by default. Criteria-based Sharing Rules grant them read-only access. This affects all Salesforce orgs. For details, see [Guest User Record Access Development Best Practices](#). Vlocity allows guest users to create and

update the records to which [Sharing Rules](#) grant access. No additional configuration is necessary for this expanded access.

You can use Salesforce [Sharing Settings](#) to secure access to Data Mappers and Integration Procedures. If you use [caching](#), you must set **CheckCachedMetadataRecordSecurity** to true as described here.

You can allow access to a Data Mapper or Integration Procedure based on the Custom Permissions enabled in a user's Salesforce [Profiles](#) or [Permission Sets](#). An Apex class added to your Salesforce Org allows the Vlocity Managed Package to check user Custom Permissions. The custom settings described here are related to this approach. Vlocity recommends using Custom Permissions in Profiles or Permission Sets for ease of use and better performance.

 **Note** To ensure data security and maintain compliance with Salesforce encryption access controls, always check that a user has the **View Encrypted Data** permission before displaying or processing decrypted values of encrypted fields.

For Salesforce access basics, see [Control Who Sees What](#) and [Salesforce Data Security Model – Explained Visually](#). For Vlocity-specific information about Profiles, see [Overview of Profiles and Security for Vlocity](#)

Sharing Settings, Sharing Sets, Profiles, and Permission Sets control access to Data Mappers and Integration Procedures as object records.

To ensure field-level security for a Data Mapper, go to the Data Mapper's Options tab and select **Check Field Level Security**. To automatically enforce field-level security for all Data Mappers, enable *EnforceDMFLSAndDataEncryption* in the Omni Interaction Configuration.

If you're using the Omnistudio standard designer, and if the user has the **View Encrypted Data** permission, the classic encrypted fields are shown in plain text for that user by default. However, if you are using Omnistudio managed package designer, you must enable *EnforceDMFLSAndDataEncryption* in the Omni Interaction Configuration to enforce this behavior.

To enable *EnforceDMFLSAndDataEncryption* in the Omni Interaction Configuration, follow these steps: From Setup, search and open Omni Interaction Configuration. Click **New Omni Interaction Configuration**, enter *EnforceDMFLSAndDataEncryption* for both name and label, set the value to *true*, and save your changes.

 **Important** A user's access to a Data Mapper or Integration Procedure includes more than the ability to run it directly. Access also applies if an application the user is using [calls](#) the Data Mapper or Integration Procedure. If a user has access to a parent Integration Procedure, the parent can invoke child Integration Procedures and Data Mappers to which the user doesn't have direct access.

[Configure Omnistudio Data Mapper and Integration Procedure Security Settings](#)

You can change settings for Data Mapper and Integration Procedure security in Setup.

[Omnistudio Data Mapper and Integration Procedure Security Settings](#)

These settings affect Data Mapper and Integration Procedure security.

Syntax of the Required Permission Property

Omnistudio Data Mappers and Integration Procedures have a Required Permission property, which determines who has runtime access. You can specify roles, profiles, permission sets, custom permissions, or any combination. If Required Permission is blank, any user can run the Data Mapper or Integration Procedure unless the DefaultRequiredPermission property is set.

Implement the `VlocityRequiredPermissionCheck` Class

For the **DefaultRequiredPermission** setting to work, you must implement the `VlocityRequiredPermissionCheck` class manually because Salesforce handles classes in managed and unmanaged packages differently. This class doesn't work properly if it's included in the Vlocity managed package.

Configure Omnistudio Data Mapper and Integration Procedure Security Settings

You can change settings for Data Mapper and Integration Procedure security in Setup.

The settings you configure in this task are listed in [Data Mapper and Integration Procedure Security Settings](#).

These steps apply if you use Omnistudio on the standard runtime.

1. Go to Setup.
 - In Lightning Experience, click the gear icon and select **Setup** from the menu.
 - In Salesforce Classic, click the user menu and select **Setup** from the menu.
2. In the **Quick Find** box, type *omni*, then select **Omni Interaction Configuration**.
3. Click **New Omni Interaction Configuration**.
4. In the **Label** and **Value** fields, type the name and value of the setting.

For a list of settings, see [Data Mapper and Integration Procedure Security Settings](#).

5. Click **Save**.

Omnistudio Data Mapper and Integration Procedure Security Settings

These settings affect Data Mapper and Integration Procedure security.

To configure these settings, see [Configure Omnistudio Data Mapper and Integration Procedure Security Settings](#) if you use Omnistudio.

Setting	Description	Data Type	Default Value
DefaultRequiredPermission	<p>Specifies the default value for the Required Permission setting, which determines which users can run Data Mappers and Integration Procedures.</p> <p>The Required Permission setting, which you can optionally specify when creating a Data Mapper or Integration Procedure, overrides this setting.</p> <p>If this setting is absent or blank, all users can run any Data Mappers or Integration Procedures that don't have Required Permission values.</p> <p>The syntax for this setting matches the Required Permission syntax. See Syntax of the Required Permission Property.</p>	String	(none)
CheckCachedMetadataRecordSecurity	<p>By default, the cached metadata is not secured when Salesforce Sharing Settings or Sharing Sets are used to control access.</p> <p>This setting is disabled by default. If set to true, this setting performs a</p>	True or False	False

Setting	Description	Data Type	Default Value
	record-level security check for cached metadata. This check lessens the performance benefit of metadata caching slightly. This setting isn't needed if you use Custom Permissions to secure access.		
EnforceDMFLSAndDataEncryption	To automatically enforce field-level security for all Data Mappers, enable <i>EnforceDMFLSAndDataEncryption</i> in the Omni Interaction Configuration. See Security for Omnistudio Data Mappers and Integration Procedures .	True or False	(none)

Syntax of the Required Permission Property

Omnistudio Data Mappers and Integration Procedures have a Required Permission property, which determines who has runtime access. You can specify roles, profiles, permission sets, custom permissions, or any combination. If Required Permission is blank, any user can run the Data Mapper or Integration Procedure unless the DefaultRequiredPermission property is set.

! **Important** When a Flexcard, Integration Procedure, or Omniscript calls a Data Mapper or another Integration Procedure, only the required permission of the calling parent component is enforced by Omnistudio. Omnistudio does not automatically enforce the required permission of the child Data Mapper or Integration Procedure. This can lead to security risks if the calling parent component has broader access. To ensure consistent access control, always configure appropriate required permissions on all Flexcards, Integration Procedures, or Omniscripts and their contained actions, especially when handling sensitive data.

The syntax of the Required Permission property is:

```
prefix:name, prefix:name, ...
```

The prefix can be `Role`, `Profile`, `PermSet`, or `CustomPerm`. The name is the name of a role, profile, permission set or permission set group, or custom permission. For example, here's a Required Permission value that includes two roles and one permission set group:

```
Role:Architect,Role:Developer,PermSet:OmniStudio Admin Group
```

Implement the `VlocityRequiredPermissionCheck` Class

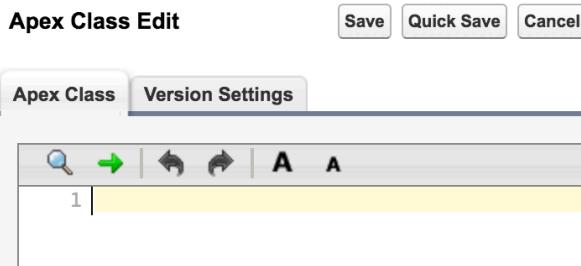
For the `DefaultRequiredPermission` setting to work, you must implement the `VlocityRequiredPermissionCheck` class manually because Salesforce handles classes in managed and unmanaged packages differently. This class doesn't work properly if it's included in the Vlocity managed package.

-  **Note** If you're using Omnistudio Spring '22 or greater, and the **Managed Package Runtime** is set to **Disabled**, you don't have to implement this class.

For `DefaultRequiredPermission` details, see [Omnistudio Data Mapper and Integration Procedure Security Settings](#).

1. From Setup, in the **Quick Find** box, type `apex`.
2. Click **Apex Classes**.
3. Click **New**.

Apex Class



4. Enter the following Apex code in the **Apex Class** tab:

```
global class VlocityRequiredPermissionCheck implements Callable
{
    global Boolean call(String action, Map<String, Object> args)
    {
        if (action == 'checkPermission')
        {
            return checkPermission((String)args.get('requiredPermission'));
        }
    }
}
```

```
        return false;
    }

    private Boolean checkPermission(String requiredPermission)
    {
        Boolean hasCustomPermission = false;
        List<String> customPermissionsName = requiredPermission.split(',');
        for (String permissionName : customPermissionsName)
        {
            Boolean hasPermission = FeatureManagement.checkPermission(permissionName.normalizeSpace());
            if (hasPermission == true)
            {
                hasCustomPermission = true;
                break;
            }
        }
        return hasCustomPermission;
    }
}
```

5. Click **Save**.

Test Procedures: Integration Procedures for Unit Testing

An Integration Procedure that performs a unit test is a *Test Procedure*. You can use a Test Procedure to unit test almost anything an Integration Procedure can invoke, such as an Omnistudio Data Mapper, a Calculation Matrix, an Apex class, or another Integration Procedure.

Using Test Procedures and the test framework, you can:

- Provide sample input to the entity being tested using a [Set Values Action for Integration Procedures](#) component.
- Mock responses of specific steps if the entity being tested is an Integration Procedure.
- Compare expected and actual results using an [Assert Action for Integration Procedures](#) component.
- Use IDX Workbench to [run Test Procedures](#).

After a Test Procedure finishes, its transaction is rolled back. This lets you run tests that create sObjects without affecting the database.

How you organize your Test Procedures is up to you. For example, you can test the same Data Mapper multiple times using different inputs, or you can test several different Data Mappers using the same Test Procedure.

When to Mock Integration Procedure Components

When you use a Test Procedure to test another Integration Procedure, you can simulate, or mock, the responses of some of the components in the testing Integration Procedure or the Integration Procedure being tested. For example, you can mock a component that would normally retrieve a credit score, using a hard-coded score for testing. This has no effect on how the Integration Procedure normally runs when it isn't being tested.

To mock a component that returns a single value, scroll to the bottom of the Procedure Configuration and specify a Key/Value pair under Mock Responses Map. The Key must be the Name of a component, and the Value can be literal text or a merge field.

To mock a component with a more complex response, go to the Procedure Configuration and click **Edit as JSON**, then edit the `mockResponseMap` node. For example, the following JSON code mocks a Set Values component named `IfOtherState` that sets a `DefaultSalesTax` value:

```
"mockResponseMap": {  
    "IfOtherState": {  
        "DefaultSalesTax": 0.06  
    }  
}
```

The following component types must be mocked in the testing Integration Procedure or the Integration Procedure being tested. Mocking other component types is permitted but not required.

Component	Reason for Mocking
Docusign Envelope Action for Integration Procedures	The response to the delivered email isn't available.
Email Action for Integration Procedures	The response to the delivered email isn't available.
HTTP Action for Integration Procedures	Salesforce has specific requirements for testing HTTP callouts .

HTTP Callouts in Called Apex Classes

If a Test Procedure or an Integration Procedure being tested includes a Remote Action, and the Apex class the Remote Action invokes includes an HTTP callout, then you must ensure that the HTTP callout isn't part of the test. To do this, edit the Apex class and enclose the HTTP callout in an `if (IntegrationProcedureService.isTest == false)` statement, for example:

```
if (IntegrationProcedureService.isTest == false) {  
    HttpRequest request = new HttpRequest();  
    request.setMethod('GET');  
    request.setEndpoint('http://example.com');  
  
    Http httpCall = new Http();  
    HttpResponse response = httpCall.send(request);  
}
```

Workflow for Test Procedure Example

A Test Procedure tests a calculation performed by another Integration Procedure.

Workflow for Test Procedure Example

A Test Procedure tests a calculation performed by another Integration Procedure.

To download a DataPack of this example for Omnistudio, click [here](#).

To build this example:

1. Enable the Tracking Service for Integration Procedures.



Note Tracking Service isn't currently available for Omnistudio.

2. Create the Integration Procedure to be tested.

See [Define Execution Logic](#).

3. [Mock a Response in the Tested Integration Procedure](#).

4. [Create an Example Test Procedure](#).

5. Run the Test Procedure from IDX Workbench.

See [Run Test Procedures](#).

Mock a Response in the Tested Integration Procedure

Before you create the example Test Procedure, add a Mock Response Map to the Integration Procedure to be tested.

Create an Example Test Procedure

After you mock a response in the Integration Procedure to be tested, create the Test Procedure that tests a calculation it performs.

Mock a Response in the Tested Integration Procedure

Before you create the example Test Procedure, add a Mock Response Map to the Integration Procedure to be tested.

1. Go to the Omnistudio Integration Procedures tab and open the Integration Procedure to be tested.
2. If the Integration Procedure is active, click **Deactivate Version** so you can edit it.
3. In the Procedure Configuration, click **Edit as JSON**.

4. Edit the `mockResponseMap` node as follows:

```
"mockResponseMap": {  
    "IfOtherState": {  
        "DefaultSalesTax": 0.06  
    }  
}
```

5. Click **Edit in Property Editor**.
6. Click **Activate Version**. The Integration Procedure to be tested must be active.

Create an Example Test Procedure

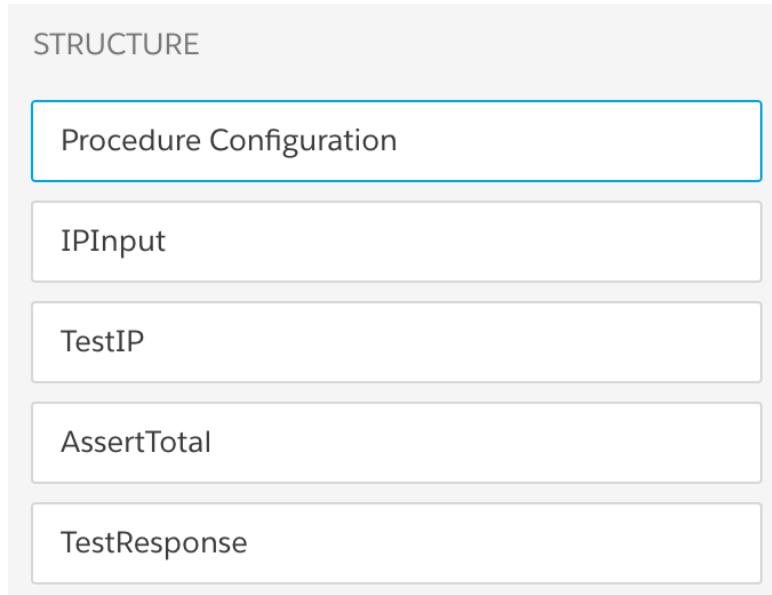
After you mock a response in the Integration Procedure to be tested, create the Test Procedure that tests a calculation it performs.

The Test Procedure has these components:

- A Set Values component, named `IPInput`
- An Integration Procedure Action component, named `TestIP`
- An Assert Action, named `AssertTotal`
- A Response Action, named `TestResponse`

A Response Action isn't required, but it's useful for testing the Test Procedure. After you have verified that the Test Procedure works, you can deactivate or delete it.

The Structure panel looks like this:



To create the Test Procedure:

1. From the Omnistudio Integration Procedures tab, click **New**.

2. Provide an **Integration Procedure Name**, a **Type**, and a **SubType**, and click **Save**.
3. Scroll to the bottom of the Procedure Configuration and check **Is Test Procedure**.
4. Drag a Set Values component into the Structure panel and give it the following settings:

Property	Value
Element Name	IPInput
Element Value Map: Price	250
Element Value Map: State	NY
Response JSON Node	IPInput

5. Drag an Integration Procedure Action component into the Structure panel and give it the following settings:

Property	Value
Element Name	TestIP
Integration Procedure	Name you gave the Integration Procedure to be tested, described in Define Execution Logic .
Send JSON Path	IPInput

6. Drag an Assert Action component into the Structure panel and give it the following settings:

Property	Value
Element Name	AssertTotal
Assert Conditional Formula	TestIP:Output>Total == 265
Assert Failure Message	Calculation is incorrect.

7. Drag a Response Action component into the Structure panel and give it the following settings:

Property	Value
Element Name	TestResponse
Return Full Data JSON	(checked)

After you have verified that the Test Procedure works, you can deactivate this component.

8. Go to the Preview tab and click **Execute**. The output should look like this:

```
{
  "response": {
    "testResult": "success"
  },
  "TestResponseStatus": true,
  "AssertTotal": {
    "assertFailureMessage": "Calculation is incorrect.",
    "variableData": {
      "TestIP:Output:Total": 265
    },
    "assertConditionalFormula": "TestIP:Output:Total == 265",
    "assertResult": true
  },
  "AssertTotalStatus": true,
  "TestIP": {
    "testResult": "success",
    "Output": {
      "Price": 250,
      "State": "NY",
      "Tax": 15,
      "TaxRate": 6,
      "Total": 265
    }
  },
  "TestIPStatus": true,
  "IPIInput": {
    "Price": 250,
    "State": "NY"
  },
  "IPIInputStatus": true,
  "options": {
    "useQueueableApexRemoting": false,
    "queueableChainable": false,
    "ignoreCache": true,
  }
}
```

```
        "resetCache": false,  
        "chainable": false  
    }  
}
```

9. (Optional) In the IPIInput component, change the Price, or change the State to WA, OR, NV, or CA. Then return to the Preview tab and click **Execute** again. Note how the output changes.
10. Prepare for [running the Test Procedure](#).
 - a. In the TestResponse component, click **Active** to remove the check.
 - b. In the Procedure Configuration, click **Activate Version**. IDX Workbench only runs active Test Procedures.

See Also

[Test Procedures: Integration Procedures for Unit Testing](#)
[Workflow for Test Procedure Example](#)

Integration Procedure Actions

To compose an Integration Procedure, you add actions that run sequentially. These actions can set data values, perform functions, call Omnistudio Data Mappers, invoke Apex classes, send emails, invoke REST endpoints, run other Integration Procedures, and more.

To add actions, drag elements from the Action elements panel to the canvas or use the connectors (+) directly on the canvas. You can easily add, edit, rename, move, or delete actions, and rearrange elements between blocks using +. You can also use blocks to group actions for conditional execution, caching, list processing, and error handling.

This table describes the properties common to all Integration Procedure Actions.

Property	Description
Element Name	Name of the element or node.
Send JSON Path	Trims the incoming JSON to the specified path before the action is executed. See Manipulate JSON with the Send/Response Transformations Properties .
Send JSON Node	Represents the incoming JSON under the specified node. See Manipulate JSON with the Send/Response Transformations Properties .

Property	Description
Response JSON Path	After the action is executed, trims the output JSON to the specified path. See Manipulate JSON with the Send/Response Transformations Properties .
Response JSON Node	Represents the output JSON under the specified node. To delimit the path, use colons; for example, <code>level1:level2:level3</code> . See Manipulate JSON with the Send/Response Transformations Properties .
Send Only Additional Input	If checked, accepts only the data in the Additional Input property.
Additional Input	Additional key-value pairs to be included in the input JSON data. Values can include formulas and merge fields.
Return only additional output	Returns only the data in the Additional Output property.
Additional Output	Additional key-value pairs to be returned in the output JSON data. Values can include formulas and merge fields.
Send Only Failure Response	Returns only the failure response if the action fails.
Failure Response	Key-value pairs to be returned in the output JSON data if the action fails. Values can include formulas and merge fields.
Execution Conditional Formula	Specifies a formula that runs before the step is executed. If the formula returns TRUE, the step is executed. If the formula returns FALSE, the step isn't executed. If no formula is specified, the step is executed.
Failure Conditional Formula	Specifies a custom failure condition that runs after

Property	Description
	<p>the step is executed.</p> <p>For example, the Omnistudio Data Mapper Extract Action failing to find any records isn't normally considered an error. But you can use a Failure Conditional Formula to specify this condition. See the second example under Handle Errors by Using a Try-Catch Block.</p> <p>If the formula returns TRUE, execution of the step has failed and any key-value pairs configured in the Failure Response list are added to the JSON data. For example, if this JSON is returned:</p> <pre data-bbox="817 819 1460 1121">{ "Result": { "ErrorCode": "ER R-123", "Success": "FALSE" } }</pre> <p>These settings identify the error and add the error code to the JSON data:</p> <ul style="list-style-type: none"> Failure Conditional Formula: Result:Success == 'FALSE' Failure Response: <ul style="list-style-type: none"> Key: ErrorCode Value: %StepName:Result:ErrorCode%
<ul style="list-style-type: none"> Terminate when step fails Fail on Step Error (If you're using the designer on a managed package) 	<p>The Integration Procedure ends if this step fails.</p>
Chain On Step	<p>Allows the action to run in its own Salesforce transaction. This property can slow performance but decreases the likelihood of exceeding the Salesforce governor limits. For more information, see Settings for Long-Running Integration</p>

Property	Description
	Procedures .
Additional Chainable Response	Specifies a key-value pair that is sent in the response. The value field accepts merge field syntax.

[Assert Action for Integration Procedures](#)

The Assert Action compares the expected and actual results of a Test Procedure using an expression that evaluates to true or false. You can use environment variables in the Assert Conditional Formula to test performance.

[Chatter Action for Integration Procedures](#)

The Chatter Action creates a Chatter post and sends it to a Chatter feed.

[Decision Matrix Action for Integration Procedures](#)

The Decision Matrix Action calls a Decision Matrix with specified inputs and returns the result to the Integration Procedure.

[Delete Action](#)

Enable users to delete one or more sObject records by using the Delete Action. Use an Object's Record Id to determine which record to delete. Vlocity recommends using a merge field in the Path to Id field that refers to an Id or a list of Ids in the data JSON.

[DocuSign Envelope Action for Integration Procedures](#)

The DocuSign Envelope Action emails a set of documents for signing.

[Email Action for Integration Procedures](#)

An Email Action sends the specified email. You can either specify all the email field values or use a Salesforce email template.

[Expression Set Action for Integration Procedures](#)

An Expression Set Action invokes the specified Expression Set and returns the results to the Integration Procedure.

[HTTP Action for Integration Procedures](#)

The HTTP Action executes a REST call and returns its results to the Integration Procedure. The HTTP Action in Integration Procedures supports only `getBody()` for returning the response. Use Apex directly via a Remote Action to handle binary responses.

[Integration Procedure Action for Integration Procedures](#)

The Integration Procedure Action runs a subordinate Integration Procedure.

[Intelligence Action for Integration Procedures](#)

The Intelligence Action provides input to and runs an Intelligence Machine on Omnistudio.

[List Action for Integration Procedures](#)

The List Action merges multiple lists by matching values of specified list item JSON nodes. A basic merge matches node names exactly. An advanced merge matches nodes that have different names or reside at different levels in the incoming lists.

Omnistudio Data Mapper Extract Action for Integration Procedures

The Data Mapper Extract Action calls the specified Data Mapper Extract to read data from Salesforce and returns it to the Integration Procedure.

Omnistudio Data Mapper Post Action for Integration Procedures

The Data Mapper Post Action calls a Data Mapper Load (post) to write data to Salesforce.

Omnistudio Data Mapper Transform Action for Integration Procedures

A Data Mapper Transform Action calls the specified Data Mapper Transform to execute transformations on the Data JSON and returns the transformed data.

Omnistudio Data Mapper Turbo Action for Integration Procedures

A Data Mapper Turbo Action calls the specified Data Mapper Turbo Extract to read data from Salesforce and returns it to the Integration Procedure.

Remote Action for Integration Procedures

The Remote Action element calls the specified Apex class and method or the specified invocable action.

Response Action for Integration Procedures

The Response Action ends an Integration Procedure and returns data to the entity that called it. It can also add data to the data JSON or end conditionally. Response Actions are useful for debugging Integration Procedures.

Set Values Action for Integration Procedures

The Set Values action sets values in the data JSON of an Integration Procedure using literal values, merge fields, or formulas.

Assert Action for Integration Procedures

The Assert Action compares the expected and actual results of a Test Procedure using an expression that evaluates to true or false. You can use environment variables in the Assert Conditional Formula to test performance.

Property	Description
Assert Conditional Formula	Expression that tests results of previous steps and evaluates to true or false. If the result is false, the assertResult for the Assert Action is set to <code>false</code> . This sets the testResult for the Test Procedure to <code>failed</code> .
Assert Failure Message	Message that explains why the test failed.
Fail Test On Assert	If checked, the Test Procedure stops and returns the result of the transaction if the Assert

Property	Description
	<p>Conditional Formula evaluates to false.</p> <p>If not checked, the Test Procedure continues running even if the Assert Conditional Formula evaluates to false.</p>

See Also

[Test Procedures: Integration Procedures for Unit Testing](#)

Chatter Action for Integration Procedures

The Chatter Action creates a Chatter post and sends it to a Chatter feed.

To obtain the Id for the Mentioned User Id, Subject Id, Image Id, or File Id, go to the object's record page and select the Id from the URL. To find the Id of a community, run a query (for example, `SELECT Id, Name FROM Network`) in the Developer Console.

Property	Description
Community Id	Id of the Community to which to post.
<ul style="list-style-type: none"> • User Id • Mentioned User Id (If you're using the designer on a managed package) 	User Id to mention in the post. Can be a single value or a JSON list.
Markup Type	Leave blank or select Bold , Italic , or Underline to determine the style of the Text property.
Subject Id	(Required) Id of the object that has the Chatter feed. Can be any object that supports Chatter feeds, such as Account, Contact, User, or Group.
Image Id	Id of a File object to be used as an image for the post. Can be a single value or a JSON list.
<ul style="list-style-type: none"> • Uploaded File Id • File Id (If you're using the designer on a 	Id of a File object to be included as an attachment to the post. Can be a single value or a JSON list.

Property	Description
managed package)	
Text	Text of the post. Can include merge fields and HTML markup.

Decision Matrix Action for Integration Procedures

The Decision Matrix Action calls a Decision Matrix with specified inputs and returns the result to the Integration Procedure.

-  **Note** You can ignore a `Calculation Matrix: Not Found` error when exporting an Integration Procedure that calls a Decision Matrix.

Property	Description
<ul style="list-style-type: none"> • Input Parameters: Data Source • Matrix Input Parameters: Data Source (If you're using the designer on a managed package) 	Name of a data JSON node in the Integration Procedure that contains a value to pass to the Decision Matrix
<ul style="list-style-type: none"> • Input Parameters: Filter Value • Matrix Input Parameters: Filter Value (If you're using the designer on a managed package) 	Name of the corresponding Decision Matrix input parameter that accepts the value
<ul style="list-style-type: none"> • Decision Matrix Name • Matrix Name (If you're using the designer on a managed package) 	Name of the Decision Matrix.
Remote Options	<p>Additional options to pass to the Decision Matrix. These options are predefined, but you can also pass options specific to the matrix.</p> <ul style="list-style-type: none"> • <code>executionDateTime</code> – Sets the Effective Date of the Decision Matrix.

Property	Description
Default Matrix Result	Value to return if the Decision Matrix returns null

See Also

[Business Rules Engine](#)[Business Rules Engine](#)[Decision Matrices](#)

Delete Action

Enable users to delete one or more sObject records by using the Delete Action. Use an Object's Record Id to determine which record to delete. Vlocity recommends using a merge field in the Path to Id field that refers to an Id or a list of Ids in the data JSON.

To identify the record's JSON path for the Path to Id field, preview the Omniscript. For example, this data JSON contains a list of Accounts.

```
{ "Account": { "accId": ["001f400000EPQ8o", "001f400000BAkbn"] } }
```

Using his example, the path to delete the array of Account ids is: `%Account:accId%`.

Property	Description
Type	Type of sObject record to delete. For example, Account is a Type of sObject.
Path to Id	Path to the JSON node that contains the Id or list of Ids of the records to delete. Supports merge syntax.
All Or None	When checked, the operation fails if any of the records are not deleted.
Entity Is Deleted Message	Message that displays when a record is deleted.
Delete Failed Message	Message that displays when the action fails to delete a record.
Invalid Id Message	Message that displays when an invalid Id is sent to

Property	Description
	the Delete Action.
Configuration Error Message	Message that displays when the Delete Action has a configuration error.
Confirm	Controls whether the modal displays.
Confirmation Dialog Message	The Confirmation Modal's message text. The default message text is <i>Are you sure? This action cannot be undone.</i>
Confirm Label	Button label for the Confirmation Modal's confirm action.
Cancel Label	Button label for the Confirmation Modal's cancel action.

Docusign Envelope Action for Integration Procedures

The Docusign Envelope Action emails a set of documents for signing.

Property	Description
Docusign Templates	Templates for the documents you want signed. See Prepare a DocuSign Template and Omnistudio Data Mapper Transform for Omniscript Use .
Recipients	After you add a template, specify the Signer Name, Signer Email, and Template Role for each recipient. Signer Name and Signer Email support merge fields. Template roles are configured during template setup.
Email Subject	Subject line for the email requesting signatures.
Email Body	Body text for the email requesting signatures.

Property	Description
Date and Time Formats	Specify the format for display of date and time values. (More information)

See Also

[Integrate DocuSign with Omniscripts](#)

[Use the DocuSign Envelope Action to Email Documents for Signature](#)

[Prepare a DocuSign Template and Omnistudio Data Mapper Transform for Omniscript Use](#)

Email Action for Integration Procedures

An Email Action sends the specified email. You can either specify all the email field values or use a Salesforce email template.

- To use an email template, view it in Email Template Builder and copy the ID from the URL. The ID looks something like this: `00X4N000000aCE5UAM\`
- When using a template, you can specify Contacts as recipients using their IDs.
- To specify email values, deselect **Use Template**.

Property	Available if Use Template Is Checked	Available if Use Template Is Not Checked	Supports Merge Fields	Description
Use Template	Yes	Yes	No	Uses a Salesforce email template if checked. The property determines which other properties are available.
To Email Address List	No	Yes	Yes	Specifies the recipients in the TO field of the email. Click Add Recipient to add each email address. The property accepts a maximum of 100 addresses.
CC Email Address List	No	Yes	Yes	Specifies the recipients in the CC field of the email. Click Add Recipient to add each email address. The property accepts a maximum of 25 addresses.
BCC Email Address List	No	Yes	Yes	Specifies the recipients in the BCC field of the email. Click Add Recipient to add each email address. The

Property	Available if Use Template Is Checked	Available if Use Template Is Not Checked	Supports Merge Fields	Description
				property accepts a maximum of 25 addresses.
Email Subject	No	Yes	Yes	Specifies the subject line of the email.
Email Body	No	Yes	Yes	Specifies the body of the email.
Set HTML Body	No	Yes	No	Determines whether the Email Body is read as plain text (not checked) or HTML (checked).
Org Wide Email Address	Yes	Yes	Yes	Specifies the sender in the FROM field of the email. The address is assumed to represent the entire org. If not specified, the default sender is the user who invoked the Integration Procedure. See Organization-Wide Email Addresses in Salesforce Help.
Select Email Template	Yes	No	No	Specifies the Salesforce email template to use. Enter the sObject Id, then select the template from the dropdown list.
Email Target Object Id	Yes	No	No	Specifies the Id of a Contact, User, or Lead with an email address.
What Id	Yes	No	No	If you specify a Contact in the Email Target Object Id field, the property ensures that merge fields in the template contain the correct data. See the Salesforce Email API documentation for more information.
Save As Activity	Yes	No	No	Saves the email as an activity record on the recipient's page in Salesforce if checked.
Content Versions	Yes	Yes	No	Specifies the Id of a ContentVersion sObject that is included as an attachment.

Property	Available if Use Template Is Checked	Available if Use Template Is Not Checked	Supports Merge Fields	Description
Select Document Attachments	Yes	Yes	No	Specifies the Attachment sObjects to add to the email.
Attachment List	No	Yes	Yes	Specifies a node in the data JSON that contains a list of attachment Ids.

See Also

[Email Templates](#)

Expression Set Action for Integration Procedures

An Expression Set Action invokes the specified Expression Set and returns the results to the Integration Procedure.

-  **Note** You can ignore a `Calculation Procedure: Not Found` error when exporting an Integration Procedure that calls an Expression Set.

Property	Description
Configuration Name	Set to the Expression Set name.
Remote Options	<p>Additional invocation options. The following options are predefined, but you can also pass options specific to the Expression Set.</p> <ul style="list-style-type: none"> • <code>includeInputs</code> – Set to <code>true</code> to include the Expression Set input in the Expression Set output, which is useful for building pricing data. • <code>reserialize</code> – Set to <code>true</code> if an Expression Set doesn't return what you expect.

See Also

[Business Rules Engine](#)

[Expression Sets](#)

HTTP Action for Integration Procedures

The HTTP Action executes a REST call and returns its results to the Integration Procedure. The HTTP Action in Integration Procedures supports only `getBody()` for returning the response. Use Apex directly via a Remote Action to handle binary responses.

Property	Description
<ul style="list-style-type: none"> Request URL HTTP Path (If you're using the designer on a managed package) 	URL to call. For Apex REST actions, you can use merge fields to set this property. You can pass percentage signs in the URL by replacing the percentage sign in the Path with the variable <code>\$Vlocity.Percent</code> .
HTTP Method	GET, POST, PUT, PATCH, or DELETE
Named Credential	<p>Salesforce named credential required for endpoint. We recommend that you use Named Credentials for securely managing authentication data.</p> <p>Starting in Summer '24, Omnistudio supports Salesforce Private Connect. You can use named credentials to make outbound HTTP calls from an Omnistudio org to an external service that runs on AWS. For information on how to set up a named credential with Private Connect, see Establish an Outbound Connection with AWS.</p>
REST Options: Header	<p>HTML header settings specified as key-value pairs. For data security purposes, avoid hardcoding sensitive data in the HTML header.</p> <p>To change the character set for a REST call, set the key to <code>content-type</code> and the value to something like <code>application/json; charset=utf-8</code>. Note that <code>content-type</code> is all lowercase. The default <code>content-type</code> is <code>text/plain; charset=ISO-8859-1</code> if none is set.</p>

Property	Description
REST Options: Params	URL parameters specified as key-value pairs
<ul style="list-style-type: none"> • Request Body • Send Body (If you're using the designer on a managed package) 	Enable a POST action to send Body contents
Timeout	How long in milliseconds to wait for a response
Client Certificate Name	For two-factor authentication, the name of the client certificate to be used.
<p>Debug Logging:</p> <ul style="list-style-type: none"> • Pre-Action Log and Post-Action Log • Pre-action Logging and Post-action Logging (If you're using the designer on a managed package) 	<p>Specifies information to be added to the Preview tab's Debug Output pane before or after the action is attempted. You can log values from PropertySetMap and these merge fields:</p> <ul style="list-style-type: none"> • Pre-Action: <code>%endpoint%</code> and <code>%body%</code> • Post-Action: <code>%stepName + 'Info'%</code> and <code>%stepName + 'Info:Content-Type'%</code> <p>For example:</p> <pre data-bbox="840 1214 1444 1298"><code>%stepName + 'Status'%</code></pre>
Retry Count	Number of times to retry action when it fails
Escape XML Response	Remove XML escapes from an XML response

Integration Procedure Action for Integration Procedures

The Integration Procedure Action runs a subordinate Integration Procedure.

To specify that the subordinate Integration Procedure runs asynchronously, as a [Salesforce future method](#) (which can return no data to the calling Integration Procedure), select **Remote Options** and enter **useFuture:** as a key and **true** as its value. When you run the Integration Procedure, you can see the future call in the debug log. To pass in data as key/value pairs, use the **Additional Input** property.

Property	Description
Integration Procedure	Specifies the Integration Procedure to be run in the format Type_Subtype.
<ul style="list-style-type: none"> • Disable Chaining • Disable Chainable (If you're using the designer on a managed package) 	Disables the Chainable settings of the subordinate Integration Procedure. Doesn't affect the Queueable settings. Unchecked by default.
Remote Options	Specifies additional properties for the Integration Procedure as key-value pairs.
Additional Input	Specifies additional data for the Integration Procedure as key-value pairs.

Intelligence Action for Integration Procedures

The Intelligence Action provides input to and runs an Intelligence Machine on Omnistudio.

Property	Description
Machine Developer Name	Specifies the name of the Intelligence Machine.
Input Data	<p>Specifies values for the Input Parameters of the Intelligence Machine. Typically these are:</p> <ul style="list-style-type: none"> • ContextId – The Id of the person to whom the Intelligence Resources are directed. This is typically a Contact, but it can be any sObject that supports Profile Attributes. • pageSize – The number of Intelligence Resources to present to the person. This is optional. The default is 2.
Items to Rank Path	Specifies the JSON path to the list of Intelligence Resources that the Intelligence Machine can offer the user. This is optional. By default all applicable Intelligence Resources are presented.

List Action for Integration Procedures

The List Action merges multiple lists by matching values of specified list item JSON nodes. A basic merge matches node names exactly. An advanced merge matches nodes that have different names or reside at different levels in the incoming lists.

- Tip** The inputs to a List Action must each be in list format even if the list contains only one item. Omnistudio Data Mapper Extracts that return a list with one item often convert it to a single object. To convert a single object back into a list, you can use the first example in [Omnistudio Data Mapper Transform Examples](#).

For more list processing options, see [Transforming Lists with Data Mappers](#) and the AVG, FILTER, IF, LIST, LISTMERGE, LISTMERGEPRIMARY, LISTSIZE, MAX, MIN, SORTBY, and SUM functions in the [Supported Data Mapper and Integration Procedure Functions](#).

To match nodes that have different names or reside at different levels in the incoming lists, click **Advanced Merge** and specify settings for the nodes to be matched, as follows:

- **List Key:** Enter the name of the JSON list node where the node to be matched resides.
- **Matching Path:** Enter the path within the list to the node to be matched.
- **Matching Group:** Specify the same number for all nodes you want to match.

For example, this figure shows how to match first and last names from two incoming lists in which the nodes have different names.

The screenshot shows the 'Advanced Merge Map' dialog. It has four rows of mappings:

- Row 1: List Key 'DRExtractAddresses:contact' maps to 'firstName' with Matching Path '1'.
- Row 2: List Key 'DRExtractBirthdates:contact' maps to 'FN' with Matching Path '1'.
- Row 3: List Key 'DRExtractAddresses:contact' maps to 'lastName' with Matching Path '2'.
- Row 4: List Key 'DRExtractBirthdates:contact' maps to 'LN' with Matching Path '2'.

Property	Description
<ul style="list-style-type: none"> • Merge Order • Merge Lists Order (If you're using the designer on a managed package) 	Specifies the order in which lists are merged. If a list contains a value for a key that was populated by an earlier list, the later entry overwrites the earlier one.
Merge Fields	The name of the JSON nodes that must match for entries to be merged. To specify nodes below the top level of the JSON structure, use colon-

Property	Description
	<p>delimited paths. By default, the nodes in both lists are assumed to have identical names and reside at the same level. If you don't specify merge fields, the lists are merged into a single list but nodes with matching keys aren't merged. To match a value that resides in a JSON list, append <code> index</code> to its path. For example, to match <code>value1</code> in the <code>list2</code> list, specify <code>list1:node1:node2:list2 1</code> as the merge field.</p> <pre data-bbox="833 720 1445 1241">{ "list1": { "node1": { "node2": { "list2": ["value1", "value2"] } } } }</pre>
Advanced Merge	Matches nodes that have different names or reside at different levels in the incoming lists. See List Action for Integration Procedures .
<ul style="list-style-type: none"> Include NULL as a valid matching value for merging NULL is a Valid Matching Value when Merging (If you're using the designer on a managed package) 	When merge fields are all null, specifies whether to treat them as matching.
<ul style="list-style-type: none"> Don't merge nodes from the same list Prevent Intra List Merge (If you're using the designer on a managed package) 	Specifies whether to merge entries within a list if there are duplicate entries in the list. In this example, <code>List1</code> has fields with the same data

Property	Description
	<p>["Node2": "value2"]. To merge <code>List1</code> and <code>List2</code> with the Merge Fields set to <code>Node2</code>, select Prevent Intra List Merge so that <code>List1</code> entries aren't merged.</p> <pre data-bbox="833 481 1449 1104">{ "List1": [{ "Node1": "value1", "Node2": "value2", "Node3": "value3", "Node8": "Value8" }, { "Node1": "value11", "Node2": "value2", "Node3": "value13" }] }</pre>
	<pre data-bbox="833 1146 1449 1727">{ "List2": [{ "Node2": "value2", "Node4": "value4", "Node5": "value5" }, { "Node2": "value3", "Node4": "value4", "Node5": "value5" }] }</pre> <p>Merged list output when Prevent Intra List Merge isn't selected:</p>

Property	Description
	<pre>[{ "Node5": "value5", "Node4": "value4", "Node8": "Value8", "Node3": "value13", "Node2": "value2", "Node1": "value11" }, { "Node5": "value5", "Node4": "value4", "Node8": "Value8", "Node3": "value13", "Node2": "value2", "Node1": "value11" }]</pre>
	<p>Merged list output when Prevent Intra List Merge is selected:</p> <pre>[{ "Node5": "value5", "Node4": "value4", "Node8": "Value8", "Node3": "value3", "Node2": "value2", "Node1": "value1" }, { "Node5": "value5", "Node4": "value4", "Node3": "value13", "Node2": "value2", "Node1": "value11" }]</pre>

Property	Description				
<ul style="list-style-type: none"> Specify keys to be retained in the output Has Primary (If you're using the designer on a managed package) 	Specifies an allowlist of keys to be retained in the output.				
Primary List Key	(Optional) Allows the list of entries to be retained after the merge is performed. If you specify a primary list, only entries from that list are retained in the output.				
Filter List Formula	Specifies a formula that is run on each node of the list to determine if it remains in the list. If FormulaResult returns TRUE after evaluating a node, the node is retained; otherwise the node is removed. For example, you can specify <code>nodename != ""</code> to remove null values from a list.				
Dynamic Output Fields	By default, all nodes are returned. To return a specified subset of the result nodes, add a node to the input JSON and set its value to a comma-separated list of the desired node names. Don't use spaces in this list. Set this property to the name of the node containing this list of node names.				
Sort By	Specifies one or more keys on which the output list is sorted.				
Update Field Value	<p>Assigns a value or specify a formula that can evaluate and conditionally replace values in the output list. To specify a formula, precede the expression with "=".</p> <p>For example, to replace blank first or last names with "None Specified," add the entries shown in this figure.</p> <div data-bbox="829 1712 1449 1833"> <p>▼ MODIFY LIST</p> <p>Update Field Value</p> <table border="1"> <tr> <td>FirstName</td> <td><input type="text"/> <small>=IF(ISBLANK(FirstName), "None specified", FirstName)</small></td> </tr> <tr> <td>LastName</td> <td><input type="text"/> <small>=IF(ISBLANK(LastName), "None specified", LastName)</small></td> </tr> </table> </div> <p>For more information, see Supported Data Mapper and Integration Procedure Functions.</p>	FirstName	<input type="text"/> <small>=IF(ISBLANK(FirstName), "None specified", FirstName)</small>	LastName	<input type="text"/> <small>=IF(ISBLANK(LastName), "None specified", LastName)</small>
FirstName	<input type="text"/> <small>=IF(ISBLANK(FirstName), "None specified", FirstName)</small>				
LastName	<input type="text"/> <small>=IF(ISBLANK(LastName), "None specified", LastName)</small>				

See Also

[Work with Data and Lists](#)

[Create a List Merge Example with Dynamic Output Fields](#)

Omnistudio Data Mapper Extract Action for Integration Procedures

The Data Mapper Extract Action calls the specified Data Mapper Extract to read data from Salesforce and returns it to the Integration Procedure.

For Integration Procedure examples that include at least one Data Mapper Extract Action, see [Work with Data and Lists](#) and [Handle Errors by Using a Try-Catch Block](#).

The Data Mapper Extract Action doesn't support BLOB data returned from the Data Mapper Extract.

Property	Description
<ul style="list-style-type: none">• Data Mapper Name• Data Mapper Interface (If you're using the designer on a managed package)	Name of Data Mapper Extract
Data Mapper Input Parameters: Data Source	Name of data JSON node containing value to filter on
Data Mapper Input Parameters: Filter Value	Value to match to qualify as an input parameter
Ignore Cache	Disables caching in the Data Mapper Extract.

See Also

[Omnistudio Data Mappers](#)

Omnistudio Data Mapper Post Action for Integration Procedures

The Data Mapper Post Action calls a Data Mapper Load (post) to write data to Salesforce.

For Integration Procedure examples that include at least one Data Mapper Post Action, see [Work with Data and Lists](#) and [Handle Errors by Using a Try-Catch Block](#).

Although a Data Mapper post action primarily creates or updates sObjects, it also produces JSON output, which you can view in the Debug log on the Preview tab. This is important when subsequent Integration Procedure steps must reference the sObjects.

In the JSON response returned by a Data Mapper post, node names are appended with the sequence number of the Data Mapper step that created them. For example:

```
{  
    "Contact_1": [ {  
        "Id": "0036A000002PeaAQAS",  
        "LastName": "Smith",  
        "UpsertSuccess": true  
    } ],  
    "Attachment_2": [ {  
        "Id": "00P6A000000EJ3RUAW",  
        "Name": "angular-route.min.js",  
        "ParentId": "0036A000002PeaAQAS",  
        "UpsertSuccess": true  
    } ]  
}
```

Data Mapper Post Action output, which can be viewed in the Debug log on the Preview tab, allows other steps in the Integration Procedure to access and use the IDs and other details of the processed sObjects.

If the Data Mapper name is CreateContact, you can reference the Id of Contact_1 in the example using the merge field `%CreateContact:Contact_1:Id%`.

Property	Description
<ul style="list-style-type: none">• Data Mapper Name• Data Mapper Interface (If you're using the designer on a managed package)	Name of Data Mapper Post

See Also

[Omnistudio Data Mappers](#)

Omnistudio Data Mapper Transform Action for Integration Procedures

A Data Mapper Transform Action calls the specified Data Mapper Transform to execute transformations on the Data JSON and returns the transformed data.

Property	Description
<ul style="list-style-type: none"> • Data Mapper Name • Data Mapper Interface (If you're using the designer on a managed package) 	Name of Data Mapper Transform
Ignore Cache	Disables caching in the Data Mapper Transform.

See Also[Omnistudio Data Mappers](#)[Omnistudio Data Mapper Output Data Types](#)

Omnistudio Data Mapper Turbo Action for Integration Procedures

A Data Mapper Turbo Action calls the specified Data Mapper Turbo Extract to read data from Salesforce and returns it to the Integration Procedure.

The Data Mapper Turbo Action is exactly the same as the Data Mapper Extract Action in how it works. The only difference is the type of Data Mappers the Data Mapper Interface property lists. For Integration Procedure examples that include at least one Data Mapper Extract Action, see [Work with Data and Lists](#) and [Handle Errors by Using a Try-Catch Block](#).

Property	Description
<ul style="list-style-type: none"> • Data Mapper Name • Data Mapper Interface (If you're using the designer on a managed package) 	Name of Data Mapper Turbo Extract
Data Mapper Input Parameters: Data Source	Name of data JSON node containing value to filter on
Data Mapper Input Parameters: Filter Value	Value to match to qualify as an input parameter
Ignore Cache	Disables caching in the Data Mapper Turbo Extract.

See Also[Omnistudio Data Mappers](#)

Remote Action for Integration Procedures

The Remote Action element calls the specified Apex class and method or the specified invocable action.

Property	Description
Remote Class	<p>Callable class or <code>DefaultInvocableAction</code> for an Invocable Action. In Omnistudio Standard, to prevent an error that says the Apex class can't be loaded, include the namespace: <code>omnistudio.ClassName</code>.</p>
Remote Method	Method or Invocable Action name.
Remote Options	<p>Additional class invocation options. The following options are predefined, but you can also pass options specific to the class.</p> <ul style="list-style-type: none"> • <code>InvocableInputKey</code> -- Specifies a JSON map as Invocable Action input. • <code>reserialize</code> -- If <code>true</code>, reserializes the class output. If a remote action returns data that isn't in <code>Map<String, Object></code> format, sometimes an Integration Procedure or DataRaptor can't process it unless it's reserialized. If the returned data isn't an sObject, you can't use it to update Product2 objects without reserializing. • <code>useStandardRuntime</code> -- If <code>true</code>, forces the Apex class to run in the Omnistudio standard runtime, even if the Managed Package Runtime is enabled in Omnistudio Settings in Setup.
Additional Input	Data passed to the method or Invocable Action.

See Also

[Create a Remote Action Example for an Invocable Action](#)

[Create a Remote Action Example for an Invocable Action with an Input Key](#)

[Create the Integration Procedure with the Remote Action](#)

Response Action for Integration Procedures

The Response Action ends an Integration Procedure and returns data to the entity that called it. It can also add data to the data JSON or end conditionally. Response Actions are useful for debugging Integration Procedures.

Response Actions are also useful for trimming the data response. For specific trimming strategies, see [Manipulate JSON with the Send/Response Transformations Properties](#).

For Integration Procedure examples that include at least one Response Action, see [Work with Data and Lists](#), [Handle Errors by Using a Try-Catch Block](#), and [Integration Procedure Action for Integration Procedures](#).

Property	Description
Additional Output	<p>Key/value pairs to add to data JSON. Can merge other data. To specify a level below the top level for a key/value pair, use a colon-delimited path in the Key field. For example:</p> <p>Key Field: <code>Key1:Key2</code></p> <p>Resulting JSON:</p> <div data-bbox="833 1121 1460 1406" style="border: 1px solid #ccc; padding: 10px;"><pre>{ "Key1": { "Key2": "Value" } }</pre></div>
Return Only Additional Output	Enable to discard data other than the key/value pairs defined as additional output. If you enable this option, disable Return Full Data JSON.
Return Full Data JSON	Enable to return entire data JSON. If you enable this option, disable Return Only Additional Output.
Response Headers	To add data to the response header, add key/value pairs to this property. To override the HTTP status code returned in the header, create a key

Property	Description
	named StatusCode and assign it a valid HTTP response value.
Execution Conditional Formula	Ends the Integration Procedure only if the specified condition is true.

Set Values Action for Integration Procedures

The Set Values action sets values in the data JSON of an Integration Procedure using literal values, merge fields, or formulas.

-  **Note** Any text between two percent (%) signs in a Set Values formula is treated as a [merge field](#). To represent literal percent (%) signs, use the `$VLOCITY.Percent` environment variable. The Set Values action doesn't support custom labels.

Properties

Property	Description
Element Value Map: Element Name	The JSON node for which the value is to be set.
Element Value Map: Value	<p>The value to assign to the JSON node. Options:</p> <ul style="list-style-type: none"> Merge fields from a previous step (%elementName%) Literal value Concatenated values Results of formulas and functions Expressions that combine the options: "Case Status: %caseStatus%"

Examples

For the following examples, consider this sample input JSON.

{

```

    "Name": {
        "FirstName": "John",
        "LastName": "Smith"
    },
    "Finances": {
        "GrossIncome": "100000",
        "Expenses": "60000"
    }
}

```

Rename Nodes in the Input Data

To shorten and simplify the input JSON nodes, set the Element Value Map like this example.

Element Name	Value
FirstName	%Name:FirstName%
LastName	%Name:LastName%
GrossIncome	%Finances:GrossIncome%
Expenses	%Finances:Expenses%

Set Literal Values

To set literal values, which are useful for testing or for specifying data that doesn't change, set the Element Value Map like this example.

Element Name	Value
Location	San Francisco
Code	2345

Concatenate Literal Values and Merge Fields

To concatenate a literal string with a merge field, set the Element Value Map like this example.

Element Name	Value
LocationCode	Code is %LocationData:Code%

The `%LocationData:Code%` merge field retrieves data from the previous component, LocationData.

Define an Array

To set up an array, click **Edit as JSON** and edit the `ElementValueMap` node like this example.

```
"elementValueMap": {
  "Departments": [
    "Sales",
    "Development",
    "Support",
    "Training"
  ]
},
```

Perform Simple Calculations Using Formulas

To perform calculations, use operators in the Element Value Map like these examples.

Element Name	Value
AfterTaxIncome	=%Inputs:GrossIncome% * 0.7

Set the Element Value Map of CalculateNet as follows:

Element Name	Value
NetIncome	=%AfterTax:AfterTaxIncome% - %Inputs:Expenses%

Concatenate Two Merge Fields

To concatenate two merge fields, set the Element Value Map like this example.

Element Name	Value
FullName	=%Inputs:FirstName% + " " + %Inputs:LastName%

Notice the literal space included to separate the first and last names.

Use a Function and Retrieve an Array Value

This example uses the IF function. It also uses listnode | listnumber notation to retrieve array values.

Element Name	Value
Department	=IF(%Inputs:FirstName% = "John",%DeptArray:Departments 2%,%DeptArray:Departments 3%)

This component assigns a Department value of Development to anyone named John and a Department value of Support to anyone not named John. Notice that the second equals sign has spaces before and after it, to distinguish it from the function.

-  **Note** By default, if an IF function output is a large number, it's converted to scientific notation. To prevent this conversion if the large number is an identifier and not a math operand, wrap the IF function in a TOSTRING function.

Perform a Complex Date Calculation

This example uses four nested date functions to return the first day of the current month.

Element Name	Value
FirstOfMonth	=ADDDAY(EOM(ADDMONTH(TODAY(),-1)),1)

Create a Structure for Output Data

This example creates a JSON structure for the transformed data. To set up the structure, click **Edit as JSON** and edit the **ElementValueMap** node like this sample JSON.

```
"elementValueMap": {
    "Output": {
        "FullName": "%ConcatName:FullName%"}
```

```
"Department": "%RetrieveDept:Department%",
"AfterTaxIncome": "%AfterTax:AfterTaxIncome%",
"NetIncome": "%CalculateNet:NetIncome%",
"Location": "%LocationData:Location%",
"LocationCode": "%TextWithCode:LocationCode%",
"FirstOfThisMonth": "%CalcFirstOfThisMonth:FirstOfThisMonth%"

},
},
```

Integration Procedure Blocks

In an Omnistudio Integration Procedure, you can run a group of related steps as a unit inside a block to execute them conditionally, cache them, repeat them for each item in a list, or return an error if they fail.

All blocks have one property in common, which is the Execution Conditional Formula. If this formula evaluates to true or is not defined, the block is executed. If it evaluates to false, the block is skipped.

-  **Note** You can nest blocks within other blocks. For example, you can nest a Loop Block within a Try-Catch Block or a Cache Block.

Cache Block

Using a cache to store frequently accessed, infrequently updated data saves round trips to the database and improves performance. Use Cache Blocks if some parts of the Integration Procedure update data and therefore shouldn't be cached, or if different cached data should expire at different times.

Conditional Block

Use a Conditional Block to group actions and blocks behind a formula without performing a loop or merging a list. The Conditional Block's only job is to control the Integration Procedure flow. A Conditional Block executes in its entirety if an expression is true, executes one of a set of mutually exclusive conditions defined in the steps it contains, or both.

Loop Block

A Loop Block iterates over the items in a data array, enabling the Actions within it to repeat for each item. The array input to the Loop Block is processed so that each iteration receives only one item in the array.

Try-Catch Block

A Try-Catch Block lets you "try" running the steps inside it and then "catch" the error if a step fails.

Cache Block

Using a cache to store frequently accessed, infrequently updated data saves round trips to the database and improves performance. Use Cache Blocks if some parts of the Integration Procedure update data

and therefore shouldn't be cached, or if different cached data should expire at different times.

Property	Description
Salesforce Platform Cache Type	<p>Specifies the cache type:</p> <ul style="list-style-type: none"> • Session Cache – For data related to users and their login sessions • Org Cache – For all other types of data
Time To Live In Minutes	<p>Determines how long data remains in the cache. The minimum value is 5. Default and maximum values depend on the cache type:</p> <ul style="list-style-type: none"> • Session Cache – The default value is 5. The maximum value is 480, equivalent to 8 hours. However, the cache is cleared when the user's session expires. • Org Cache – The default value is 5. The maximum value is 2880, equivalent to 48 hours. <p>Top-level caching overrides Cache Block caching for the duration of the top-level Time To Live In Minutes value.</p>
Cache Keys	<p>Configurable Key/Value pairs to be stored in the cache. Enter a Key and set the Value to the response of an Action within the Cache Block. The value can use merge field syntax, percentage signs on either side of the node name, to access that response. Cache keys are available, or scoped, only within the Cache Block.</p>
Cache Block Output	<p>Configurable Key/Value pairs to be available to subsequent Integration Procedure steps. Enter a Key and set the Value to the response of an Action within the Cache Block. The value can use merge field syntax, percentage signs on either side of the node name, to access that response.</p>
Refresh Cache Conditional Formula	<p>If the formula evaluates to true, forces data to be saved to the cache.</p>

Property	Description
Ignore Cache Conditional Formula	If the formula evaluates to true, neither clears nor saves data to the cache.
Add to Cache Conditional Formula	If the formula evaluates to true, saves data to the cache. If false, does not cache data.
Fail On Step Error	If this box is checked in a step within the Cache Block and that step fails, no data is cached.

Conditional Block

Use a Conditional Block to group actions and blocks behind a formula without performing a loop or merging a list. The Conditional Block's only job is to control the Integration Procedure flow. A Conditional Block executes in its entirety if an expression is true, executes one of a set of mutually exclusive conditions defined in the steps it contains, or both.

Property	Description
Execution Conditional Formula	Specifies that the entire Conditional Block runs only if this formula evaluates to true.
Is If Else Block	Specifies that all actions within the block except optionally the last have mutually exclusive Execution Conditional Formula values. If the last action has a blank Execution Conditional Formula , it runs only if the Execution Conditional Formula values of all the other actions evaluate to false.

Loop Block

A Loop Block iterates over the items in a data array, enabling the Actions within it to repeat for each item. The array input to the Loop Block is processed so that each iteration receives only one item in the array.

Property	Description
Loop List	Accepts a JSON node containing an array.

Property	Description
Loop Output	<p>Configurable Key/Value pairs to be available to subsequent Integration Procedure steps. Enter a Key and set the Value to the response of an Action within the Loop Block. The value can use merge field syntax, percentage signs on either side of the node name, to access that response. By default, if no Key/Value pairs are specified, a response of <code>{ "success": "OK" }</code> is returned for each item processed in the array.</p> <p>Use this property with care. It returns the data you request for every iteration. If the Loop Block iterates 1000 times, it returns 1000 responses. This property is most useful for debugging or returning a limited dataset.</p>
Execution Conditional Formula	Controls whether the Loop Block executes based on an expression that evaluates to true or false.

Try-Catch Block

A Try-Catch Block lets you "try" running the steps inside it and then "catch" the error if a step fails.

Property	Description
Fail on Block Error	Specifies that if the Try-Catch Block fails, the entire Integration Procedure fails.
Failure Response	<p>A response to return if the Try-Catch Block fails. The value can be a formula.</p> <p>A Try-Catch Block can have both a Failure Response and a Custom Failure Response.</p>
Custom Failure Response	A Remote Class and Remote Method of an Apex class to execute if the Try-Catch Block fails. The Apex class must implement VlocityOpenInterface.

Omnistudio

Use Omnistudio Design Assistant to get real-time feedback on component health, design quality, complexity, performance, and best practices during the design stage of your Flexcards and Omniscripts. Use this feedback to create or refine the Omniscript and Flexcard designs, and use the system resources more optimally to prevent issues such as high latency, low throughput, and resource overuse during the design stage, and potential run-time failures.

The visual indicators and notifications that appear on Flexcards and Omniscripts help you identify and resolve design inefficiencies in real time. For example, the recommended number of Action Block elements in an Omniscript is 6, and the maximum recommended number is 10. If you exceed this limit by adding a seventh Action Block element, an informational message icon  appears on the Omniscript designer. Similarly, when you add the 10th Action Block element, a warning message icon  appears on the designer. When you click the icon, it shows the number of informational messages or warning messages, and provides a link to a detailed report in the Design Assistant panel. After reviewing the report, you can take the necessary steps to manage your resources and implement best practices, and keep them within the recommended limit.

For more information about the recommended limits for Flexcards and Omniscripts, see [Recommended Limits](#).

Omnistudio Design Assistant:

- Provides a comprehensive report of the Flexcard and Omniscript design alerts, if any.
- Shows real-time, actionable alerts within the Omnistudio designers through visual indicators to highlight performance, complexity, component health, and potential issues.
- Assesses the complexity of the component by measuring the number of elements, data sources, actions, and other factors. This information helps you to simplify complex components.
- Proactively checks the design for best practices, identifies common issues and deviations, and provides feedback to help you align the design with best practices.

Benefits

Omnistudio Design Assistant significantly enhances the design experience within Omnistudio, and helps you create optimized, high-performing components while adhering to best practices.

Omnistudio Design Assistant offers these benefits.

- Design consistency: Promotes consistent design standards by validating component designs against best practices, promoting uniformity, ensuring the creation of high-quality components, and reducing maintenance overhead.
- Enhanced efficiency: Minimizes time spent on debugging and testing, streamlining the design process.
- Real-time feedback: Provides real-time feedback on component health, performance, and complexity. This immediate insight helps you make informed decisions during the design process, reducing the need for extensive rework.
- Error prevention: Detects when component size limits are likely to exceed, and provides actionable alerts, preventing runtime errors.
- Simplified onboarding: Provides dynamic alerts and recommendations, helping new users quickly learn best practices and align with design standards.
- Best practices suggestions: Helps you adhere to best practices and create efficient, high-quality components while minimizing technical inaccuracies.

Monitor and Manage Your Flexcards and Omniscripts

Use Omnistudio Design Assistant to monitor and proactively manage the overall health of your Flexcards and Omniscripts during the design stage, and implement best practices.

Monitor and Manage Your Flexcards and Omniscripts

Use Omnistudio Design Assistant to monitor and proactively manage the overall health of your Flexcards and Omniscripts during the design stage, and implement best practices.

1. From the App Launcher, find and select **Flexcards**.
2. Select the Flexcard version or Omniscript that you want to monitor and manage.
3. View the comprehensive report of the health of your Flexcards and Omniscripts through one of these options.
 - Click the  or  icon that appears on the Flexcard or Omniscript designers. It shows the number of informational or warning messages. For a detailed report, click the **More Details** link.
 - Click **Omnistudio Design Assistant** on your Flexcard or Omniscript designers.
4. Click the **Information** or **Warning** tab, and expand the message panel.
5. Click **>** on the footer panel that shows {current_count}/{limit_count}.
6. View the detailed report, and take the necessary steps.

Omnistudio

Generate unique numbers for specific use cases and configure your numbering system. Keep numbering consistent across records.

With Omni Global Auto Number, you can create custom numbers that fulfill various use cases. For instance, you can autogenerate numbers for sales orders, cases, loan applications, and so on. You can customize attributes such as the number's increment value, padding, minimum length, prefix, and separator. Learn more about these configurations in Set Up Global Auto Numbers.

Once you set up global auto numbers, you can use them via functions in Data Mappers and Integration Procedures. To learn more, see Using the GLOBALAUTONUMBER Function.

Set Up Omni Global Auto Numbers

Use Omni Global Auto Number to create a custom numbering system that applies to specific information such as versions, years, and product codes.

Set Up Omni Global Auto Numbers

Use Omni Global Auto Number to create a custom numbering system that applies to specific information such as versions, years, and product codes.

To access this feature, make sure you turn on the Omni Global Auto Number Omnistudio Setting and fulfill the prerequisites as described in [Enable Autogenerated Numbers](#).

To set up global auto numbers, perform these tasks. Note that all fields mentioned here are mandatory, except Prefix and Separator.

1. From App Launcher, find and select **Omni Global Auto Number**.
2. Click **New**.
3. In the Name field, enter *GlobalAutoNumber* or a name of your choice.

The value you enter here serves as the unique identifier for the GLOBALAUTONUMBER function in Data Mappers and Integration Procedures. Make sure that each auto number has its own distinct name.

4. Enter an integer in the Increment field. This sets the increment value for Omnistudio's auto-generated numbers. For example, if you enter 5, Omnistudio counts the numbers up by 5 each time it generates a new one.
5. Enter a number in the Last Generated Number field to provide an initial starting number. When you use the GLOBALAUTONUMBER function, the numbers generated will be a continuation of the value entered here. The default value is 0. Every time the GLOBALAUTONUMBER function is executed, this field is updated with the latest value.
6. Enter a digit or letter in the Left Pad field. If a number is shorter than the minimum length specified, Omnistudio pads the number with the digit or letter specified here. For instance, if you enter 0, and the minimum length is 5, the number 123 is saved as 00123.
7. Enter a value for Minimum Length.
8. To add a prefix, enter a value in the Prefix field.
9. To add a separator character between the prefix and the number, enter a value in the Separator field. You can leave the field blank if you don't need it.
10. Save your global auto number.

As an example, if you set the GlobalAutoNumber settings like this:

- Name: GlobalAutoNumber
- Increment: 1
- LastGeneratedNumber: 27
- LeftPad: 0
- Minimum Length: 6
- Prefix: OS
- Separator: -

The next number Omnistudio generates is OS-000028.

You can use the GLOBALAUTONUMBER function in Data Mappers and Integration Procedures.

Omnistudio

Omnistudio now supports automated UI testing for Omniscripts and Flexcards by using the UI Test Automation Model (UTAM) framework. Automated testing is crucial for ensuring the stability and integrity of Omnistudio deployments.

UTAM is a modern, open-source framework that helps you validate the entire customer experience. It validates by simulating UI interaction to test user actions (example, clicks, input), asserting correct element values, verifying expected navigation behaviors (example, moving between steps), and confirming accurate visual states (example, error messages, visibility). This test ensures customer journeys function correctly after deployments or code changes.

UTAM uses the Page Object Model (POM) pattern to abstract away underlying UI changes, creating stable, resilient tests that remain durable across Salesforce releases. This approach provides reliable, end-to-end validation of the customer experience. See [UTAM](#) and [Salesforce Page Objects](#).

Benefits of UTAM Testing

UTAM testing provides several key advantages for quality and stability:

- Durable and reliable UI testing: Your tests remain stable and reliable even when platform updates change the component structure. UTAM achieves this durability through Document Object Model (DOM) abstraction by separating the test logic from the UI code.
- Comprehensive user-journey validation: Tests Omniscripts and Flexcards end-to-end by simulating real user actions (form entries, button clicks, data checks, and conditional steps) and navigation.
- Reusable and maintainable tests: Uses the Page Object Model (POM) to organize UI test code into reusable Page Objects (POs), making tests easier to read, build, and maintain.

Key UTAM Testing Concepts

These concepts explain the underlying architecture that ensures your tests remain stable during any Salesforce UI changes:

- Page Object Model (POM): POM is the main design pattern that UTAM uses. It separates the test logic (what you test) from the UI structure (where the element is present in the code).
- Abstraction and resilience: UTAM achieves test stability by abstracting the underlying DOM structure. Even if Salesforce modifies the HTML or CSS of an Omniscript or a Flexcard element, your test script remains stable and functions as expected.

- **Page Objects (POs):** POs are the central modules that represent specific UI components, such as an `OmniscriptStep` or an input field. The developer calls the stable methods defined in the POs . For example, `setValue()` , `click()` .
- **Test Scripting:** The developer writes code that calls the PO methods in a sequence, followed by an assertion to verify the outcomes. For example, verifying field values or error messages.

UTAM Testing Workflow

The UI Test Automation Model (UTAM) testing workflow focuses on preparing the testing environment, writing UTAM Page Objects (POs), and running end-to-end tests for UI components.

How to Write End-to-End Tests with UTAM for Flexcards

To fully automate and stabilize your guided user flows, structure and build end-to-end tests for Flexcards by using UTAM's commands.

How to Write End-to-End Tests with UTAM for Omniscripts

To fully automate and stabilize your guided user flows, structure and build end-to-end tests for Omniscripts by using UTAM's commands.

UTAM Testing Workflow

The UI Test Automation Model (UTAM) testing workflow focuses on preparing the testing environment, writing UTAM Page Objects (POs), and running end-to-end tests for UI components.

1. Environment setup: Install the Salesforce CLI and configure your preferred test runner (example, WebDriverIO, TestNG, Maven) to work with the UTAM framework. See [Install Salesforce CLI](#), [Guide for Java](#), and [Guide for JavaScript](#).
2. PO creation: Create custom UTAM POs for your complex Omniscripts and Flexcards. These POs define the UI elements (inputs, buttons, steps) and the actions you can perform on them. See [Salesforce Page Objects](#).
3. Test scripting: Write test classes in Java or TypeScript that import the necessary Omnistudio and custom POs, then define a logical sequence of user actions that include login, navigate, data input, and component interaction.
4. Assertion: Use assertion libraries (example, TestNG Assert) to confirm that elements display correctly, input values are retained, and UI logic (example, error messages) works as expected.
5. End-to-end tests: For commands to run your tests, see [UTAM Java Readme](#) and [UTAM JavaScript Readme](#). Example, `mvn test -Dtest=<TestClassName>`.

How to Write End-to-End Tests with UTAM for Flexcards

To fully automate and stabilize your guided user flows, structure and build end-to-end tests for Flexcards by using UTAM's commands.

Sample UTAM Methods for Flexcards

These methods are critical parts of the Omniscript Page Objects (POs). Use them to replicate essential user actions.

Method	Component	Purpose	Example Action
<code>waitForVisible()</code>	General PO	Ensures the UI component has finished loading and is ready for interaction.	Waits for the spinner to disappear.
<code>getFlexCardStates()</code>	Flexcard	Retrieves a list of the states in the flexcard.	Finds the first state in the flexcard.
<code>getElements()</code>	FlexCardState	Retrieves the list of all elements in a flexcard state.	Finds the number of elements in a flexcard state.

Method	Component	Purpose	Example Action
<code>isValid()</code> / <code>getErrorMessage()</code>	Input POs	Checks if the element currently passes validation rules.	Confirms a required field shows: <code>Error: Field is required.</code>

Write End-to-End Tests for Flexcards with UTAM

Use these steps to structure your UTAM test class, set up your environment, and run the main logic required for an end-to-end Flexcard validation. This example uses [Maven](#).

- Note** Before you start writing end-to-end tests for Flexcards using UTAM, make sure you have a good understanding of [UTAM](#).

- **1.** Configure the initial setup. See [UTAM](#), [Java](#), and [JavaScript](#).
- **2.** Setup and navigation: Before writing your test logic, make sure that your environment is set up and your test class is configured.
 - - Helper methods: It's highly recommended to abstract login, navigation, and environment configuration into reusable helper methods defined in a base test class (example, `SalesforceWebTestBase`). This keeps your test logic clean and maintainable.
 - - Test setup: Use the `@BeforeTest` annotation to define the setup logic, including browser initialization and logging into the target Salesforce org.

```
// Handles browser setup and login before tests run
@BeforeTest
public void setup() {
    setupChrome();
    login(testEnvironment, "home");
}
```

- **3.** Test logic: The actual test method uses the `@Test` annotation (example, for Maven) and orchestrates the user interaction sequence.
 - a. Load and wait for visibility: After retrieving the Flexcard PO, always wait until it is visible before interacting with elements. This makes sure that the UI and underlying LWC components are rendered.

Example:

```
FlexCard flexcard = goDirectlyToFlexcard();
flexcard.waitForVisible();
```

- **b.** Retrieve card state and elements: Each Flexcard contains one or more `FlexcardState` objects. Access the required state and retrieve its list of elements:

Example:

```
FlexCardState state = flexcard.getFlexCardStates().get(0);
```

```
List<Element> elements = state.getElements();
```

C. Testing element validation:

- a. To read labels or values displayed by the Flexcard, access the nested `OutputField` PO:

Example:

```
String label = elements.get(0).getOutputField().getLabel().getText();
```

- b. Assert that the label matches the expected value::

Example:

```
Assert.assertEquals(label, "Result", "The label is not correct");
```

Sample Flexcard Test (Java)

This example demonstrates the complete end-to-end user journey, including setup, navigation, and interaction with Flexcards.

Sample Flexcard Test (Java)

This example demonstrates the complete end-to-end user journey, including setup, navigation, and interaction with Flexcards.



Example

```
package utam.examples.salesforce.web;

import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;
import org.testng.Assert;
import utam.utils.salesforce.TestEnvironment;
import utam.omnistudio.flexcard.pageobjects.FlexCardState;
import utam.omnistudio.core.pageobjects.FlexCard;
import utam.omnistudio.flexcard.pageobjects.Element;
import java.util.List;
import utam.flexipage.pageobjects.RecordHomeTemplateDesktop2;
import utam.global.pageobjects.RecordHomeFlexipage2;
import utam.omnistudio.flexcard.pageobjects.OutputField;

public class FlexcardTest extends SalesforceWebTestBase {

    private final TestEnvironment testEnvironment = getTestEnvironment("sa
ndbox44");
}
```

```
@BeforeTest
public void setup() {
    setupChrome();
    login(testEnvironment, "home");
}

private FlexCard goDirectlyToFlexcard() {
    getDriver().get(YOUR_FLEXCARD_EMBEDDED_RECORD_PAGE_URL);
    RecordHomeFlexipage2 recordHome = from(RecordHomeFlexipage2.clas
s);
    FlexCard flexcard = recordHome.getDecorator().getEventBroker()
        .getGeneratedTemplate(RecordHomeTemplateDesktop2.class)
        .getComponent2("runtime_omnistudio_flexcard")
        .getContent(FlexCard.class);

    return flexcard;
}

@Test
public void testFlexcard() throws InterruptedException {
    FlexCard flexcard = goDirectlyToFlexcard();
    flexcard.waitForVisible();
    FlexCardState state = flexcard.getFlexCardStates().get(0);
    List<Element> elements = state.getElements();
    String label = elements.get(0).getOutputField().getLabel().getTex
t();
    Assert.assertEquals(label, "Result", "The label is not correct");
}

@AfterTest
public void tearDown() {
    quitDriver();
}
```

How to Write End-to-End Tests with UTAM for Omniscripts

To fully automate and stabilize your guided user flows, structure and build end-to-end tests for Omniscripts by using UTAM's commands.

Sample UTAM Methods for Omniscripts

These methods are critical parts of the Omniscript Page Objects (POs). Use them to replicate essential user actions, from loading the component to navigating between steps and invoking actions in your automated tests.

Method	Component	Purpose	Example Action
<code>waitForVisible()</code>	General PO	Ensures the UI component finished loading and is ready for interaction.	Waits for the spinner to disappear.
<code>getSteps()</code>	Omniscript	Retrieves a list of the flow's steps so you can target a specific one (example, index 0).	Finds the first step in the flow.
<code>getElementWithName()</code>	OmniscriptStep	Retrieves a specific element PO (example, a text input) using the element's Name (API Name).	Targets the Username input field.
<code>getElements</code>	OmniscriptStep	Retrieves a list of all elements of a particular type from a given omniscript step.	Targets all text elements in a step.
<code>setValue(value)</code>	Input POs	Simulates a user typing input into a field.	<code>username.setValue("test data");</code>
<code>getNextButton().click()</code>	Omniscript	Simulates the user clicking the navigation button.	Moves the flow from Step 1 to Step 2.
<code>isValid() / getErrorMessage()</code>	Input POs	Checks if the element currently passes validation rules.	Indicates that a field is required: <code>Error: Field is required.</code>
<code>invokeAction()</code>	Action POs	Triggers actions linked to the component, such as Remote Actions.	Executes a button-linked server-side process.

Write End-to-End Tests for Omniscripts with UTAM

Use these steps to structure your UTAM test class, set up your environment, and run the main logic required for an end-to-end Omniscript validation. This example uses [Maven](#).

-  **Note** Before you start writing end-to-end tests for Omniscripts using UTAM, make sure you have a good understanding of [UTAM](#).

- **1.** Configure the initial setup. See [UTAM](#), [Java](#), and [JavaScript](#).
- **2.** Setup and navigation: Before writing your test logic, make sure that your environment is set up and your test class is configured.
 - - Helper methods: It's highly recommended to abstract login, navigation, and environment configuration into reusable helper methods defined in a base test class (example, `SalesforceWebTestBase`). This keeps your test logic clean and maintainable.
 - - Test setup: Use the `@BeforeTest` annotation to define the setup logic, including browser initialization and logging into the target Salesforce org.

```
// Handles browser setup and login before tests run
@BeforeTest
public void setup() {
    setupChrome();
    login(testEnvironment, "home");
}
```

- - Omniscript navigation: Use a helper method (example, `goDirectlyToOmniscript`) to navigate to the Omniscript's specific App URL, returning the main Omniscript Page Object (PO).
- **3.** Test logic: The actual test method uses the `@Test` annotation (example, for Maven) and orchestrates the user interaction sequence.
 - a. Load and navigate to the first step: Make sure that the test successfully retrieves the Omniscript PO and waits until the UI is ready.
 - a. Retrieve PO: Call your helper method to launch the Omniscript and assign it to the main Omniscript PO.
Example: `Omniscript omniscript = goDirectlyToOmniscript("OmniScript", "Test", "English", "lightning");`
 - b. Wait for load: Use the `omniscript.load()` and `omniscript.waitForVisible()` methods to wait for the UI spinner to disappear and the component to be ready for interaction.
 - c. Get step PO: Retrieve the PO for the first step in the flow. The `getSteps()` method returns a list of steps, where the first step is at index 0.
Example: `OmniscriptStep step1 = omniscript.getSteps().get(0);`
 - d. Wait for step: Make sure that the step is fully visible before attempting to interact with elements.
Example: `step1.waitForVisible();`
 - b. Interact with elements and assert input: To interact with an element, you must retrieve its specific PO using its Element name (not the Label).
 - a. Get element PO: Retrieve the PO for an element inside the step using `getElementWithName()`, passing the element's name and its specific PO class.
Example: `OmniscriptText username = step1.getElementWithName("Username", OmniscriptText.class);`
 - b. Set Value: Input the test data using the `setValue()` method.

Example: `username.setValue("omnistudio");`

- C. Assert Input: Verify that the element correctly holds the value you set.

Example: `Assert.assertEquals(username.getValue(), "omnistudio", "The value of Username is not correct");`

- C. Advance flow and test actions for Omniscripts: Test the primary actions of the guided flow, such as navigation and Remote Actions.

- a. Navigate next: Click the Next button to advance the flow or trigger validation checks.

Example: `omniscipt.getNextButton().click();`

- b. Test Remote Action: Retrieve the PO for an action element and use the `invokeAction()` method to test the functionality.

Example: `OmnisciptRemoteAction submit =
step2.getElementWithName("Submit", OmniscriptRemoteAction.class);
submit.invokeAction();`

- 4. Testing element validation: Testing validation logic is critical for ensuring data quality within the Omniscript. This step also ensures that required fields show an error when empty and allow navigation only when valid.

- a. Leave the required field empty.

- b. Click the **Next** button or trigger the validation.

- c. Assert that the element is not valid using `assertFalse("Should show error after navigation attempt", element.isValid());`

- d. Assert that the expected error message is displayed using `assertEquals(expectedMessage, element.getErrorMessage());`

- e. Fill the element and assert that the element is now valid using `assertTrue("Should be valid after setting value", element.isValid());`

Sample Omniscript Test (Java)

This example demonstrates the complete end-to-end user journey, including setup, navigation, interaction, and assertion across two steps.

Sample Omniscript Test (Java)

This example demonstrates the complete end-to-end user journey, including setup, navigation, interaction, and assertion across two steps.



Example

```
package utam.examples.salesforce.web;

import java.net.URLEncoder;
import java.nio.charset.StandardCharsets;
import org.testng.Assert;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
```

```
import org.testng.annotations.Test;
import utam.utils.salesforce.TestEnvironment;
import utam.omnistudio.core.pageobjects.Omniscript;
import utam.omnistudio.omniscript.pageobjects.OmniscriptDate;
import utam.omnistudio.omniscript.pageobjects.OmniscriptPassword;
import utam.omnistudio.omniscript.pageobjects.OmniscriptRemoteAction;
import utam.omnistudio.omniscript.pageobjects.OmniscriptStep;
import utam.omnistudio.omniscript.pageobjects.OmniscriptText;

public class OmniscriptsTests extends SalesforceWebTestBase {

    private final TestEnvironment testEnvironment = getTestEnvironment("sandbox44");

    @BeforeTest
    public void setup() {
        setupChrome();
        login(testEnvironment, "home");
    }

    protected String createOmniscriptAppUrl(String type, String subType, String language, String theme) {
        String baseOrigin = extractOrigin(testEnvironment.getRedirectUrl());
        if (baseOrigin == null || baseOrigin.isEmpty()) {
            String currentUrl = getDomDocument().getUrl();
            baseOrigin = extractOrigin(currentUrl);
        }
        String query = "lightning/page/omnistudio/omniscript"
                + "?omniscript__type=" + urlEncode(type)
                + "&omniscript__subType=" + urlEncode(subType)
                + "&omniscript__language=" + urlEncode(language)
                + "&omniscript__theme=" + urlEncode(theme)
                + "&omniscript__tabIcon=custom%3Acustom18"
                + "&omniscript__tabLabel=a";
        return baseOrigin + query;
    }

    private static String extractOrigin(String urlStr) {
        try {
            if (urlStr == null || urlStr.isEmpty()) return null;
            java.net.URL url = new java.net.URL(urlStr);
            String portPart = url.getPort() == -1 ? "" : ":" + url.getPort();
            return url.getProtocol() + "://" + url.getHost() + portPart + "/";
        }
    }
}
```

```
        } catch (Exception e) {
            return null;
        }
    }

    private static String urlEncode(String value) {
        return URLEncoder.encode(value == null ? "" : value, StandardCharsets.UTF_8);
    }

    protected void navigateToPage(String url) {
        log("Navigate to URL: " + url);
        getDriver().get(url);
    }

    protected Omniscript getOmniscript() {
        return from(Omniscript.class);
    }

    protected Omniscript goDirectlyToOmniscript(
        String type, String subType, String language, String theme) {
        try {
            log("Navigate to Omniscript App URL: " + createOmniscriptAppUrl(type,
e, subType, language, theme));
            navigateToPage(createOmniscriptAppUrl(type, subType, language, theme));
            return getOmniscript();
        } catch (Exception error) {
            throw new TestInfrastructureException("Could Not Load Omniscript: " +
error.getMessage(), error);
        }
    }

    @Test
    public void testNavigateToOmniscript() throws InterruptedException {
        // Example values; update to valid Omniscript in the target org
        Omniscript omniscript = goDirectlyToOmniscript("OmniScript", "Test",
"English", "lightning");
        omniscript.load();
        omniscript.waitForVisible();

        OmniscriptStep step1 = omniscript.getSteps().get(0);
        step1.waitForVisible();
```

```
OmniscriptText username = step1.getElementWithName("Username", OmniscriptText.class);
OmniscriptPassword password = step1.getElementWithName("Password", OmniscriptPassword.class);

username.setValue("omnistudio");
password.setValue("omniscritps");

Assert.assertEquals(username.getValue(), "omnistudio", "The value of Username is not correct");
Assert.assertEquals(password.getValue(), "omniscritps", "The value of Password is not correct");

omniscritp.getNextButton().click();

OmniscriptStep step2 = omniscritp.getSteps().get(1);
step2.waitForVisible();

OmniscriptDate dateOfBirth = step2.getElementWithName("DateOfBirth", OmniscriptDate.class);
dateOfBirth.setValue("01-01-1990");
Assert.assertEquals(dateOfBirth.getValue(), "01-01-1990", "The value of DateOfBirth is not correct");

OmniscriptRemoteAction submit = step2.getElementWithName("Submit", OmniscriptRemoteAction.class);
submit.invokeAction();
}

@AfterTest
public final void tearDown() {
    quitDriver();
}

public static class TestInfrastructureException extends RuntimeException
{
    public TestInfrastructureException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Omnistudio

Get instant product assistance, solution build plans, and troubleshooting help with the Omnistudio Assistance AI pilot agent.

Agentforce Topics for Omnistudio (Pilot)

Whether you're a designer, developer, or implementation specialist, the agent provides instant, relevant information about any Omnistudio component or element you're working with. It can also provide a complete build plan that helps design end-to-end solutions, based on an industry vertical. For Omniscripts, you can also run diagnostics that help you discover any issues that might lead to runtime errors.

Agentforce Topics for Omnistudio (Pilot)

Whether you're a designer, developer, or implementation specialist, the agent provides instant, relevant information about any Omnistudio component or element you're working with. It can also provide a complete build plan that helps design end-to-end solutions, based on an industry vertical. For Omniscripts, you can also run diagnostics that help you discover any issues that might lead to runtime errors.

Agent Topic: Data Mapper Information (Pilot)

Data Mapper designers can use the Data Mapper Information topic for getting information about Data Mappers and the integration among Data Mappers, Integration Procedures, Omniscripts, and Flexcards.

Agent Topic: Integration Procedure Information (Pilot)

Use the Integration Procedure Information topic to get answers to questions about Integration Procedures, their elements, and usage.

Agent Topic: Flexcard Information (Pilot)

Flexcard designers can use the Flexcard Information topic for getting information about Flexcards and the integration among Data Mappers, Integration Procedures, Omniscripts, and Flexcards.

Agent Topic: Omniscript Information (Pilot)

Omniscript designers can use the Omniscript Information topic for getting information about Omniscripts and the integration among Data Mappers, Integration Procedures, Omniscripts, and Flexcards.

Agent Topic: Product Strategy (Pilot)

Implementation experts, solution architects, and developers can use the Product Strategy topic to design a reusable architecture, a build plan, or a planning guide for various business processes such as vehicle registration, service appointment booking, warranty activation, recall management, work order creation .

Agent Topic: Diagnostics (Pilot)

The Diagnostics topic offers Omnistudio component designers valuable insights for troubleshooting. They can use it during the design phase to identify potential issues and error scenarios that can occur during runtime.

Agent Topic: Data Mapper Information (Pilot)

Data Mapper designers can use the Data Mapper Information topic for getting information about Data Mappers and the integration among Data Mappers, Integration Procedures, Omniscripts, and Flexcards.

Topic Details

API Name	DataMapperInformation
----------	-----------------------

Included Agent Actions	Answer Questions with Salesforce Documentation (Beta)
------------------------	---

Example

A Data Mapper designer wants to understand when to use a Data Mapper Load vs. Data Mapper Transform. They launch Agentforce and enter this utterance: I have an Omniscript in which the user enters some data. I want to design a Data Mapper that formats this data into a specific JSON format. Should I use a Data Mapper Load or Transform?

The agent generates the answer with the specified details. It also links to relevant topics in Salesforce Help.

Agent Topic: Integration Procedure Information (Pilot)

Use the Integration Procedure Information topic to get answers to questions about Integration Procedures, their elements, and usage.

Topic Details

API Name	IntegProcedureInformation
Included Agent Actions	Answer Questions with Salesforce Documentation (Beta)

Example

An Integration Procedure designer wants to understand how to invoke an Integration Procedure from Salesforce Flow. They launch Agentforce and enter this utterance: Can I invoke an Integration Procedure from Salesforce Flow? If so, how do I do that?

The agent generates the answer with the specified details. It also links to relevant topics in Salesforce Help.

Agent Topic: Flexcard Information (Pilot)

Flexcard designers can use the Flexcard Information topic for getting information about Flexcards and the integration among Data Mappers, Integration Procedures, Omniscripts, and Flexcards.

Topic Details

API Name	FlexcardInformation
----------	---------------------

Included Agent Actions	Answer Questions with Salesforce Documentation (Beta)
------------------------	---

Example

A Flexcard designer wants to understand how they can launch an Omniscript from a Flexcard. They launch Agentforce and enter this utterance: How to launch an Omniscript from a Flexcard?

The agent generates the answer with the specified details. It also links to relevant topics in Salesforce Help.

Agent Topic: Omniscript Information (Pilot)

Omniscript designers can use the Omniscript Information topic for getting information about Omniscripts and the integration among Data Mappers, Integration Procedures, Omniscripts, and Flexcards.

Topic Details

API Name	OmniscriptInformation
Included Agent Actions	Answer Questions with Salesforce Documentation (Beta)

Example

An Omniscript designer wants to understand whether they need to use an edit block or a block for their specific use case. They launch Agentforce and enter this utterance: I'm trying to structure information in an Omniscript and want to link to SObjects directly. Should I use a block or an edit block?

The agent generates the answer with the specified details. It also links to relevant topics in Salesforce Help.

Agent Topic: Product Strategy (Pilot)

Implementation experts, solution architects, and developers can use the Product Strategy topic to design a reusable architecture, a build plan, or a planning guide for various business processes such as vehicle registration, service appointment booking, warranty activation, recall management, work order creation .

Topic Details

API Name	ProductStrategy
Included Agent Actions	<ul style="list-style-type: none"> • Answer Questions with Salesforce Documentation (Beta) • Get Org Objects (Pilot)

Example

An implementation expert in the education industry vertical wants to understand how to structure a website they're building. They launch Agentforce and enter this utterance: I want to design a form that students can use to submit assignments. How do I use Omnistudio to build this?

The agent asks you to specify an industry vertical. After it receives that information, it generates a complete build plan. You can ask the agent to clarify and change components of the plan that you aren't happy with.

Agent Topic: Diagnostics (Pilot)

The Diagnostics topic offers Omnistudio component designers valuable insights for troubleshooting. They can use it during the design phase to identify potential issues and error scenarios that can occur during runtime.

Topic Details

API Name	Diagnostics
Included Agent Actions	Diagnose Components

Example

An Omniscript designer wants to troubleshoot their Omniscript to check for potential issues. They launch Agentforce and enter this utterance: Diagnose my Omniscript with the ID 00000000111111.

The agent diagnoses the Omniscript and provides a list of potential errors and warnings. You can then fix these errors and re-design your Omniscript to fix any warnings.

Omnistudio

Use operators and functions to create formulas for Omnistudio Data Mappers and Integration Procedures.

Use data types and operators to represent values and simple expressions. Include functions to perform conditional operations and to evaluate and manipulate numbers, dates and times, strings, lists and arrays, and JSON objects.

Supported Data Mapper and Integration Procedure Data Types

Understand data types to determine the kinds of values you can use with operators and pass to functions. Data types also define the kinds of values that functions can return. Topics related to data types include constants that have special meaning and error cases that you can encounter.

Supported Data Mapper and Integration Procedure Operators

Use operators to develop more complex formulas for Omnistudio Data Mappers and Integration Procedures. For example, operators can perform logical, comparison, or mathematical operations on values in expressions. Operators are evaluated according to a strict precedence.

Supported Data Mapper and Integration Procedure Functions

Use functions to evaluate and manipulate data in formulas for Omnistudio Data Mappers and Integration Procedures. For example, functions can perform conditional operations that take different actions depending on the values you pass them. They can invoke queries and other functions, and they can operate on numbers, dates and times, strings, lists and arrays, and JSON objects.

Sample Apex Code for Custom Functions

Develop Apex classes to implement custom functions for use with Omnistudio Data Mappers and Integration Procedures. A sample Apex class implements three methods that can be called as custom functions in Omnistudio formulas. The class includes code that passes a list to a custom function.

Supported Data Mapper and Integration Procedure Data Types

Understand data types to determine the kinds of values you can use with operators and pass to functions. Data types also define the kinds of values that functions can return. Topics related to data types include constants that have special meaning and error cases that you can encounter.

-  **Note** These data types, constants, and error cases apply to formulas for Omnistudio Data Mappers and Integration Procedures. For information about data types for Omniscript formulas and aggregates, see [Create a Formula or Aggregate in an Omniscript](#).

Omnistudio Data Types

Omnistudio data types are used to indicate the kinds of values to be evaluated or manipulated by Data Mapper and Integration Procedure formulas. Data types exist for numbers, strings, dates and times, lists and arrays, and JSON objects.

Omnistudio Constants

Omnistudio constants are keywords that have specific values. They are reserved words that you can use only to indicate the values they represent in Data Mapper and Integration Procedure formulas.

Omnistudio Error Cases

Omnistudio functions and operators return errors when they can't perform a requested operation in a Data Mapper or Integration Procedure formula. Error cases can occur for missing or invalid parameters, or for values or expressions that cannot be evaluated in the context in which they're used. Error cases can also occur for invalid operators or functions.

Omnistudio Data Types

Omnistudio data types are used to indicate the kinds of values to be evaluated or manipulated by Data Mapper and Integration Procedure formulas. Data types exist for numbers, strings, dates and times, lists and arrays, and JSON objects.

The Omnistudio engine is loosely typed. The engine performs type coercion when the resulting data type is clear. For example, if you concatenate a string and a number, the engine converts the number to a string. These formulas return "abc 123".

- `CONCAT("abc", ' ', 123)`
- `"abc" + ' ' + 123`

However, you can't use other mathematical operators with a combination of alphabetic and numeric operands. These formulas return the error response {}.

- `"abc" - 123`
- `"abc" * 123`
- `"abc" / 123`

Some data types encapsulate subtypes. The documentation refers to the subtypes for clarity when describing parameters and return values.

Data Type	Examples	Description
Boolean	<code>true</code> <code>false</code>	True and false values for use in logical expressions. For more information about Boolean values, see Omnistudio Constants .
Number	1 1.50 -1 -1.50	Numeric values for use in mathematical operations. Some values are of a specific subtype. <ul style="list-style-type: none"> Integers represent whole numbers, for example, <code>1</code> and <code>10000</code>. FLOATS represent decimal values with limited size and precision, for example, <code>12345.67</code> and <code>3.141592</code>. Doubles represent decimal values with twice the size and precision of floats, for example, <code>1234567890.1234</code> and <code>3.14159265358979</code>.
Datetime	"2002-02-15 16:35:30" "02/15/2002 16:35:30" "2002-02-15T16:35:30-0500" " "2/15/ 2002T16:35:30-0500" "2002-02-15T16:00:00 -0800" "02/15/2002T16:00:00 -0800" "2002-02-15T00:00:00.000Z" " "2/15/ 2002T00:00:00.000Z"	Date and time values for use in temporal operations. Some values have a specific subtype. <ul style="list-style-type: none"> Dates include only the date value from a datetime, for example, <code>2002-02-01</code> (<code>YYYY-MM-DD</code> format) and <code>02/01/2000</code> (<code>MM/DD/YYYY</code> format). Times include only the time value from a date time, for example, <code>16:35:30</code>, <code>16:35:30:500</code>, <code>T16:35:00</code>, and <code>T16:35:00:500</code>. In all cases, you can specify input as a full date and time that conforms to the Day.js format . Functions extract only the data they need from the format. Date and time values, like strings, must be enclosed in single or double quotes.

Data Type	Examples	Description
String	<p>"abc, DEF: 123." 'abc, DEF: 123.'</p>	<p>Text values that represent sequences of characters, words, numbers, or dates. Strings are sorted in lexicographical order by comparing the values of their individual characters. You must enclose strings in single or double quotes. You can nest single and double quotes so long as the pairs of nested quotes match, for example, "a 'b' c".</p> <p>Comparisons with operators and most functions are case-insensitive. Comparisons with string functions are case-sensitive.</p>
List	<p><code>1, 2, 3</code> <code>"a", "b", "c"</code> <code>(1, 2, 3)</code> <code>("a", "b", "c")</code></p>	<p>A comma-separated series of values often enclosed in <code>()</code> (parentheses). The values of a list can have any data type and can include nested lists. Lists are anonymous (unnamed).</p> <ul style="list-style-type: none"> • A lists of integers: <code>1, 2, 3</code> or <code>(1, 2, 3)</code> • A lists of strings: <code>"a", "b", "c"</code> or <code>("a", "b", "c")</code>
Array	<p><code>[1, 2, 3]</code> <code>["a", "b", "c"]</code></p> <div data-bbox="319 1341 744 1552"> <pre>[{ "Item": 1 }, { "Item": 2 }]</pre> </div> <div data-bbox="319 1573 744 1848"> <pre>{ "ItemList": [{ "Item": 1 }, { "Item": 2 }] }</pre> </div>	<p>A comma-separated series of values enclosed in <code>[]</code> (square brackets). The values of an array can have any data type and can include nested arrays. Arrays can be named or anonymous (unnamed). An anonymous array is referred to as a JSON list.</p> <ul style="list-style-type: none"> • An anonymous array (JSON list) of strings: <code>["a", "b", "c"]</code> • A named array of integers: <code>"IntegerList": [1, 2, 3]</code> • An anonymous array (JSON list) of JSON objects: <div data-bbox="793 1657 1455 1848"> <pre>[{ "Item": 1 }, { "Item": 2 }]</pre> </div> <ul style="list-style-type: none"> • A named array of JSON objects:

Data Type	Examples	Description
		<pre>{ "ItemList": [{ "Item": 1 }, { "Item": 2 }] }</pre>
JSON object	<pre>{ "Name": "Mike Smith" }</pre> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>"Contact": { "FirstName": "Mike", "LastName": "Smith" }</pre> </div> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>"Cases": [{ "Case1": { "CreatedDate": "...", "LastUpdate": ... }, { "Case2": { ... } }]</pre> </div>	<p>A logical structure that encapsulates information in the form of one or more key-value pairs. Objects can be nested and they can be grouped into arrays. Objects and key-value pairs can be referred to as JSON nodes. A JSON string is a serialized form of a JSON object.</p> <ul style="list-style-type: none"> A simple JSON object that contains two key-value pairs, both of which contain strings. <pre>"Contact": { "FirstName": "Mike", "LastName": "Smith" }</pre> <ul style="list-style-type: none"> A named JSON array that contains an object which itself contains an array of objects. Each object in the nested array is named and contains two key-value pairs, both of which represent datetime strings. <pre>"Account": [{ "Cases": [{ "Case1": { "CreatedDate": "2/1/2024T16:3 5:30 GMT -0500 (EDT)", "LastUpdate": "2/8/2024T09:1 5:00 GMT -0500 (EDT)" } }, { ... }] }</pre>

Data Type	Examples	Description
		<pre data-bbox="784 318 1437 741"> "Case2": { "CreatedDate": "2/2/2024T11:0 5:05 GMT -0500 (EDT)", "LastUpdate": "2/3/2024T15:5 0:57 GMT -0500 (EDT)" } }] }]</pre> <p>You can refer to a JSON object by name or by a path of objects and keys. In both cases, you can use a merge field to refer to the object. These values refer to the <code>Contact</code> object and its <code>FirstName</code> field.</p> <ul style="list-style-type: none"> • <code>Contact</code> and <code>%Contact%</code> refer to the entire object. • <code>Contact:FirstName</code> and <code>%Contact:FirstName%</code> refer to the value of the <code>FirstName</code> key for the object. <p>These values refer to the <code>Account</code> object and its fields.</p> <ul style="list-style-type: none"> • <code>Account:Cases</code> and <code>%Account:Cases%</code> refer to the <code>Cases</code> array. • <code>Account:Cases:Case1</code> and <code>%Account:Cases:Case1%</code> refer to the first object of the <code>Cases</code> array. • <code>Account:Cases:Case1:CreatedDate</code> and <code>%Account:Cases:Case1:CreatedDate%</code> refer to the value of the <code>CreatedDate</code> key of the first object of the <code>Cases</code> array.

Omnistudio Constants

Omnistudio constants are keywords that have specific values. They are reserved words that you can use only to indicate the values they represent in Data Mapper and Integration Procedure formulas.

Constant	Examples	Description
TRUE True true	<code>1 = 1</code> <code>2 > 1</code> <code>"abc" == "abc"</code>	A Boolean constant that indicates a value or expression that is true. All of the example expressions are true. Other values are equivalent to true. Each of these IF functions returns <code>"trueResult"</code> . <ul style="list-style-type: none"> • <code>IF((1), "trueResult", "falseResult")</code> • <code>IF((1.5), "trueResult", "falseResult")</code> • <code>IF(("abc"), "trueResult", "falseResult")</code> • <code>IF(('abc'), "trueResult", "falseResult")</code>
FALSE False false	<code>2 = 1</code> <code>2 < 1</code> <code>"abc" == "def"</code>	A Boolean constant that indicates a value or expression that is false. All of the example expressions are false.
NULL Null null	<code>IF((NULL), "trueResult", "falseResult")</code> <code>IF((Null), "trueResult", "falseResult")</code> <code>IF((null), "trueResult", "falseResult")</code>	A constant that indicates an undefined value. Null values resolve to false. All of the example IF functions return <code>"falseResult"</code> . Many operators and functions do not accept a null value. Other values are equivalent to null. Each of these IF functions returns <code>"falseResult"</code> . The final example shows that an unquoted string that does not represent an object resolves to null, or false. <ul style="list-style-type: none"> • <code>IF((()), "trueResult", "falseResult")</code> • <code>IF((0), "trueResult", "falseResult")</code> • <code>IF(([]), "trueResult", "falseResult")</code> • <code>IF((string), "trueResult", "falseResult")</code>

Some functions define constants specific to their use. For example, the **ROUND** function accepts an

optional constant that indicates the direction in which to round the result: `CEILING`, `DOWN`, `FLOOR`, `HALF_DOWN`, `HALF_EVEN`, `HALF_UP`, and `UP`.

Omnistudio Error Cases

Omnistudio functions and operators return errors when they can't perform a requested operation in a Data Mapper or Integration Procedure formula. Error cases can occur for missing or invalid parameters, or for values or expressions that cannot be evaluated in the context in which they're used. Error cases can also occur for invalid operators or functions.

- Functions and operators return `{ }` (empty curly braces) when the values or parameters they expect are missing or are not of the correct type. They can also return `{ }` when they can't evaluate a value or expression, or when they can't perform a requested operation.

Error Case	Description
<pre>IF(([("")], "trueResult", "falseResult") IF(([%emptyField%]), "trueResult", "falseResult")</pre>	<p>First example: The <code>IF</code> function can't evaluate an array that contains an empty string.</p> <p>Second example: The value of <code>%emptyField%</code> is <code>" "</code>.</p>
<pre>SQRT(-4) 3 / 0 IF((3 \ 3 == 1), "trueResult", "falseResult")</pre>	<p>First example: The <code>SQRT</code> function can't calculate the square root of a negative number.</p> <p>Second example: The division operator can't divide by zero.</p> <p>Third example: The division operator is entered incorrectly.</p>
<pre>AGE("01 Jan 2025") HOUR("21:00:56 +0900")</pre>	<p>First example: The <code>AGE</code> function doesn't accept a date in the format shown.</p> <p>Second example: The <code>HOUR</code> function doesn't accept a time where the time zone is separated by a space.</p>
<pre>1.2 ~= 1.2</pre>	The <code>~=</code> operator accepts only strings, not numbers.
<pre>BASE64ENCODE(null) CONCAT("") TOSTRING(null) TOSTRING("")</pre>	These string functions don't accept a null or empty value.
<pre>JOIN("The string.") SPLIT("The string.") SPLIT("numerator/denominator" "/")</pre>	<p>First and second examples: The <code>JOIN</code> and <code>SPLIT</code> functions require an additional <i>token</i> parameter that specifies the character by which a string is to be split or with which strings are to be joined.</p> <p>Third example: The <i>token</i> parameter is present but malformed: it's missing closing double quotes.</p>

Error Case	Description
QUERY("SELECT Name FROM Account WHERE BillingState ='PA'")	The <code>SELECT</code> clause of the <code>QUERY</code> function didn't find any accounts whose billing state is <code>PA</code> .

- Functions and operators return `error` messages for important problems they encounter. The most common error cases occur for nonexistent or mistyped function or operator names.

Error Case	Description
Sqrt(4)	"error": "Formula expression is invalid: Sqrt is not a Function or Operator" The <code>Sqrt</code> function is not found. Function names must be entered in all uppercase letters.
IF(NO(true), "trueResult", "falseResult")	"error": "Formula expression is invalid: NO is not a Function or Operator" The <code>NO</code> operator is not found. The name of the intended operator is <code>NOT</code> .
DAYOFWEEK("2025-01-01")	"error": "Formula expression is invalid: DAYOFWEEK is not a Function or Operator" The <code>DAYOFWEEK</code> function is an Omniscript function that's not available for Data Mapper and Integration Procedure formulas.
IF(%Amount% > 0, "Positive", "Zero") IF(%Result% == "Positive", "Pass", "Fail")	"error": "Formula expression is invalid: \nIF is not a Function or Operator" A formula can include only a single operation. It can include nested functions and operators, each of which can appear on a separate line, as can parameters and arguments. But a formula can't include two independent operations. The formula editor doesn't support line continuation characters (for example, <code>\n</code>) or termination characters (for example, <code>;</code>) between independent operations. These formulas are valid because they contain linebreaks within a single operation. <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> IF(%Amount% > 0, "Positive", "Zero") </div> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> ("abc" ==) </div>

Error Case	Description
	"ABC")

Supported Data Mapper and Integration Procedure Operators

Use operators to develop more complex formulas for Omnistudio Data Mappers and Integration Procedures. For example, operators can perform logical, comparison, or mathematical operations on values in expressions. Operators are evaluated according to a strict precedence.

-  **Note** These operators work with formulas for Omnistudio Data Mappers and Integration Procedures. For information about operators that you can use with Omniscript formulas and aggregates, see [Create a Formula or Aggregate in an Omniscript](#).

Omnistudio Operators

Omnistudio operators are grouped into categories: logical, comparison, mathematical, string, and precedence. The categories reflect the role each type of operator performs in a Data Mapper or Integration Procedure formula.

Omnistudio Operator Precedence

Precedence indicates the order in which the engine evaluates Omnistudio operators in expressions in Data Mapper and Integration Procedure formulas.

Omnistudio Operators

Omnistudio operators are grouped into categories: logical, comparison, mathematical, string, and precedence. The categories reflect the role each type of operator performs in a Data Mapper or Integration Procedure formula.

-  **Note** Documentation about Omnistudio operators was updated Summer '25:

- The `=` operator is an equality operator, not an assignment operator.
- All string comparisons with operators are case-insensitive. For strings, the `==`, `=`, and `~=` operators return the same results.
- The `NOT` logical complement operator was added to the documentation.
- The `%` remainder operator was removed from the documentation.

Logical operators return either true or false depending on the values and expressions they evaluate. They don't accept `NULL` values, and they can never return `NULL`.

Operator	Syntax	Description
<code>&&</code> <code>AND</code>	<code>x && y</code> <code>x AND y</code>	AND logical operator. If both <code>x</code> and <code>y</code> evaluate to true, the expression evaluates to true. Otherwise, the expression evaluates to false.
<code> </code> <code>OR</code>	<code>x y</code> <code>x OR y</code>	OR logical operator. If both <code>x</code> and <code>y</code> evaluate to false, the expression evaluates to false. Otherwise, the expression evaluates to true.
<code>NOT</code>	<code>NOT(x)</code> <code>NOT(x == y)</code>	Logical complement operator. Inverts the value of a Boolean value or expression so that true becomes false and false becomes true. If <code>x</code> is true, the operator inverts its Boolean value to false. If <code>x</code> is equal to <code>y</code> , the operator inverts the evaluation of the expression to false.

Comparison operators return either true or false depending on the values and expressions they evaluate. All mathematical comparisons are based on numeric value. All string comparisons are case-insensitive and lexicographical. Comparison operators treat dates and times as strings, so don't rely on them to compare dates and times. A comparison can never return `NULL`.

Operator	Syntax	Description
<code>></code>	<code>x > y</code>	Greater than operator. If <code>x</code> is greater than <code>y</code> , the expression evaluates to true. Otherwise, the expression evaluates to false.
<code>>=</code>	<code>x >= y</code>	Greater than or equal to operator. If <code>x</code> is greater than or equal to <code>y</code> , the expression evaluates to true. Otherwise, the expression evaluates to false.
<code><</code>	<code>x < y</code>	Less than operator. If <code>x</code> is less than <code>y</code> , the expression evaluates to true. Otherwise, the expression evaluates to false.
<code><=</code>	<code>x <= y</code>	Less than or equal to operator. If <code>x</code> is less than or equal to <code>y</code> , the expression evaluates to true. Otherwise, the expression evaluates to false.
<code>==</code> <code>=</code>	<code>x == y</code> <code>x = y</code>	Equality operator. If <code>x</code> is equal to <code>y</code> , the expression evaluates to true. Otherwise, the expression evaluates to false.

Operator	Syntax	Description
<code>!=</code>	<code>x != y</code>	Inequality operator. If <code>x</code> is not equal to <code>y</code> , the expression evaluates to true. Otherwise, the expression evaluates to false.
<code><></code>	<code>x <> y</code>	

Mathematical operators return a number, which can be an integer or a decimal value. If either operand is a decimal value, the result is a decimal value.

Operator	Syntax	Description
<code>+</code>	<code>x + y</code>	Addition operator. Adds the value of <code>x</code> to the value of <code>y</code> .
<code>-</code>	<code>x - y</code>	Subtraction operator. Subtracts the value of <code>x</code> from the value of <code>y</code> .
<code>*</code>	<code>x * y</code>	Multiplication operator. Multiplies the value of <code>x</code> by the value of <code>y</code> .
<code>/</code>	<code>x / y</code>	Division operator. Divides the value of <code>x</code> by the value of <code>y</code> . If the operation results in a fraction, the result is a decimal value.
<code>^</code>	<code>x ^ y</code>	Exponential operator. Raises the value of <code>x</code> to the power of <code>y</code> . For example, <code>2 ^ 3</code> results in <code>8</code> .

String operators return either true or false depending on the strings they evaluate. They don't accept NULL values, and they can never return NULL.

Operator	Syntax	Description
<code>LIKE</code>	<code>x LIKE y</code>	Case-insensitive substring operator. If <code>x</code> contains <code>y</code> (if <code>y</code> is a substring of <code>x</code>), the expression returns true. Otherwise, it returns false. For example, <code>"ABC" LIKE "A"</code> and <code>"ABC" LIKE "a"</code> both return true.
<code>NOTLIKE</code>	<code>x NOTLIKE y</code>	Case-insensitive substring complement operator. If <code>x</code> does not contain <code>y</code> (if <code>y</code> is not a substring of <code>x</code>), the expression returns true. Otherwise, it returns false. For example, <code>"ABC" NOTLIKE "A"</code> and <code>"ABC" NOTLIKE "a"</code> both return false.

Operator	Syntax	Description
<code>~=</code>	<code>x ~= y</code>	Case-insensitive string equality operator. If <code>x</code> and <code>y</code> are the same string regardless of case, the expression returns true. Otherwise, it returns false. For example, <code>"ABC" ~= "abc"</code> returns true.

The precedence operator causes an expression to be evaluated with higher precedence.

Operator	Syntax	Description
<code>()</code>	<code>(x = y)</code> <code>(x + y)</code>	Precedence operator. Elevates the precedence of the expression <code>x</code> so that it's evaluated first in a compound expression. Precedence operators can be nested.

Omnistudio Operator Precedence

Precedence indicates the order in which the engine evaluates Omnistudio operators in expressions in Data Mapper and Integration Procedure formulas.

Operators in higher rows have precedence over operators in lower rows. For example, the `()` operator appears in the top row of the table and has a precedence value of 1. It is always given the highest precedence. Operators in the same row have equivalent precedence; in an expression, they're evaluated left to right in the order in which they appear.

Precedence	Operators	Description
1	<code>()</code>	Precedence operator
2	<code>NOT</code>	Logical complement operator
3	<code>^</code>	Exponential operator
4	<code>* /</code>	Multiplication and division operators
5	<code>+ -</code>	Addition and subtraction operators
6	<code>> >= < <=</code>	Greater-than and less-than comparison operators

Precedence	Operators	Description
7	<code>== = != <></code>	Equality and inequality comparison operators
8	<code>LIKE</code> <code>NOTLIKE ~=</code>	String comparison operators
9	<code>&& AND</code>	AND logical operators
10	<code> OR</code>	OR logical operators

Supported Data Mapper and Integration Procedure Functions

Use functions to evaluate and manipulate data in formulas for Omnistudio Data Mappers and Integration Procedures. For example, functions can perform conditional operations that take different actions depending on the values you pass them. They can invoke queries and other functions, and they can operate on numbers, dates and times, strings, lists and arrays, and JSON objects.

-  **Note** These functions work with Omnistudio Data Mappers and Integration Procedures. For information about functions that you can use with Omniscript formulas and aggregates, see [Create a Formula or Aggregate in an Omniscript](#).

Syntax conventions and a summary of recent changes follow the table.

Omnistudio Functions	Supported Functions	
Conditional Functions	<code>IF(expression, trueResult, falseResult)</code> <code>ISBLANK(expression)</code>	<code>ISNOTBLANK(expression)</code>
Mathematical Functions	<code>ABS(expression)</code> <code>ROUND(expression, precision, direction)</code>	<code>SQRT(expression)</code>
Date and Time Functions	<code>ADDDAY(date, days)</code>	<code>FORMATDATETIMEGMT(datetime,</code>

Omnistudio Functions	Supported Functions
	<p><code>ADDMONTH(date, months)</code></p> <p><code>ADDYEAR(date, years)</code></p> <p><code>AGE(birthDate)</code></p> <p><code>AGEON(birthDate, date)</code></p> <p><code>DATEDIFF(firstDate, secondDate)</code></p> <p><code>DATETIMEUNIX(datetime)</code></p> <p><code>DAY(date)</code></p> <p><code>EOM(date)</code></p> <p><code>FORMATDATETIME(datetime, format, timezone)</code></p>
	<p><code>timezone, format)</code></p> <p><code>HOUR(time)</code></p> <p><code>MINUTE(time)</code></p> <p><code>MONTH(date)</code></p> <p><code>NOW(format)</code></p> <p><code>SECOND(time)</code></p> <p><code>TIMEDIFF(firstTime, secondTime)</code></p> <p><code>TODAY()</code></p> <p><code>UNIXDATETIME(timestamp)</code></p> <p><code>YEAR(date)</code></p>
String Functions	<p><code>BASE64ENCODE(data)</code></p> <p><code>CONCAT(string...)</code></p> <p><code>JOIN(string..., token)</code></p> <p><code>MAXSTRING(string...)</code></p>
List and Array Functions	<p><code>SPLIT(string, token)</code></p> <p><code>STRINGINDEXOF(string, substring)</code></p> <p><code>SUBSTRING(string, startIndex, endIndex)</code></p> <p><code>TOSTRING(data)</code></p>
	<p><code>AVG(list)</code></p> <p><code>FILTER(LIST(list), condition)</code></p> <p><code>LIST(expression)</code></p> <p><code>LISTMERGE(mergeKey..., LIST(list) ...)</code></p> <p><code>LISTMERGEPRIMARY(mergeKey..., LIST(list) ...)</code></p>
	<p><code>MAPTOLIST(jsonObject)</code></p> <p><code>MAX(list)</code></p> <p><code>MIN(list...)</code></p> <p><code>SORTBY(LIST(list), key..., [:DSC])</code></p> <p><code>SUM(list)</code></p>

Omnistudio Functions	Supported Functions	
	<code>LISTSIZE(list)</code>	
JSON Object Functions	<code>DESERIALIZE(jsonString)</code> <code>RESERIALIZE(jsonString)</code>	<code>SERIALIZE(jsonObject)</code> <code>VALUELOOKUP(startNode, node...)</code>
Invocation Functions	<code>COUNTQUERY(query)</code> <code>FUNCTION(class, method, input...)</code>	<code>GENERATEGLOBALKEY(prefix)</code> <code>QUERY(query)</code>

Omnistudio Function Syntax

Many Omnistudio functions use these parameters and conventions:

- *expression* refers to a single value or to a construct that contains variables, operators, and functions. The expression must resolve to the data type of the parameter.
- *value* refers to a single value that matches the data type of the parameter.
- *data* refers to multiple data types. The parameter can accept a value, an expression, or a list of values. The parameter description documents the valid data types.
- *...* (ellipses) indicate parameters that accept multiple arguments as a comma-separated list of values. The arguments must match the data type of the parameter.
- A formula can include only a single operation. It can include nested functions and operators, each of which can appear on a separate line, as can parameters and arguments. But a formula can't include two independent operations. The formula editor doesn't support line continuation characters (such as `\`) or termination characters (such as `;`) between independent operations. For an example of the error for invalid newlines in a formula, see [Omnistudio Error Cases](#).

For more information about specifying different data types, using constants, and understanding the errors that a function can return, see [Supported Data Mapper and Integration Procedure Data Types](#).

Omnistudio Function Updates

Documentation about Omnistudio functions was updated Summer '25:

- Parameter names and syntax were changed for consistency and clarity. Unless otherwise noted, all functions have the same parameters and usage.
- These date and time functions were added to the documentation:
 - `DAY(date)`
 -

- HOUR(*time*)
- MINUTE(*time*)
- SECOND(*time*)
- TIMEDIFF(*firstTime, secondTime*)

- The date and time `TODAY()` function no longer accepts a *format* parameter.
- These string functions were added to the documentation:
 - JOIN(*string..., token*)
 - SPLIT(*string, token*)
- The list and array function `MAPTOLIST(jsonObject)` was added to the documentation.
- These functions were removed from the documentation:
 - BASEURL
 - InvokeIP
 - ORDERITEMATTRIBUTES

Omnistudio Conditional Functions

Functions that perform conditional operations.

Omnistudio IF Function

Evaluates an expression to determine whether it's true or false and returns the indicated response. If the expression evaluates to true, the function returns the true (first) result. Otherwise, the function returns the false (second) result.

Omnistudio ISBLANK Function

Returns true if an expression is blank or empty. Otherwise, the function returns false.

Omnistudio ISNOTBLANK Function

Returns true if an expression is not blank or empty. Otherwise, the function returns false.

Omnistudio IF Function

Evaluates an expression to determine whether it's true or false and returns the indicated response. If the expression evaluates to true, the function returns the true (first) result. Otherwise, the function returns the false (second) result.

! **Important** Both the true and false branches are always evaluated and it returns the result of the false branch. This becomes significant when either branch contains operations with side effects (such as `InvokeIP`, , DML operations, or callouts), because actions in the true branch may still execute even when the condition is false.

For example:

```
IF(  
    ISNOTBLANK(Account.Id),
```

```

InvokeIP(
    'AccountCreateIP_AccountCreateIP',
    INPUT('SomeKey', 'SomeVal'),
    'OutputKey'
),
'IP Not Called'
)

```

In this example, even if `Account.Id` is blank (which should lead to returning `"IP Not Called"`), the `InvokeIP` function in the true branch is still evaluated.

Signature

```
IF(expression, trueResult, falseResult)
```

Return Value

Any data type

Parameters

Parameter	Data Type	Necessity	Description
<code>expression</code>	Expression	Required	An expression that resolves to true or false.
<code>trueResult</code>	Expression	Required	An expression that resolves to the value to be returned if the <code>expression</code> parameter is true.
<code>falseResult</code>	Expression	Required	An expression that resolves to the value to be returned if the <code>expression</code> parameter is false.

 **String Comparison Example** Formula: `IF(("abc" LIKE "a"), "true", "false")` Return value: `"true"`

 **String Comparison Example** Formula: `IF(("a" LIKE "abc"), "true", "false")` Return value: `"false"`

 **Evaluate a Date Example** Sample data: `InputDate: "1999-07-01"` Formula: `IF(InputDate < "2000-01-01", "20th Century", "21st Century")` Return value: `"20th Century"`

 **Parse a Substring Example** Sample data: `"Price": "$800.00"` Formula: `IF(SUBSTRING(Price, 0, 1) == "$", SUBSTRING(Price, 1), Price)` Return value: `"800.00"`

-  **Avoid Division by Zero Example** Sample data: `"Amount": 2` Formula: `IF(Amount > 0, 1 / IF(Amount > 0, Amount, 1), 0)` Return value: `0.5`
-  **Avoid Division by Zero Example** Sample data: `"Amount": 0` Formula: `IF(Amount > 0, 1 / IF(Amount > 0, Amount, 1), 0)` Return value: `0`
-  **Alternative Division by Zero Example** Sample data: `"Amount": 0` Formula: `IF(Amount > 0, 1 / IF(Amount > 0, Amount, 1), "Undefined")` Return value: `"Undefined"`

Omnistudio ISBLANK Function

Returns true if an expression is blank or empty. Otherwise, the function returns false.

Signature

```
ISBLANK(expression)
```

Return Value

Boolean

Parameters

Parameter	Data Type	Necessity	Description
<code>expression</code>	String or list	Required	An expression that resolves to a string or list to determine whether it is empty.

-  **String Example** Formula: `ISBLANK("")` Return value: `true`
-  **List Example** Formula: `ISBLANK(())` Return value: `true`
-  **Array Example** Formula: `ISBLANK([])` Return value: `true`
-  **JSON Array Example** Sample data: `"EmptyList": []` Formula: `ISBLANK(%EmptyList%)` Return value: `true`

Omnistudio ISNOTBLANK Function

Returns true if an expression is not blank or empty. Otherwise, the function returns false.

Signature

`ISNOTBLANK (expression)`

Return Value

Boolean

Parameters

Parameter	Data Type	Necessity	Description
<code>expression</code>	String or list	Required	An expression that resolves to a string or list to determine whether it is not empty.

 **String Example** Formula: `ISNOTBLANK('Hello world')` Return value: `true`

 **List Example** Formula: `ISNOTBLANK(("a", "b"))` Return value: `true`

 **Array Example** Formula: `ISNOTBLANK([1, 2])` Return value: `true`

 **JSON Array Example** Sample data: `"NumberList": [1, 2]` Formula:
`ISNOTBLANK (%NumberList%)` Return value: `true`

Omnistudio Mathematical Functions

Functions that operate on numbers.

Omnistudio ABS Function

Returns the absolute value of a numeric expression.

Omnistudio ROUND function

Rounds a numeric expression up or down to a specified level of precision.

Omnistudio SQRT Function

Returns the square root of a numeric expression.

Omnistudio ABS Function

Returns the absolute value of a numeric expression.

Signature

`ABS (expression)`

Return Value

Number

Parameters

Parameter	Data Type	Necessity	Description
<i>expression</i>	Number	Required	An expression that resolves to the number for which the absolute value is to be returned.

 **Positive Expression Example** Formula: `ABS(4)` Return value: `4`

 **Negative Expression Example** Formula: `ABS(-4.50)` Return value: `4.5`

 **Negative Expression Example** Formula: `ABS(2 - (2 * 3))` Return value: `4`

Omnistudio ROUND function

Rounds a numeric expression up or down to a specified level of precision.

Signature

```
ROUND(expression, precision, direction)
```

Return Value

Number

Parameters

Parameter	Data Type	Necessity	Description
<i>expression</i>	Number	Required	An expression that resolves to the number to be rounded.
<i>precision</i>	Integer	Optional	A non-negative integer that indicates the number of decimal places to which to round the result. Specify <code>0</code> to return an integer. Specify a positive integer to round to that number of decimal places. Do not specify a negative number or a decimal

Parameter	Data Type	Necessity	Description
			<p>value.</p> <p>If the number has fewer decimal places than the specified precision, the function returns only the lower number of decimal places. Specify a precision of 0 to return an integer. By default, the function rounds the result to two decimal places of precision.</p>
<i>direction</i>	Constant	Optional	<p>A constant that indicates the direction in which to round the result:</p> <ul style="list-style-type: none"> • <code>CEILING</code> rounds the least significant digit of the result up. • <code>DOWN</code> rounds the least significant digit of the result down. The digit remains unchanged regardless of the digit to its right. • <code>FLOOR</code> rounds the least significant digit of the result down. • <code>HALF_DOWN</code> rounds the least significant digit of the result down if the digit to its right is less than or equal to 5. • <code>HALF_EVEN</code> rounds the least significant digit of the result up or down to the nearest even digit if the digit to its right is equal to 5. • <code>HALF_UP</code> rounds the least significant digit of the result up if the digit to its right is greater than or equal to 5. • <code>UP</code> rounds the least significant digit of the result up. The digit increases by one regardless of the digit to its right. <p>By default, the function rounds up or down based on the digit to the right of the least significant digit of the result (to the right of the specified level of precision). If you specify a <i>direction</i>, you must also specify a <i>precision</i>.</p>

 **Round with No Precision and No Direction Example** Formula: `ROUND (4.115)` Return value: `4.12`

 **Round with Precision 0 and No Direction Example** Formula: `ROUND (4.115, 0)` Return value: `4`

-  **Round with Precision 2 and Direction DOWN Example** Formula: `ROUND(2.119, 2, DOWN)`
Return value: `2.11`
-  **Round with Precision 2 and Direction UP Example** Formula: `ROUND(2.111, 2, UP)` Return value: `2.12`
-  **Round with Precision 1 and Direction FLOOR Example** Formula: `ROUND(2.55, 1, FLOOR)`
Return value: `2.5`
-  **Round with Precision 1 and Direction CEILING Example** Formula: `ROUND(2.55, 1, CEILING)`
Return value: `2.6`
-  **Round with Precision 0 and Direction FLOOR Example** Formula: `ROUND(2.5, 0, FLOOR)`
Return value: `2`
-  **Round with Precision 0 and Direction CEILING Example** Formula: `ROUND(2.5, 0, CEILING)`
Return value: `3`
-  **Round with Precision 2 and Direction HALF_DOWN Example** Formula: `ROUND(2.575, 2, HALF_DOWN)` Return value: `2.57`
-  **Round with Precision 2 and Direction HALF_UP Example** Formula: `ROUND(2.575, 2, HALF_UP)` Return value: `2.58`
-  **Round with Precision 2 and Direction HALF_EVEN Example** Formula: `ROUND(2.225, 2, HALF_EVEN)` Return value: `2.22`

Omnistudio SQRT Function

Returns the square root of a numeric expression.

Signature

```
SQRT(expression)
```

Return Value

Number

Parameters

Parameter	Data Type	Necessity	Description
<code>expression</code>	Number	Required	An expression that resolves to the non-negative

Parameter	Data Type	Necessity	Description
			number for which the square root is to be returned.

 **Positive Integer Example** Formula: `SQRT(3 * 12)` Return value: `6`

 **Positive Decimal Number Example** Formula: `SQRT(2.5)` Return value: `1.5811388300841898`

 **Negative Integer Example** Formula: `SQRT(-4)` Return value: `{}`

Omnistudio Date and Time Functions

Functions that operate on dates and times.

[Omnistudio ADDDAY Function](#)

Returns a date and time after adding a specified number of days.

[Omnistudio ADDMONTH Function](#)

Returns a date and time after adding a specified number of months.

[Omnistudio ADDYEAR Function](#)

Returns a date and time after adding a specified number of years.

[Omnistudio AGE Function](#)

Returns the age in years for a specified birth date.

[Omnistudio AGEON Function](#)

Returns the age in years for a specified birth date on a specified date.

[Omnistudio DATEDIFF Function](#)

Returns the difference in number of days between two specified dates as an integer value.

[Omnistudio DATETIMETOUNIX Function](#)

Returns the number of milliseconds in UNIX time for a specified date and time.

[Omnistudio DAY Function](#)

Returns the day for a specified date as an integer value.

[Omnistudio EOM Function](#)

Returns the date and time of the last day of the month for a specified date.

[Omnistudio FORMATDATETIME Function](#)

Returns a date and time as a string in a specified format and time zone.

[Omnistudio FORMATDATETIMEGMT Function](#)

Returns a date and time as a string in a specified format and in the GMT time zone.

[Omnistudio HOUR Function](#)

Returns the hour for a specified time as an integer value.

[Omnistudio MINUTE Function](#)

Returns the minute for a specified time as an integer value.

Omnistudio MONTH Function

Returns the month for a specified date as an integer value.

Omnistudio NOW Function

Returns the current date and time in a specified format.

Omnistudio SECOND Function

Returns the second for a specified time as an integer value.

Omnistudio TIMEDIFF Function

Returns the difference in number of milliseconds between two specified times as an integer.

Omnistudio TODAY Function

Returns the current date.

Omnistudio UNIXTODATETIME Function

Returns the date and time for a number of seconds or milliseconds specified in UNIX time.

Omnistudio YEAR Function

Returns the year for a specified date as an integer value.

Omnistudio ADDDAY Function

Returns a date and time after adding a specified number of days.

If you specify a date and time, use the date format `"YYYY-MM-DD"` to preserve the input time. Use the date format `"MM/DD/YYYY"` to reset the time to `00:00:00`.

Signature

`ADDDAY(date, days)`

Return Value

Datetime

Parameters

Parameter	Data Type	Necessity	Description
<code>date</code>	Datetime	Required	A date specified as a string in Day.js format . You can specify a date and time.
<code>days</code>	Integer	Required	The number of days to be added to the specified date. Use a negative number to subtract days from the date.

 **Add Days Example** Formula: `ADDDAY("2025-01-01", 31)` Formula: `ADDDAY("01/01/2025", 31)` Formula: `ADDDAY("01/01/2025T12:00:00Z", 31)` Return value: `"2025-02-01T00:00:00.000Z"`

 **Add Days Example** Formula: `ADDDAY("2025-01-01T12:00:00Z", 31)` Return value: `"2025-02-01T12:00:00.000Z"`

 **Subtract Days Example** Formula: `ADDDAY("2025-01-01", -100)` Formula: `ADDDAY("2025-01-01T00:00:00Z", -100)` Return value: `"2024-09-23T00:00:00.000Z"`

Omnistudio ADDMONTH Function

Returns a date and time after adding a specified number of months.

If you specify a date and time, use the date format `"YYYY-MM-DD"` to preserve the input time. Use the date format `"MM/DD/YYYY"` to reset the time to `00:00:00`.

Signature

`ADDMONTH(date, months)`

Return Value

Datetime

Parameters

Parameter	Data Type	Necessity	Description
<code>date</code>	Datetime	Required	A date specified as a string in Day.js format . You can specify a date and time.
<code>months</code>	Integer	Required	The number of months to be added to the specified date. Use a negative number to subtract months from the date.

 **Add Months Example** Formula: `ADDMONTH('2025-25-01', 15)` Formula: `ADDMONTH("01/25/2025", 15)` Formula: `ADDMONTH("01/25/2025T16:35:30Z", 15)` Return value: `"2026-04-25T00:00:00.000Z"`

 **Add Months Example** Formula: `ADDMONTH("2025-01-25T16:35:30Z", 15)` Return value: `"2026-04-25T16:35:30.000Z"`

 **Subtract Months Example** Formula: `ADDMONTH("01/25/2025", -15)` Formula: `ADDMONTH('01/25/2025T16:35:30Z', -15)` Return value: `"2023-10-25T00:00:00.000Z"`

Omnistudio ADDYEAR Function

Returns a date and time after adding a specified number of years.

If you specify a date and time, use the date format `"YYYY-MM-DD"` to preserve the input time. Use the date format `"MM/DD/YYYY"` to reset the time to `00:00:00`.

Signature

`ADDYEAR(date, years)`

Return Value

Datetime

Parameters

Parameter	Data Type	Necessity	Description
<code>date</code>	Datetime	Required	A date specified as a string in Day.js format . You can specify a date and time.
<code>years</code>	Integer	Required	The number of years to be added to the specified date. Use a negative number to subtract years from the date.

 **Add Years Example** Formula: `ADDYEAR("2025-01-01", 10)` Formula: `ADDYEAR("01/01/2025", 10)` Formula: `ADDYEAR("01/01/2025T16:35:30", 10)` Return value: `"2035-01-01T00:00:00.000Z"`

 **Add Years Example** Formula: `ADDYEAR("2025-01-01T16:35:30", 10)` Return value: `"2035-01-01T16:35:30.000Z"`

 **Subtract Years Example** Formula: `ADDYEAR("2025-01-01", -10)` Formula: `ADDYEAR("2025-01-01T00:00:00", -10)` Return value: `"2015-01-01T00:00:00.000Z"`

Omnistudio AGE Function

Returns the age in years for a specified birth date.

Signature

`AGE (birthDate)`

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>birthDate</code>	Datetime	Required	A date specified as a string in Day.js format . You can specify a date and time.

 **Date String Example** Formula: `AGE ("02/15/2002")` Formula: `AGE ("2002-02-15")` Return value: `23`

 **Date and Time String Example** Formula: `AGE ("2002-02-15T16:35:30")` Formula: `AGE ("2002-02-15T16:35:30Z")` Return value: `23`

Omnistudio AGEON Function

Returns the age in years for a specified birth date on a specified date.

The function calculates an age based only on dates; it ignores times. The function returns `0` if the specified `date` is less than a year from or is earlier than the `birthDate`.

Signature

`AGEON (birthDate, date)`

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>birthDate</code>	Datetime	Required	The birth date for which the age is to be calculated. Specify a date as a string in Day.js

Parameter	Data Type	Necessity	Description
			format . You can specify a date and time.
<code>date</code>	Datetime	Required	A date for which the function is to calculate an age based on the birth date. Specify a date as a string in Day.js format . You can specify a date and time.

 **Date and Time String Example** Formula: `AGEON ("02/15/2002T16:35:30Z", "2030-02-28")`

Formula: `AGEON ("2002-02-15", "2030-02-28T16:35:30Z")` Return value: `28`

 **JSON Object Example** Sample data:

```
"BirthDate": "6/25/2004T09:05:00 GMT -0500 (EDT)"
"FutureDate1": "02/28/2030",
"FutureDate2": "2030-02-28"
```

Formula: `AGEON (%Birthdate%, %FutureDate1%)` Formula: `AGEON ("BirthDate%,
%FutureDate2%)` Return value: `25`

Omnistudio DATEDIFF Function

Returns the difference in number of days between two specified dates as an integer value.

The first date is subtracted from the second date. If the first date is greater than the second date, the function returns a negative integer.

If you include a time, the function considers it in its calculation. If you specify a time zone, the function considers it in its calculation.

Signature

```
DATEDIFF (firstDate, secondDate)
```

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>firstDate</code>	Date	Required	A date specified as a string in Day.js format . You can specify a date and time.
<code>secondDate</code>	Date	Required	A date for which the function is to calculate the difference in number of days between the first date. Specify a date as a string in Day.js format . You can specify a date and time.

👁 **Positive Days Example** Formula: `DATEDIFF("02/01/2000", "2001-02-01")` Return value: `366`

👁 **Negative Days Example** Formula: `DATEDIFF("2001-02-01", "02/01/2000")` Return value: `-366`

👁 **Date and Time Example** Formula: `DATEDIFF("2025-02-04T00:00:00", "2025-03-04T00:00:00")` Return value: `28`

👁 **Date and Time Example** Formula: `DATEDIFF("2025-02-04T12:00:00", "2025-03-04T11:59:59")` Return value: `27`

👁 **Date and Time with Time Zone Example** Formula: `DATEDIFF("2025-02-04T12:00:00-0500", "2025-03-04T11:59:59+0000")` Return value: `28`

👁 JSON Object Examples

Sample data:

```
"Account": [
  {
    "Cases": [
      {
        "Case1": {
          "CreatedDate": "2/1/2024T16:35:30 GMT -0500 (EDT)",
          "LastUpdate": "2/8/2024T09:15:00 GMT -0500 (EDT)"
        }
      },
      {
        "Case2": {
          "CreatedDate": "2/2/2025T11:05:05 GMT -0500 (EDT)",
          "LastUpdate": "2/3/2025T15:50:57 GMT -0500 (EDT)"
        }
      }
    ]
  }
]
```

```

    }
]
}
]
```

Formula: `DATEDIFF(Account:Cases:Case1:CreatedDate, Account:Cases:Case1:LastUpdate)` **Return value:** 7 **Formula:** `DATEDIFF(%Account:Cases:Case2:CreatedDate%, TODAY())` **Return value:** 8

Omnistudio DATETIMETOUNIX Function

Returns the number of milliseconds in UNIX time for a specified date and time.

UNIX time is the number of non-leap seconds that have passed since 00:00:00 on 1 January 1970, Coordinated Universal Time (UTC). UTC is the same as Greenwich Mean Time (GMT). The value is referred to as the UNIX epoch.

Signature

`DATETIMETOUNIX(datetime)`

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>datetime</code>	Datetime	Required	A date and time specified as a string in Day.js format . You can also specify just a date.

 **Date and Time Example** **Formula:** `DATETIMETOUNIX("2002-02-01T16:35:30:000Z")` **Formula:** `DATETIMETOUNIX("2002-02-01T16:35:30:000")` **Formula:** `DATETIMETOUNIX("2002-02-01T16:35:30")` **Formula:** `DATETIMETOUNIX("02/01/2002T16:35:30")` **Return value:** 1012581330000

 **Date And Zero Time Example** **Formula:** `DATETIMETOUNIX("2002-02-01")` **Formula:** `DATETIMETOUNIX("02/01/2002")` **Formula:** `DATETIMETOUNIX("2002-02-01T00:00:00")` **Formula:** `DATETIMETOUNIX("02/01/2002T00:00:00")` **Formula:** `DATETIMETOUNIX("2002-02-01T00:00:00Z")` **Formula:** `DATETIMETOUNIX("02/01/2002T00:00:00Z")` **Return value:** 1012521600000

Omnistudio DAY Function

Returns the day for a specified date as an integer value.

Signature

`DAY(date)`

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>date</code>	Datetime	Required	A date specified as a string in Day.js format . You can specify a date and time.

 **Date String Example** Formula: `DAY("2025-01-31")` Formula: `DAY("01/31/2025")` Formula: `DAY("1/31/2025")` Return value: `31`

 **JSON Object Example** Sample data: `"Birthdate": "2002-02-15T16:35:30 GMT -0500 (EDT)"` Formula: `DAY(Birthdate)` Return value: `15`

Omnistudio EOM Function

Returns the date and time of the last day of the month for a specified date.

Signature

`EOM(date)`

Return Value

Datetime

Parameters

Parameter	Data Type	Necessity	Description
<code>date</code>	Datetime	Required	A date specified as a string in Day.js format . You

Parameter	Data Type	Necessity	Description
			can specify a date and time.

 **Date Example** Formula: `EOM("2024-02-01")` Formula: `EOM("02/01/2024")` Return value: `"2024-02-29T00:00:00.000Z"`

 **Date and Time Example** Formula: `EOM("02/01/2024T16:35:30")` Formula: `EOM("02/01/2024T16:35:30Z")` Return value: `"2024-02-29T00:00:00.000Z"`

Omnistudio FORMATDATETIME Function

Returns a date and time as a string in a specified format and time zone.

Greenwich Mean Time (GMT) is the same as Coordinated Universal Time (UTC), which serves as the basis of UNIX time and the UNIX epoch.

Signature

```
FORMATDATETIME(datetime, format, timezone)
```

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>datetime</code>	Datetime	Required	A date and time specified as a string in Day.js format . You can specify just a date or time. If you specify only a date, the default time is <code>00:00:00</code> . If you specify only a time, the default date is <code>1970-01-01</code> (the UNIX epoch).
<code>format</code>	String	Optional	A string that specifies the format in which to return the date and time. Use Java SimpleDateFormat notation to specify the format of the response. By default, the function returns the date and time in the format <code>"MM/DD/YY HH:mm aaa"</code> .

Parameter	Data Type	Necessity	Description
<code>timezone</code>	String	Optional	A string that indicates the time zone in which to return the date and time. By default, the function returns the date and time in the caller's time zone. To use a different time zone, specify a time zone identifier, for example, <code>America/New_York</code> . For a complete list of time zone identifiers, see IANA (TZDB) time zone information . If you specify a <i>timezone</i> , you must also specify a <i>format</i> .

- 🕒 **Date with No Time, Format, or Time Zone Example** Formula: `FORMATDATETIME ("01/31/2025")`
Return value: `"1/30/25 4:00 PM"`
- 🕒 **Datetime with No Format or Time Zone Example** Formula: `FORMATDATETIME ("01/31/2025T12:00:00")`
Return value: `"1/31/25 4:00 AM"`
- 🕒 **Datetime with No Time Zone Example** Formula: `FORMATDATETIME ("01/31/2025T12:00:00", "yyyy-MM-dd'T'HH:mm:ssZ")`
Return value: `"2025-01-31T04:00:00-0800"`
- 🕒 **Datetime Example** Formula: `FORMATDATETIME ("01/31/2025T12:00:00", "yyyy-MM-dd'T'HH:mm:ssZ", "America/Los_Angeles")`
Return value:
`"2025-01-31T04:00:00-0800"`
- 🕒 **Datetime Example** Formula: `FORMATDATETIME ("01/31/2025T12:00:00", "yyyy-MM-dd'T'HH:mm:ssZ", "America/New_York")`
Return value: `"2025-01-31T07:00:00-0500"`

Omnistudio FORMATDATETIMEGMT Function

Returns a date and time as a string in a specified format and in the GMT time zone.

Greenwich Mean Time (GMT) is the same as Coordinated Universal Time (UTC), which serves as the basis of UNIX time and the UNIX epoch.

Signature

```
FORMATDATETIMEGMT (datetime, timezone, format)
```

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>datetime</code>	Datetime	Required	A date and time specified as a string in Day.js format . You can specify just a date or time. If you specify only a date, the default time is <code>00:00:00</code> . If you specify only a time, the default date is <code>1970-01-01</code> (the UNIX epoch).
<code>timezone</code>	String	Optional	A string that indicates the time zone of the specified date and time. By default, the function uses the caller's time zone. To use a different time zone, specify a time zone identifier, for example, <code>America/New_York</code> . For a complete list of time zone identifiers, see IANA (TZDB) time zone information .
<code>format</code>	String	Optional	A string that specifies the format in which to return the date and time. Use Java SimpleDateFormat notation to specify the format of the response. By default, the function returns the date and time in the format <code>"MM/DD/YY HH:mm aaa"</code> . If you specify a <code>format</code> , you must also specify a <code>timezone</code> .

👁 **Date with No Time, Time Zone, or Format Example** Formula: `FORMATDATETIMEGMT ("01/31/2025")` Return value: `"2025-01-31T08:00:00.000Z"`

👁 **Datetime with No Time Zone or Format Example** Formula: `FORMATDATETIMEGMT ("01/31/2025T12:00:00")` Return value: `"2025-01-31T20:00:00.000Z"`

👁 **Datetime with No Format Example** Formula: `FORMATDATETIMEGMT ("01/31/2025T12:00:00", "America/Los_Angeles")` Return value: `"2025-01-31T20:00:00.000Z"`

👁 **Datetime Example** Formula: `FORMATDATETIMEGMT ("01/31/2025T12:00:00", "America/Los_Angeles", "yyyy-MM-dd'T'HH:mm:ssZ")` Return value:
`"2025-01-31T20:00:00+0000"`

👁 **Datetime Example** Formula: `FORMATDATETIMEGMT ("01/31/2025T12:00:00", "America/New_York", "yyyy-MM-dd'T'HH:mm:ssZ")` Return value: `"2025-01-31T17:00:00+0000"`

Omnistudio HOUR Function

Returns the hour for a specified time as an integer value.

Signature

```
Hour(time)
```

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<i>time</i>	Datetime	Required	A time specified as a string in Day.js format . You can specify a date and time.

 **Time Example** Formula: `HOUR("08:00:00")` Formula: `HOUR("T08:00:00")` Return value: `8`

 **Date and Time Example** Formula: `HOUR("2004-02-01T08:00:00Z")` Formula: `HOUR("02/01/2004T08:00:00Z")` Return value: `8`

 **JSON Object Example** Sample data: `"DateField": "2003-02-01T16:35:30Z"` Formula: `HOUR(%DateField%)` Return value: `16`

Omnistudio MINUTE Function

Returns the minute for a specified time as an integer value.

Signature

```
MINUTE(time)
```

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>time</code>	Datetime	Required	A time specified as a string in Day.js format . You can specify a date and time.

 **Time Example** Formula: `MINUTE("08:17:00")` Formula: `MINUTE("T08:17:00")` Return value: `17`

 **Date and Time Example** Formula: `MINUTE("2004-02-01T08:17:00Z")` Formula: `MINUTE("02/01/2004T08:17:00Z")` Return value: `17`

 **JSON Object Example** Sample data: `"DateField": "2003-02-01T16:35:30Z"` Formula: `MINUTE(DateField)` Return value: `35`

Omnistudio MONTH Function

Returns the month for a specified date as an integer value.

Signature

`MONTH(date)`

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>date</code>	Datetime	Required	A date specified as a string in Day.js format . You can specify a date and time.

 **Date String Example** Formula: `MONTH("2025-01-01")` Formula: `MONTH("01/01/2025")` Formula: `MONTH("1/1/2025")` Return value: `1`

 **JSON Object Example** Sample data: `"Birthdate": "2007-06-25T08:35:00Z"` Formula: `MONTH(%Birthdate%)` Return value: `6`

Omnistudio NOW Function

Returns the current date and time in a specified format.

By default, the function returns the date and time in Coordinated Universal Time (UTC). UTC is the same as Greenwich Mean Time (GMT). Use the `$Vlocity.NOW` environment variable to apply the time zone of the user or org.

You can use the `NOW` function with other functions to specify a date and time relative to the current date and time.

Signature

`NOW (format)`

Return Value

Datetime

Parameters

Parameter	Data Type	Necessity	Description
<code>format</code>	String	Optional	A format string in <code>SimpleDateFormat</code> notation that specifies the format of the response. By default, the function returns the date and time in the format <code>"YYYY-MM-dd'T'HH:mm:ss:SSS"</code> UTC (GMT).

 **No Format Example** Formula: `NOW()` Return value: `"2025-03-08T22:01:46.684Z"`

 **Java SimpleDateFormat Date and Time Example** Formula: `NOW("YYYY-MM-dd'T'HH:mm:ss")`
Return value: `"2025-03-08T22:10:42"`

 **Java SimpleDateFormat Date and Time Example** Formula: `NOW("YYYY-MM-dd'T'HH:mm:ss z")`
Return value: `"2025-03-08T22:08:24 GMT"`

 **Java SimpleDateFormat Date and Time Example** Formula: `NOW("YYYY-MM-dd'T'HH:mm:ss z")`
Return value: `"2025-03-08T22:07:30 +0000"`

 **Java SimpleDateFormat Date and Time Example** Formula: `NOW("YYYY-MM-dd'T'HH:mm:ss:SSS")`
Return value: `"2025-03-08T21:59:24:895"`

 **Java SimpleDateFormat Date and Time Example** Formula: `NOW("YYYY-MM-dd'T'HH:mm:ss:SSSz")`
Return value: `"2025-03-08T21:57:48:740GMT"`

- Java SimpleDateFormat Date and Time Example** Formula: `NOW("yyyy-MM-dd'T'HH:mm:ss:SSSZ")` Return value: `"2025-03-08T21:58:08:412+0000"`
- Java SimpleDateFormat Date-Only Example** Formula: `NOW("yyyy-MM-dd")` Formula: `NOW("MM/dd/yyyy")` Return value: `"03/08/2025"`
- Java SimpleDateFormat Time-Only Example** Formula: `NOW("HH:mm:ss")` Return value: `"22:12:30"`
- Java SimpleDateFormat Time-Only Example** Formula: `NOW("HH:mm:ss.SSS")` Return value: `"22:13:43.304"`
- Multiple Functions Example** Sample data: The function is called on 8 March 2025 at 10:16:19 UTC (GMT) ("2025-03-08T22:16:19Z"). Formula: `DATETIMEOUNIX(NOW("yyyy-MM-dd'T'HH:mm:ss"))` Return value: `1741472179000`
- Multiple Functions Example** Sample data: The function is called on 8 March 2025 at 10:16:19 UTC (GMT) ("2025-03-08T22:16:19.956Z"). Formula: `DATETIMEOUNIX(NOW())` Return value: `1741472179956`

Omnistudio SECOND Function

Returns the second for a specified time as an integer value.

Signature

```
SECOND(time)
```

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>time</code>	Datetime	Required	A time specified as a string in Day.js format . You can specify a date and time.

- Time Example** Formula: `SECOND("08:00:59")` Formula: `SECOND("T08:00:59")` Return value: `59`
- Date and Time Example** Formula: `SECOND("2004-02-01T08:00:59Z")` Formula: `SECOND("02/01/2004T08:00:59Z")` Return value: `59`

-  **JSON Object Example** Sample data: `"DateField": "2003-02-01T16:35:30Z"` Formula: `SECOND(%DateField%)` Return value: `30`

Omnistudio TIMEDIFF Function

Returns the difference in number of milliseconds between two specified times as an integer.

The second time is subtracted from the first time. If the second time is greater than the first time, the function returns a negative integer. If you specify a time zone, the function considers it in its calculation.

Signature

`TIMEDIFF(firstTime, secondTime)`

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>firstTime</code>	Datetime	Required	A time specified as a string in Day.js format . You can specify a date and time. If you specify just a date, the function uses <code>00:00:00:000</code> as the time.
<code>secondTime</code>	Datetime	Required	A time for which the function is to calculate the difference in number of milliseconds between the first time. Specify a time as a string in Day.js format . You can specify a date and time. If you specify just a date, the function uses <code>00:00:00:000</code> as the time.

-  **Positive Times Example** Formula: `TIMEDIFF("16:35:30", "08:00:15")` Return value: `30915000`

-  **Negative Times Example** Formula: `TIMEDIFF("T08:00:15", "T16:35:30")` Return value: `-30915000`

-  **Positive Dates Example** Formula: `TIMEDIFF("2001-02-01", "02/01/2000")` Return value: `31622400000`

 **Date and Time with Time Zone Example** Formula: `TIMEDIFF("03/01/2025T12:00:00+0000", "03/01/2025T12:00:00-0500")` Return value: `43200000`

JSON Object Examples

Sample data:

```
"Account": [
  {
    "Cases": [
      {
        "Case1": {
          "CreatedDate": "2/1/2024T16:35:30 GMT -0500 (EDT)",
          "LastUpdate": "2/8/2024T09:15:00 GMT -0500 (EDT)"
        }
      },
      {
        "Case2": {
          "CreatedDate": "2/2/2025T11:05:05 GMT -0500 (EDT)",
          "LastUpdate": "2/3/2025T15:50:57 GMT -0500 (EDT)"
        }
      }
    ]
  }
]
```

Formula: `TIMEDIFF(Account:Cases:Case2:CreatedDate, Account:Cases:Case2:LastUpdate)` Return value: `-86400000` Formula: `TIMEDIFF(%Account:Cases:Case2:LastUpdate%, %Account:Cases:Case2:CreatedDate%)` Return value: `86400000`

Omnistudio TODAY Function

Returns the current date.

The `TODAY` function always returns the current date in the format `"YYYY-MM-DD"`. By default, the function returns the date in Coordinated Universal Time (UTC). UTC is the same as Greenwich Mean Time (GMT). Use the `$Vlocity.NOW` environment variable to apply the time zone of the user or org.

You can use the `TODAY` function with other functions to specify a date relative to the current date.

Signature

`TODAY()`

Return Value

Datetime

Parameters

None

 **Today Example** Formula: `TODAY()` Return value: `"2025-01-15"`

 **First Day of Next Year Example** Formula: `CONCAT(YEAR(ADDDATE(TODAY(), 1)), "-01-01")`
Return value: `"2026-01-01"`

Omnistudio UNIXTODATETIME Function

Returns the date and time for a number of seconds or milliseconds specified in UNIX time.

UNIX time is the number of non-leap seconds that have passed since 00:00:00 on 1 January 1970, Coordinated Universal Time (UTC). UTC is the same as Greenwich Mean Time (GMT). The value is referred to as the UNIX epoch.

Signature

`UNIXTODATETIME(timestamp)`

Return Value

Datetime

Parameters

Parameter	Data Type	Necessity	Description
<code>timestamp</code>	Integer	Required	A positive integer that represents the number of seconds or milliseconds that have passed since the UNIX epoch. The value is referred to as a UNIX timestamp.

 **Timestamp in Seconds Example** Formula: `UNIXTODATETIME(1012581330)` Return value: `"2002-02-01T16:35:30.000Z"`

 **Timestamp in Milliseconds Example** Formula: `UNIXTODATETIME(1012581330000)` Return value: `"2002-02-01T16:35:30.000Z"`

Omnistudio YEAR Function

Returns the year for a specified date as an integer value.

Signature

`YEAR(date)`

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>date</code>	Datetime	Required	A date specified as a string in Day.js format . You can specify a date and time.

 **Date String Example** Formula: `YEAR("2025-01-01")` Formula: `YEAR("01/01/2025")` Formula: `YEAR("1/1/2025")` Return value: `2025`

 **JSON Object Example** Sample data:

```
"Account": [
  {
    "Cases": [
      {
        "Case1": {
          "CreatedDate": "2/1/2024T16:35:30 GMT -0500 (EDT)",
          "LastUpdate": "2/8/2024T09:15:00 GMT -0500 (EDT)"
        }
      },
      {
        "Case2": {
          "CreatedDate": "2/2/2025T11:05:05 GMT -0500 (EDT)",
          "LastUpdate": "2/3/2025T15:50:57 GMT -0500 (EDT)"
        }
      }
    ]
  }
]
```

Formula: `YEAR(Account:Cases:Case1:CreatedDate)` Return value: `2024` Formula:

```
YEAR(Account:Cases:Case2:CreatedDate) Return value: 2025
```

Omnistudio String Functions

Functions that operate on strings.

Omnistudio BASE64ENCODE Function

Encodes an input value into Base64 format.

Omnistudio CONCAT Function

Concatenates two or more strings into a single string.

Omnistudio JOIN Function

Joins two or more string into a single string, separating each string by a specified token.

Omnistudio MAXSTRING Function

Returns the string that is last lexicographically in a list of two or more strings.

Omnistudio SPLIT Function

Splits a string into substrings at each position of a specified token.

Omnistudio STRINGINDEXOF Function

Returns the start index of a substring within a string.

Omnistudio SUBSTRING Function

Returns a substring of a string based on specified start and end indexes.

Omnistudio TOSTRING Function

Converts input data into a string.

Omnistudio BASE64ENCODE Function

Encodes an input value into Base64 format.

Base64 is a common ASCII-based encoding. In an encoded result, trailing = characters are padding.

Signature

```
BASE64ENCODE (data)
```

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>data</code>	Any data type	Required	The input value to be encoded into Base64 format. The function does not accept a null value or an empty string.

👁️ **String Encoding Example** Formula: `BASE64ENCODE ("Encode this string.")` Return value: `"RW5jb2RlIHRoaXMgc3RyaW5nLg=="`

👁️ **Date Encoding Example** Formula: `BASE64ENCODE ("2003-02-01T16:35:30-0500")` Return value: `"MjAwMy0wMi0wMVQxNjozNTozMC0wNTAw"`

👁️ **Number Encoding Example** Formula: `BASE64ENCODE (1024)` Return value: `"MTAyNA=="`

👁️ **JSON Object Encoding Example** Sample data:

```
"Contact": {
    "FirstName": "Thomas",
    "MiddleName": "Alva",
    "LastName": "Edison"
}
```

Formula: `BASE64ENCODE (Contact)` Return value:

`"e0xhc3ROYW11PUVkaXNvbiwgTWlkZGx1TmFtZT1BbHZhLCBGaXJzdE5hbWU9VGhvbWFzfQ=="`

Omnistudio CONCAT Function

Concatenates two or more strings into a single string.

Signature

`CONCAT (string...)`

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>string...</code>	String	Required	<p>A comma-separated list of one or more strings to be concatenated into a single string. To separate input strings in the result, include a separator string (such as <code>" "</code>) between each pair of strings. You can concatenate numbers, which the function coerces into strings.</p> <p>If you specify only a single string, the function returns that string. If you specify multiple strings, the function ignores null values and empty strings. You cannot specify a null value or an empty string as the only input value.</p>

 **String and Number Concatenation Example** Formula: `CONCAT ("AGE", ":", 23)` Return value: `"AGE: 23"`

 **Number and String Concatenation Example** Formula: `CONCAT(23, " years of age")` Return value: `"23 years of age"`

 **JSON Object Concatenation Example** Sample data:

```
"stringField1": "abc",
"stringField2": "ABC"
```

Formula: `CONCAT(%stringField1%, " ", %stringField2%)` Return value: `"abc ABC"`

 **JSON Object Concatenation Example** Sample data:

```
"Contact": {
  "FirstName": "Mike",
  "LastName": "Smith"
}
```

Formula: `CONCAT(Contact:FirstName, " ", Contact:LastName)` Return value: `"Mike Smith"`

Omnistudio JOIN Function

Joins two or more string into a single string, separating each string by a specified token.

Signature

```
JOIN(string..., token)
```

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>string...</code>	String	Required	<p>A comma-separated list of one or more strings to be joined into a single string. In the joined string, each specified string is separated by the <code>token</code>. You can join numbers, which the function coerces into strings. If you specify only a single string and a token, the function returns only the string.</p> <p>If you specify multiple values and any are null, the function omits the null values from the joined string. If you specify multiple values and any are empty strings, the function joins only the values that precede the empty string. You cannot specify a null value or an empty string as the only input value.</p>
<code>token</code>	String	Required	<p>A value to be placed between each input string at the position at which the strings are joined. If you specify a number, the function coerces it into a string. If you specify an empty string, the function omits the empty string from the joined string. If you specify multiple input strings and no token, the function uses the last value of the input as the token.</p>

 **Alphabetic Array Example** Sample data: `"AlphabeticArray": ["a", "A", "b", "B", "c", "C"]` Formula: `JOIN(AlphabeticArray, ", ")` Return value: `"a, A, b, B, c, C"`

 **Numeric Array Example** Sample data: `"NumericArray": [1, 2, 3, 4]` Formula: `JOIN(NumericArray, " / ")` Return value: `"1 / 2 / 3 / 4"`

 **JSON Object Example** Sample data:

```
"Contact": {
    "FirstName": "Thomas",
    "MiddleName": "Alva",
    "LastName": "Edison"
}
```

Formula: `JOIN(Contact:FirstName, Contact:MiddleName, Contact:LastName, " ")`

Return value: `"Thomas Alva Edison"`

JSON Array Example Sample data:

```
"Contacts": [
    {
        "id": "0036ab",
        "lastName": "Jones",
        "firstName": "Cathy"
    },
    {
        "id": "2787kq",
        "lastName": "Smith",
        "firstName": "Albert"
    },
    {
        "id": "3610xr",
        "lastName": "Smith",
        "firstName": "Ben"
    }
]
```

Formula: `JOIN(Contacts:id, ', ')` Return value: `"0036ab, 2787kq, 3610xr"`

Alphabetic List and No Token Example Formula: `JOIN("a", "b", "c")` Formula: `JOIN(("a", "b", "c"))` Return value: `"acb"`

Omnistudio MAXSTRING Function

Returns the string that is last lexicographically in a list of two or more strings.

The function performs case-sensitive comparisons of the input strings.

Signature

`MAXSTRING(string...)`

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>string...</code>	String	Required	<p>Two or more strings from which the string that is last lexicographically is to be returned. If you specify numbers, the function coerces them into strings.</p> <p>If you specify only a single string, the function returns that string. If you specify multiple strings, the function ignores null values and empty strings. You cannot specify a null value or an empty string as the only input value.</p>

 **Maximum String Example** Formula: `MAXSTRING("Amy", "Ziggy", "Michael")` Return value: `"Ziggy"`

 **Maximum Number Example** Formula: `MAXSTRING(1, 2, 3)` Return value: `"3"`

 **Uppercase Characters Example** Formula: `MAXSTRING("A", "B", "C")` Return value: `"C"`

 **Uppercase and Lowercase Characters Example** Formula: `MAXSTRING("A", "b", "C")` Return value: `"b"`

Omnistudio SPLIT Function

Splits a string into substrings at each position of a specified token.

The function performs a case-sensitive search for the token. If it splits the input string into two or more substrings, the function returns an array of strings. Otherwise, it returns the input string.

Signature

`SPLIT(string, token)`

Return Value

String[]

Parameters

Parameter	Data Type	Necessity	Description
<code>string</code>	String	Required	The string to be split into substrings. If you specify a number, the function coerces it into a string. The string cannot be null or an empty string.
<code>token</code>	String	Required	A string that specifies the position at which the input string is to be split into substrings. The function splits the string at each occurrence of the token. It does not include the token in the substrings that it returns. If you specify a number, the function coerces it into a string. The function returns the input string if the token is not present in the string. If you specify a null token, the function returns the input string. If you specify an empty string as the token, the function returns an array that includes each character of the input value.

 **Split String Example** Sample data: `%Name% == "Anne Marie Gupta"` Formula: `SPLIT(%Name%, " ")` Return value: `["Anne", "Marie", "Gupta"]`

 **Split Number Example** Formula: `SPLIT(12345, 3)` Return value: `["12", "45"]`

 **Token Not Found Example** Formula: `SPLIT("abcde", "C")` Return value: `"abcde"`

Omnistudio STRINGINDEXOF Function

Returns the start index of a substring within a string.

The function performs a case-sensitive search for the substring. The position of the first character in the string is `0`. If the substring occurs multiple times in the string, the function returns the index of the first occurrence. If it doesn't find the substring in the string, the function returns `-1`.

Signature

```
STRINGINDEXOF(string, substring)
```

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>string</code>	String	Required	The string in which to search for the <code>substring</code> . If you specify a number, the function coerces it into a string. The string cannot be null or an empty string.
<code>substring</code>	String	Required	The substring for which to search in the <code>string</code> . If you specify a number, the function coerces it into a string. The substring cannot be null or an empty string.

👁️ **Substring Found Example** Formula: `STRINGINDEXOF("This is the test string.", "test")` Return value: `12`

👁️ **Substring Not Found Example** Formula: `STRINGINDEXOF("This is the test string.", "testy")` Return value: `-1`

👁️ **Multiple Occurrences of Substring Example** Formula: `STRINGINDEXOF("The string of strings.", "string")` Return value: `4`

👁️ **Numeric Values Example** Formula: `STRINGINDEXOF(1234, 34)` Return value: `2`

Omnistudio SUBSTRING Function

Returns a substring of a string based on specified start and end indexes.

The position of the first character in the `string` is `0`. The position of the last character in the `string` is the length of the string. The function performs a case-sensitive search for a character or string used as a start or end index.

Signature

```
SUBSTRING(string, startIndex, endIndex)
```

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>string</code>	String	Required	The string from which a substring is to be returned. The string cannot be null or an empty string.
<code>startIndex</code>	Integer or string	Optional	<p>The position of the first character in the substring. The start index is inclusive, so the position of the first character is <code>0</code>.</p> <ul style="list-style-type: none"> If you specify a negative integer or an integer that is greater than the length of the string, the function uses <code>0</code>. If you specify a single character, the function uses the position of the first occurrence of that character. If you specify a string, the function uses the position of the first character of the first occurrence of the string. If you specify null or an empty string, the function uses <code>0</code>. <p>If you omit a start index, the function uses <code>0</code> by default. In this case, you must also omit the end index.</p>
<code>endIndex</code>	Integer or string	Optional	<p>The position of the last character in the substring. The end index is exclusive, so the position of the last character is the length of the string.</p> <ul style="list-style-type: none"> If you specify a negative integer or an integer that is greater than the length of the string, the function uses the length of the string. If you specify a single character, the function uses the position of the first occurrence of that character minus one. If you specify a string, the function uses the position of the first character of the first occurrence of the string minus one. If you specify null or an empty string, the function uses the length of the string.

Parameter	Data Type	Necessity	Description
			If you omit an end index, the function uses the length of the string by default. If you specify an end index, you must also specify a start index.

- 🕒 **No Start or End Index Example** Sample data: `"InputString": "The string."` Formula: `SUBSTRING(InputString)` Return value: `"The string."`
- 🕒 **Numeric Start Index Only Example** Sample data: `"InputString": "The string."` Formula: `SUBSTRING(InputString, 4)` Return value: `"string."`
- 🕒 **Numeric Start and End Indexes Example** Sample data: `"InputString": "The string."` Formula: `SUBSTRING(InputString, 0, 11)` Return value: `"The string."`
- 🕒 **Numeric Start and End Indexes Example** Sample data: `"InputString": "The string."` Formula: `SUBSTRING(InputString, 0, 4)` Return value: `"The "`
- 🕒 **Numeric Start and End Indexes Example** Sample data: `"InputString": "The string."` Formula: `SUBSTRING(InputString, 4, 10)` Return value: `"string"`
- 🕒 **Character Start Index Only Example** Sample data: `"InputString": "The string."` Formula: `SUBSTRING(InputString, "r")` Return value: `"ring."`
- 🕒 **Character Start and End Indexes Example** Sample data: `"InputString": "The string."` Formula: `SUBSTRING(InputString, "r", ".")` Return value: `"ring"`
- 🕒 **String Start Index Only Example** Sample data: `"InputString": "The string."` Formula: `SUBSTRING(InputString, "string")` Return value: `"string."`
- 🕒 **String Start and End Indexes Example** Sample data: `"InputString": "The string."` Formula: `SUBSTRING(InputString, "The", "ring.")` Return value: `"The st"`
- 🕒 **String Start and End Indexes Example** Formula: `SUBSTRING("The string of strings.", "The", "ring")` Return value: `"The st"`

Omnistudio TOSTRING Function

Converts input data into a string.

Signature

```
TOSTRING(data)
```

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>data</code>	Any data type	Required	The data to be converted into a string. The function does not accept a null value or an empty string.

- 👁️ **Single Number Example** Formula: `TOSTRING(3.0)` Return value: `"3.0"`
- 👁️ **JSON Object Example** Formula: `TOSTRING({ "key": "value" })` Return value: `"key, value"`
- 👁️ **JSON Object Example** Formula: `TOSTRING({ "Amount": 750.00 })` Return value: `"Amount, 750.00"`
- 👁️ **Quoted JSON Object Example** Formula: `TOSTRING(' { "key": "value" } ')` Return value: `"{ \"key\": \"value\" }"`
- 👁️ **JSON Array Example** Formula: `TOSTRING([{ "key1": "value1" }, { "key2": "value2" }])` Return value: `"key1, value1, key2, value2"`

Omnistudio List and Array Functions

Functions that operate on lists and arrays.

Omnistudio AVG Function

Returns the average of a list or a JSON array of numeric values.

Omnistudio FILTER Function

Filters a list of JSON objects by a specified condition and returns a JSON list of matching objects.

Omnistudio LIST Function

Converts a list or a named JSON array to an anonymous JSON array.

Omnistudio LISTMERGE Function

Merges nodes from two or more JSON lists of objects into a single JSON list based on the values of one or more merge keys.

Omnistudio LISTMERGEPRIMARY Function

Augments the nodes of a primary JSON list of objects with data from one or more other JSON lists based on the values of one or more merge keys.

Omnistudio LISTSIZE Function

Returns the number of items in a list or a JSON array.

Omnistudio MAPTOLIST Function

Converts a JSON object or a hierarchy of objects to a JSON list.

Omnistudio MAX Function

Returns the maximum value from a list or a JSON array of numeric values.

Omnistudio MIN function

Returns the minimum value from a list or a JSON array of numeric values.

Omnistudio SORTBY Function

Sorts a JSON list of objects by specified keys and returns a sorted JSON list. (Data Mappers only)

Omnistudio SUM function

Returns the sum of a list or a JSON array of numeric values.

Omnistudio AVG Function

Returns the average of a list or a JSON array of numeric values.

Signature

`AVG(list)`

Return Value

Number

Parameters

Parameter	Data Type	Necessity	Description
<i>list</i>	List of numbers	Required	A comma-separated list of numeric values or a list of numeric values from a JSON array.

 **List Example** Formula: `AVG(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)` Return value: `5.5`

 **JSON Array Example** Sample data:

```
"List": [
  {
    "Item": 3.5
  },
  {
    "Item": 4.25
  },
]
```

```
{
  "Item": 5.75
}
```

Formula: `AVG(List:Item)` Formula: `AVG(%List:Item%)` Return value: `4.5`

Omnistudio FILTER Function

Filters a list of JSON objects by a specified condition and returns a JSON list of matching objects.

The function performs case-insensitive matching of strings.

Signature

`FILTER(LIST(list), condition)`

Return Value

JSON list

Parameters

Parameter	Data Type	Necessity	Description
<code>LIST(list)</code>	JSON list	Required	An anonymous JSON array of objects to be filtered. Use the <code>LIST</code> function to pass a JSON array.
<code>condition</code>	Expression	Required	An equivalence expression that identifies the node to filter on and the value the node must have to qualify as a match. Specify the value as a string or a variable. You must enclose the entire expression in quotes and also quote a string value within the expression.

Filter by String Example Sample data:

```
"NameList": [
  {
    "FirstName": "Zellie",
```

```

    "LastName": "Xavier"
},
{
  "FirstName": "Aaron",
  "LastName": "Xavier"
},
{
  "FirstName": "Mike",
  "LastName": "Smith"
}
]

```

Formula: `FILTER(LIST(NameList), 'LastName == "Xavier")` **Formula:** `FILTER(LIST(NameList), "LastName == 'Xavier')")` **Return value:**

```

[
{
  "FirstName": "Aaron",
  "LastName": "Xavier"
},
{
  "FirstName": "Zellie",
  "LastName": "Xavier"
}
]

```



Filter by Variable Example

Sample data:

```

"NameList": [
  {
    "FirstName": "Zellie",
    "LastName": "Xavier"
  },
  {
    "FirstName": "Aaron",
    "LastName": "Xavier"
  },
  {
    "FirstName": "Mike",
    "LastName": "Smith"
  }
],
"FindName": "Xavier"

```

Formula: `FILTER(LIST(NameList), 'LastName == "' + FindName + '"')` Formula:
`FILTER(LIST(NameList), "LastName == '" + FindName + "')")` Return value:

```
[  
 {  
   "FirstName": "Aaron",  
   "LastName": "Xavier"  
 },  
 {  
   "FirstName": "Zellie",  
   "LastName": "Xavier"  
 }  
 ]
```

Omnistudio LIST Function

Converts a list or a named JSON array to an anonymous JSON array.

Use the `LIST` function with other functions that require an anonymous JSON array (a JSON list) as input: `FILTER`, `LISTMERGE`, `LISTMERGEPRIMARY`, `SERIALIZE`, `SORTBY`, and a custom `FUNCTION` that passes a JSON list as an argument.

Signature

`LIST(expression)`

Return Value

JSON list

Parameters

Parameter	Data Type	Necessity	Description
<code>expression</code>	List or named JSON array	Required	<p>An expression that resolves to a list of values or JSON objects or to a named JSON array of one or more values or objects. Each object can contain one or more nodes or key-value pairs.</p> <p>If you pass it an anonymous JSON array, the function returns an array with a single <code>VLOCITY-FORMULA-LIST</code> object that contains that</p>

Parameter	Data Type	Necessity	Description
			anonymous array.

 **List of Alphabetic Values Example** Formula: `LIST("a", "b", "c")` Return value: `["a", "b", "c"]`

 **List of Numeric Values Example** Formula: `LIST(1, 2, 2, 3, 3, 4)` Return value: `[1, 2, 2, 3, 3, 4]`

 **JSON Object Example** Sample data:

```
"Contact": {
    "FirstName": "Mike",
    "LastName": "Smith"
}
```

Formula: `LIST(Contact)` Return value:

```
[
{
    "LastName": "Smith",
    "FirstName": "Mike"
}]
```

 **Named JSON Array Example** Sample data:

```
"NameList": [
    {
        "FirstName": "Zellie",
        "LastName": "Xavier"
    },
    {
        "FirstName": "Aaron",
        "LastName": "Xavier"
    },
    {
        "FirstName": "Mike",
        "LastName": "Smith"
    }
]
```

Formula: `LIST(NameList)` Return value:

```
[  
  {  
    "FirstName": "Zellie",  
    "LastName": "Xavier"  
  },  
  {  
    "FirstName": "Aaron",  
    "LastName": "Xavier"  
  },  
  {  
    "FirstName": "Mike",  
    "LastName": "Smith"  
  }  
]
```



Named JSON Array Example Sample data:

```
"Account": [  
  {  
    "Cases": [  
      {  
        "Case1": {  
          "CreatedDate": "2/1/2004T16:35:30 GMT -0500 (EDT)",  
          "LastUpdate": "2/8/2024T09:15:00 GMT -0500 (EDT)"  
        }  
      },  
      {  
        "Case2": {  
          "CreatedDate": "2/2/2004T11:05:05 GMT -0500 (EDT)",  
          "LastUpdate": "2/3/2024T15:50:57 GMT -0500 (EDT)"  
        }  
      }  
    ]  
  }  
]
```

Formula: `LIST(Account:Cases)` Return value:

```
[  
  {  
    "Case1": {  
      "CreatedDate": "2/1/2024T16:35:30 GMT -0500 (EDT)",  
      "LastUpdate": "2/8/2024T09:15:00 GMT -0500 (EDT)"  
    }  
  }  
]
```

```
        }
    },
{
  "Case2": {
    "CreatedDate": "2/2/2025T11:05:05 GMT -0500 (EDT)",
    "LastUpdate": "2/3/2025T15:50:57 GMT -0500 (EDT)"
  }
}
]
```

Omnistudio LISTMERGE Function

Merges nodes from two or more JSON lists of objects into a single JSON list based on the values of one or more merge keys.

The function combines nodes from the specified anonymous JSON arrays when the values of the named merge keys match. The resulting list contains all of the nodes and keys from all of the merged lists. If lists contain nodes with identical merge keys but different values, values from later nodes overwrite values from earlier nodes. The function performs case-insensitive matching of merge keys.

Signature

```
LISTMERGE(mergeKey..., LIST(list)...)
```

Return Value

JSON list

Parameters

Parameter	Data Type	Necessity	Description
<i>mergeKey...</i>	String	Required	A comma-separated list of one or more keys. The function combines nodes from the lists when the values of their merge keys match.
LIST(<i>list</i>). ..	JSON list	Required	Two or more anonymous JSON arrays to be merged. Use the LIST function to pass each JSON array. If you pass only a single array, the function returns that array.

 **Merge Lists Example** Sample data:

```
"ContactNames": [
  {
    "id": "0036ab",
    "lastName": "Jones",
    "firstName": "Cathy"
  },
  {
    "id": "2787kq",
    "lastName": "Smith",
    "firstName": "Albert"
  },
  {
    "id": "3610xr",
    "lastName": "Smith",
    "firstName": "Ben"
  }
],
"ContactAddresses": [
  {
    "id": "0036ab",
    "city": "Phoenix",
    "state": "AZ"
  },
  {
    "id": "2787kq",
    "city": "San Francisco",
    "state": "CA"
  },
  {
    "id": "3610xr",
    "city": "Portland",
    "state": "OR"
  }
],
"ContactBirthdates": [
  {
    "id": "0036ab",
    "birthdate": "1976-10-04"
  },
  {
    "id": "2787kq",
    "birthdate": "1973-10-01"
  }
]
```

```

    },
    {
      "id": "3610xr",
      "birthdate": "1974-10-02"
    }
  ]

```

Formula: `LISTMERGE("id", LIST(ContactNames), LIST(ContactAddresses), LIST(ContactBirthdates))` **Return value:**

```

[
  {
    "id": "0036ab",
    "lastName": "Jones",
    "firstName": "Cathy",
    "city": "Phoenix",
    "state": "AZ",
    "birthdate": "1976-10-04"
  },
  {
    "id": "2787kq",
    "lastName": "Smith",
    "firstName": "Albert",
    "city": "San Francisco",
    "state": "CA",
    "birthdate": "1973-10-01"
  },
  {
    "id": "3610fw",
    "lastName": "Smith",
    "firstName": "Ben",
    "city": "Portland",
    "state": "OR",
    "birthdate": "1974-10-02"
  }
]

```

Omnistudio LISTMERGEPRIMARY Function

Augments the nodes of a primary JSON list of objects with data from one or more other JSON lists based on the values of one or more merge keys.

The function augments the nodes from the first (primary) specified anonymous JSON array with data

from all other specified anonymous JSON arrays when the values of the named merge keys match. The resulting list contains all of the nodes and keys from the first list along with keys from the other matching lists. If nodes from later lists with identical merge keys have different values from the first list, values from later nodes overwrite values from earlier nodes. The function performs case-insensitive matching of merge keys.

For example, use the function when you have a list of qualified products that you want to augment with data from related lists. The function does not include products from later lists that do appear in the primary list.

Signature

```
LISTMERGEPRIMARY (mergeKey..., LIST(list) ...)
```

Return Value

JSON list

Parameters

Parameter	Data Type	Necessity	Description
<code>mergeKey...</code>	String	Required	A comma-separated list of one or more keys. The function augments nodes from the first list with keys from the other lists when the values of their merge keys match.
<code>LIST(list)...</code>	JSON list	Required	A primary anonymous JSON array to be extended with data from one or more other anonymous JSON arrays. The function includes only nodes from the first list in the list that it returns. Use the <code>LIST</code> function to pass each JSON array. If you pass only a primary array, the function returns that array.

Primary Merge List Example Sample data:

```
"ContactNames": [
  {
    "id": "0036ab",
    "lastName": "Jones",
    "firstName": "Cathy"
  },
  {
    "id": "0036ab",
    "lastName": "Jones",
    "firstName": "Cathy"
  }
]
```

```
{  
    "id": "3610xr",  
    "lastName": "Smith",  
    "firstName": "Ben"  
}  
,  
"ContactAddresses": [  
    {  
        "id": "0036ab",  
        "city": "Phoenix",  
        "state": "AZ"  
},  
    {  
        "id": "7723hw",  
        "lastName": "Sacramento",  
        "firstName": "CA"  
},  
    {  
        "id": "2787kq",  
        "city": "San Francisco",  
        "state": "CA"  
},  
    {  
        "id": "3610xr",  
        "city": "Portland",  
        "state": "OR"  
}  
,  
"ContactBirthdates": [  
    {  
        "id": "0036ab",  
        "birthdate": "1976-10-04"  
},  
    {  
        "id": "7723hw",  
        "birthdate": "1974-07-04"  
},  
    {  
        "id": "2787kq",  
        "birthdate": "1973-10-01"  
},  
    {  
        "id": "3610xr",  
}
```

```
        "birthdate": "1974-10-02"
    }
]
```

Formula: `LISTMERGEPRIMARY("id", LIST(ContactNames), LIST(ContactAddresses), LIST(ContactBirthdates))` **Return value:**

```
[
{
    "id": "0036ab",
    "lastName": "Jones",
    "firstName": "Cathy",
    "city": "Phoenix",
    "state": "AZ",
    "birthdate": "1976-10-04"
},
{
    "id": "3610fw",
    "lastName": "Smith",
    "firstName": "Ben",
    "city": "Portland",
    "state": "OR",
    "birthdate": "1974-10-02"
}
]
```

Omnistudio LISTSIZE Function

Returns the number of items in a list or a JSON array.

To test for an empty array, use the `ISBLANK` function.

Signature

```
LISTSIZE(list)
```

Return Value

Number

Parameters

Parameter	Data Type	Necessity	Description
<i>list</i>	List or JSON array	Required	A comma-separated list of values or a JSON array of values or nodes.

 **List Example** Formula: `LISTSIZE(6, 7, 8, 9, 10)` Return value: `5`

 **Empty List Example** Formula: `LISTSIZE()` Formula: `LISTSIZE(())` Return value: `0`

 **JSON Array Example** Sample data:

```
"NameList": [
  {
    "FirstName": "Zellie",
    "LastName": "Xavier"
  },
  {
    "FirstName": "Aaron",
    "LastName": "Xavier"
  },
  {
    "FirstName": "Mike",
    "LastName": "Smith"
  }
]
```

Formula: `LISTSIZE(NameList)` Return value: `3`

 **Empty Array Example** Sample data: `"EmptyList": []` Formula: `LISTSIZE([])` Formula: `LISTSIZE(EmptyList)` Return value: `1` Formula: `IF(ISBLANK(EmptyList), 0, LISTSIZE(EmptyList))` Return value: `0`

Omnistudio MAPTOLIST Function

Converts a JSON object or a hierarchy of objects to a JSON list.

Signature

`MAPTOLIST(jsonObject)`

Return Value

JSON list

Parameters

Parameter	Data Type	Necessity	Description
<code>jsonObject</code>	JSON object	Required	A node or node path of a JSON object to convert to a JSON list. Each node can contain one or more nodes or key-value pairs.

JSON Object Example Sample data:

```
"Contact": {
    "FirstName": "William",
    "MiddleName": "Leslie",
    "LastName": "Edison",
    "Parents": {
        "Father": {
            "FirstName": "Thomas",
            "MiddleName": "Alva",
            "LastName": "Edison"
        },
        "Mother": {
            "FirstName": "Mary",
            "MiddleName": "Stilwell",
            "LastName": "Edison"
        }
    }
}
```

Formula: `MAPTOLIST(Contact)` Return value:

```
[
{
    "LastName": "Edison"
},
{
    "MiddleName": "Leslie"
},
{
    "FirstName": "William"
}
```

```

},
{
  "Parents": {
    "Father": {
      "MiddleName": "Alva",
      "LastName": "Edison",
      "FirstName": "Thomas"
    },
    "Mother": {
      "MiddleName": "Stilwell",
      "LastName": "Edison",
      "FirstName": "Mary"
    }
  }
}
]

```

Formula: `MAPTOLIST (Contact:Parents)` **Return value:**

```

[
{
  "Mother": {
    "MiddleName": "Stilwell",
    "LastName": "Edison",
    "FirstName": "Mary"
  }
},
{
  "Father": {
    "MiddleName": "Alva",
    "LastName": "Edison",
    "FirstName": "Thomas"
  }
}
]

```

Formula: `MAPTOLIST (Contact:Parents:Mother)` **Return value:**

```

[
{
  "LastName": "Edison"
},
{
  "MiddleName": "Stilwell"
}
]

```

```

    },
{
  "FirstName": "Mary"
}
]
```

Omnistudio MAX Function

Returns the maximum value from a list or a JSON array of numeric values.

Signature

`MAX(list)`

Return Value

Number

Parameters

Parameter	Data Type	Necessity	Description
<code><i>list</i></code>	List of numbers	Required	A comma-separated list of numeric values or a list of numeric values from a JSON array.

 **List Example** Formula: `MAX(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)` Return value: `10`

 **JSON Array Example** Sample data:

```

"List": [
  {
    "Item": 3
  },
  {
    "Item": 4
  },
  {
    "Item": 5
  }
]
```

Formula: `MAX(List:Item)` Return value: `5`

Omnistudio MIN function

Returns the minimum value from a list or a JSON array of numeric values.

Signature

```
MIN(list)
```

Return Value

Number

Parameters

Parameter	Data Type	Necessity	Description
<i>list</i>	List of numbers	Required	A comma-separated list of numeric values or a list of numeric values from a JSON array.

 **List Example** Formula: `MIN(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)` Return value: `1`

 **JSON Array Example** Sample data:

```
"List": [
  {
    "Item": 3
  },
  {
    "Item": 4
  },
  {
    "Item": 5
  }
]
```

Formula: `MIN(List:Item)` Return value: `3`

Omnistudio SORTBY Function

Sorts a JSON list of objects by specified keys and returns a sorted JSON list. (Data Mappers only)

The function performs case-insensitive matching of keys and sorting of values.

Signature

```
SORTBY(LIST(list), key..., [:DSC])
```

Return Value

JSON list

Parameters

Parameter	Data Type	Necessity	Description
<code>LIST(list)</code>	JSON list	Required	An anonymous JSON array of objects to be sorted. Use the <code>LIST</code> function to pass a JSON array.
<code>key...</code>	String	Required	One or more keys by which the JSON objects are to be sorted.
<code>[:DSC]</code>	String	Optional	Sorts the results in descending order. You must enclose the value in single or double quotes, for example, ' <code>[:DSC]</code> '. Omit the parameter to sort the results in ascending order by default.



Sort List in Ascending Order Example

Sample data:

```
"NameList": [
    {
        "FirstName": "Aaron",
        "LastName": "Xavier"
    },
    {
        "FirstName": "Zellie",
        "LastName": "Xavier"
    },
    {
        "FirstName": "Mike",
        "LastName": "Smith"
    }
]
```

Formula: `SORTBY(LIST(NameList), 'LastName', 'FirstName')` Return value:

```
[
```

```
{
  "FirstName": "Mike",
  "LastName": "Smith"
},
{
  "FirstName": "Aaron",
  "LastName": "Xavier"
},
{
  "FirstName": "Zellie",
  "LastName": "Xavier"
}
]
```



Sort List in Descending Order Example

Sample data:

```
"NameList": [
  {
    "FirstName": "Aaron",
    "LastName": "Xavier"
  },
  {
    "FirstName": "Zellie",
    "LastName": "Xavier"
  },
  {
    "FirstName": "Mike",
    "LastName": "Smith"
  }
]
```

Formula: `SORTBY(LIST(NameList), 'LastName', 'FirstName', '[:DSC]')` Return value:

```
[
  {
    "FirstName": "Zellie",
    "LastName": "Xavier"
  },
  {
    "FirstName": "Aaron",
    "LastName": "Xavier"
  },
  {
    "FirstName": "Mike",
    "LastName": "Smith"
  }
]
```

```
    "FirstName": "Mike",
    "LastName": "Smith"
}
```

Omnistudio SUM function

Returns the sum of a list or a JSON array of numeric values.

Signature

```
SUM(list)
```

Return Value

Number

Parameters

Parameter	Data Type	Necessity	Description
<i>list</i>	List of numbers	Required	A comma-separated list of numeric values or a list of numeric values from a JSON array.

 **List Example** Formula: `SUM(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)` Return value: `55`

 **JSON Array Example** Sample data:

```
"List": [
  {
    "Item": 3.5
  },
  {
    "Item": 4.25
  },
  {
    "Item": 5.75
  }
]
```

Formula: `SUM(List:Item)` Formula: `SUM(%List:Item%)` Return value: `13.5`

Omnistudio JSON Object Functions

Functions that operate on JSON objects.

Omnistudio DESERIALIZE Function

Converts a JSON string into a JSON object.

Omnistudio RESERIALIZE Function

Reserializes a previously serialized JSON string.

Omnistudio SERIALIZED Function

Converts a JSON object into a JSON string.

Omnistudio VALUELOOKUP Function

Returns the value of a node that exists at any depth of a JSON object hierarchy.

Omnistudio DESERIALIZE Function

Converts a JSON string into a JSON object.

Deserialization converts a serialized JSON string into its JSON object structure. Serialization and deserialization promote the efficient exchange of data between different systems and applications.

Signature

```
DESERIALIZE(jsonString)
```

Return Value

JSON object

Parameters

Parameter	Data Type	Necessity	Description
<i>jsonString</i>		Required	The previously serialized JSON string to be deserialized.

 **JSON Object Example** Sample data: "SerializedContact":

```
"{\\"LastName\\":\\"Edison\\",\\"MiddleName\\":\\"Alva\\",\\"FirstName\\":\\"Thomas\\"}"
```

Formula: `DESERIALIZE(%SerializedContact%)` Return value:

```
{  
  "MiddleName": "Alva",  
  "LastName": "Edison",  
}
```

```
    "FirstName": "Thomas"
}
```

**JSON Array Example** Sample data: `"SerializedContacts":`

```
"[{"lastName": "Jones", "firstName": "Cathy"}, {"lastName": "Smith", "firstName": "Thomas"}]
```

Formula: `DESERIALIZE(%SerializedContacts%)` Return value:

```
[
  {
    "firstName": "Cathy",
    "lastName": "Jones"
  },
  {
    "firstName": "Albert",
    "lastName": "Smith"
  },
  {
    "firstName": "Ben",
    "lastName": "Smith"
  }
]
```

Omnistudio RESerialize Function

Reserializes a previously serialized JSON string.

The `RESerialize` function is equivalent to calling the `SERIALIZE(DESERIALIZE())` functions. The function converts a JSON string into a generic `Map<String, Object>` format. It is useful in remote actions for converting Apex class output to a format that Data Mappers and Integration Procedures can accept. For more information, see [Remote Action for Integration Procedures](#)

Signature

```
RESerialize(jsonString)
```

Return Value

JSON string

Parameters

Parameter	Data Type	Necessity	Description
<code>jsonString</code>	JSON string	Required	The previously serialized JSON string to be reserialized.



JSON Object Example

Sample data: `"SerializedContact":`

```
"{\\"LastName\\":\\"Edison\\",\\"MiddleName\\":\\"Alva\\",\\"FirstName\\":\\"Thomas\\"}"
```

Formula: `RESERIALIZE(%SerializedContact%)` Return value:

```
"{\\"LastName\\":\\"Edison\\",\\"MiddleName\\":\\"Alva\\",\\"FirstName\\":\\"Thomas\\"}"
```



JSON Array Example

Sample data: `"SerializedContacts":`

```
"[{\\"lastName\\":\\"Jones\\",\\"firstName\\":\\"Cathy\\"},{\\"lastName\\":\\"Smith\\",\\"firstName\\":\\"John\\"}]"
```

Formula: `DESERIALIZE(%SerializedContacts%)` Return value:

```
[{\\"lastName\\":\\"Jones\\",\\"firstName\\":\\"Cathy\\"},{\\"lastName\\":\\"Smith\\",\\"firstName\\":\\"John\\"}]
```

Omnistudio SERIALIZED Function

Converts a JSON object into a JSON string.

Serialization converts a JSON object into a JSON string that can be easily stored, transmitted, and restored to its original object structure. Serialization and deserialization promote the efficient exchange of data between different systems and applications. (The results of the `SERIALIZED` function are different from the results of the `TOSTRING` function.)

Signature

```
SERIALIZED(jsonObject)
```

Return Value

JSON string

Parameters

Parameter	Data Type	Necessity	Description
<code>jsonObject</code>	JSON object	Required	The JSON object to be serialized. Use the <code>LIST</code> function to pass a JSON array.



JSON Object Example

Sample data:

```
"Contact": {
    "FirstName": "Thomas",
    "MiddleName": "Alva",
    "LastName": "Edison"
}
```

Formula: `SERIALIZE(%Contact%)` **Return value:**

```
"\\"LastName\\":\\"Edison\\",\\"MiddleName\\":\\"Alva\\",\\"FirstName\\":\\"Thomas\\"}"
```

JSON Array Example Sample data:

```
"Contacts": [
    {
        "lastName": "Jones",
        "firstName": "Cathy"
    },
    {
        "lastName": "Smith",
        "firstName": "Albert"
    },
    {
        "lastName": "Smith",
        "firstName": "Ben"
    }
]
```

Formula: `SERIALIZE(LIST(%Contacts%))` **Return value:**

```
"[{\\"lastName\\":\\"Jones\\",\\"firstName\\":\\"Cathy\\"}, {\\"lastName\\":\\"Smith\\",\\"firstName\\":\\"Albert\\"}, {\\"lastName\\":\\"Smith\\",\\"firstName\\":\\"Ben\\"}]"
```

Omnistudio VALUELOOKUP Function

Returns the value of a node that exists at any depth of a JSON object hierarchy.

The function returns the value of a JSON node referred to by another JSON node. The function lets you retrieve a specified node dynamically.

Signature

```
VALUELOOKUP(startNode, node...)
```

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>startNode</code>	JSON node	Required	A node or node path of a JSON object. The node or node path must begin at the root of the JSON object from which values are to be returned. For example, for the <code>Name</code> object of a <code>Contact</code> object, you can specify <code>Contact</code> , <code>Name</code> or <code>Contact:Name</code> . Use a node path to traverse a JSON array.
<code>node...</code>	JSON node	Required	A comma-separated list of one or more nodes of a JSON object. Each node must either be an immediate child of the previously specified node or refer to a child of that node. The final node must identify the node whose value is to be returned.

 **JSON Object Example** Sample data:

```
"Contact": {
    "Name": {
        "FirstName": "Thomas",
        "MiddleName": "Alva",
        "LastName": "Edison"
    },
    "Address": {
        "City": "San Francisco",
        "State": "CA",
        "ZipCode": 94110
    }
},
"GetNameGroup": "Name",
"GetLastNameField": "LastName",
"GetAddressGroup": "Address",
"GetStateField": "State"
```

Formula: `VALUELOOKUP(Contact, GetNameGroup, GetLastNameField)` Formula:
`VALUELOOKUP(Contact:Name, GetLastNameField)` Return value: `"Edison"` Formula:
`VALUELOOKUP(Contact, GetNameGroup)` Return value:

```
{
```

```
        "MiddleName": "Alva",
        "LastName": "Edison",
        "FirstName": "Thomas"
    }
```

Formula: `VALUELOOKUP(Contact, GetAddressGroup, GetStateField)` Formula:

`VALUELOOKUP(Contact:Address, GetStateField)` Return value: `"CA"` Formula:

`VALUELOOKUP(Contact, GetAddressGroup)` Return value:

```
{
    "City": "San Francisco",
    "ZipCode": 94110,
    "State": "CA"
}
```



JSON Array Example

Sample data:

```
"AccountList": [
    {
        "CaseList": [
            {
                "Case1": {
                    "CreatedDate": "2/1/2024T16:35:30 GMT -0500 (EDT)",
                    "LastUpdate": "2/8/2024T09:15:00 GMT -0500 (EDT)",
                    "NextCase": "Case2"
                }
            },
            {
                "Case2": {
                    "CreatedDate": "2/2/2025T11:05:05 GMT -0500 (EDT)",
                    "LastUpdate": "2/3/2025T15:50:57 GMT -0500 (EDT)",
                    "NextCase": "Case3"
                }
            },
            {
                "Case3": {
                    "CreatedDate": "2/4/2025T11:05:05 GMT -0500 (EDT)",
                    "LastUpdate": "2/6/2025T15:50:57 GMT -0500 (EDT)",
                    "NextCase": null
                }
            }
        ]
    }
]
```

```
],  
"GetFirstCase": "Case1",  
"GetNextCase": "NextCase"
```

Formula: `IF(GetFirstCase, GetFirstCase, "Empty list")` Return value: "Case1"

Formula: `VALUELOOKUP(AccountList:CaseList:Case1, GetNextCase)` Return value:

"Case2" Formula: `VALUELOOKUP(AccountList:CaseList:Case2, GetNextCase)` Return

value: "Case3"

Formula:

```
IF(VALUELOOKUP(AccountList:CaseList:Case3, GetNextCase),  
    VALUELOOKUP(AccountList:CaseList:Case3, GetNextCase),  
    "End of list")
```

Return value: "End of list"

Omnistudio Invocation Functions

Functions that invoke or are related to invoking other functions and operations.

[Omnistudio COUNTQUERY Function](#)

Executes a SOQL query and returns the number of matching rows.

[Omnistudio FUNCTION Function](#)

Executes a custom function defined as a method of an Apex class.

[Omnistudio GENERATEGLOBALKEY Function](#)

Generates a global key for use as an Id in a data pack.

[Omnistudio QUERY Function](#)

Executes a SOQL query and returns a JSON list of results.

[Omnistudio GLOBALAUTONUMBER Function](#)

Generates a unique number as per the configurations you define through the Omni Global Auto Number page.

Omnistudio COUNTQUERY Function

Executes a SOQL query and returns the number of matching rows.

Pass a single SOQL query to the function. The query can retrieve data from only a single column (a single field).

Signature

```
COUNTQUERY(query)
```

Return Value

Integer

Parameters

Parameter	Data Type	Necessity	Description
<code>query</code>	SOQL Query	Required	A string that contains a valid SOQL query. In the <code>WHERE</code> clause, you can pass a string directly or use ' <code>{0}</code> ' to refer to an additional value passed as a string, a JSON key, or an expression.

Count Query Example Sample Account object data from database:

```
{
    "Account Name": "Acme Shipping",
    "Account Number": "CD451796",
    "Billing State": "GA",
    "Billing Zip": 27215
},
{
    "Account Name": "GenePoint",
    "Account Number": "CC978213",
    "Billing State": "CA",
    "Billing Zip": 94043
},
{
    "Account Name": "Edge Communications",
    "Account Number": "CD451796",
    "Billing State": "TX",
    "Billing Zip": 78767
},
{
    "Account Name": "Express Logistics",
    "Account Number": "CD736025",
    "Billing State": "MO",
    "Billing Zip": 63101
},
{
    "Account Name": "Pyramid Construction",
    "Account Number": "CD112262",
    "Billing State": "CA",
}
```

```

    "Billing Zip": 94087
}

```

Sample data:

```

"InputStateWest": "CA",
"InputStateEast": "PA"

```

Formula: COUNTQUERY("SELECT Name FROM Account WHERE BillingState = 'CA'")
 Formula: QUERY("SELECT Name FROM Account WHERE BillingState LIKE '{0}'", %InputStateWest%) Return value: 2 Formula: QUERY("SELECT Name FROM Account WHERE BillingState = 'PA'") Formula: QUERY("SELECT Name FROM Account WHERE BillingState LIKE '{0}'", %InputStateEast%) Return value: 0

Omnistudio FUNCTION Function

Executes a custom function defined as a method of an Apex class.

See [Sample Apex Code for Custom Functions](#) for an Apex class that defines the custom functions shown in the examples.

Signature

```
FUNCTION(class, method, input...)
```

Return Value

Any data type

Parameters

Parameter	Data Type	Necessity	Description
<code>class</code>	String	Required	The name of the Apex class that defines the custom function to be called. The class must define the named <code>method</code> and must implement the <code>Callable</code> interface.
<code>method</code>	String	Required	The name of the Apex method that defines the custom function to be called. The method must be defined in the named <code>class</code> .

Parameter	Data Type	Necessity	Description
<i>input...</i>	Any date type	Required	A comma-separated list of one or more values to be passed as input to the custom function. To pass a list or a JSON array as an input value, use the <code>LIST</code> function to pass a JSON list (an anonymous array).

 **The convert Custom Function Example** Sample data:

```
"InputList": [
    "abc",
    "def",
    "ghi"
],
"ListKey": "Item"
```

Formula: `FUNCTION('MyCustomFunctions', 'convert', LIST(%InputList%), %ListKey%)` Formula: `FUNCTION('MyCustomFunctions', 'convert', LIST(InputList), ListKey)` Formula: `FUNCTION('MyCustomFunctions', 'convert', LIST("abc", "def", "ghi"), 'Item')` Return value:

```
[
{
    "Item": "abc"
},
{
    "Item": "def"
},
{
    "Item": "ghi"
}]
```

 **The replace Custom Function Example** Sample data:

```
"InputString": "my-input-string",
"ToReplace": "-",
"ReplaceWith": "_"
```

Formula: `FUNCTION('MyCustomFunctions', 'replace', %InputString%, %ToReplace%, %ReplaceWith%)` Formula: `FUNCTION('MyCustomFunctions', 'replace', InputString, ToReplace, ReplaceWith)` Formula: `FUNCTION('MyCustomFunctions', 'replace', "my-`

`input-string", "-", "_")` Return value: "my_input_string"

The formatPhoneNumber Custom Function Example Sample data: "Phone": 1234567890

Formula: `FUNCTION('MyCustomFunctions', 'formatPhoneNumber', %Phone%)` Formula:
`FUNCTION('MyCustomFunctions', 'formatPhoneNumber', Phone)` Formula:
`FUNCTION('MyCustomFunctions', 'formatPhoneNumber', 1234567890)` Formula:
`FUNCTION('MyCustomFunctions', 'formatPhoneNumber', "1234567890")` Return value:
`"(123) 456-7890"`

Omnistudio GENERATEGLOBALKEY Function

Generates a global key for use as an Id in a data pack.

Signature

`GENERATEGLOBALKEY (prefix)`

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>prefix</code>	String	Optional	A string to be prepended to the global key that the function returns. The function inserts a - (hyphen) between the prefix and the global key.

 **No Prefix Example** Formula: `GENERATEGLOBALKEY ()` Return value: "15c4ec2d-be7a-4f54-b342-652fdf159f30"

 **Prefix Example** Formula: `GENERATEGLOBALKEY ("SUB")` Return value: "SUB-f72bd2ca-a03b-48e3-af30-485efb4fa4d9"

Omnistudio QUERY Function

Executes a SOQL query and returns a JSON list of results.

Pass a single SOQL query to the function. The query can retrieve data from only a single column (a single field).

To get the size of the JSON list that's returned, first use the `ISBLANK` function to test for an empty list. If

the list is not empty, use the `LISTSIZE` function to return its size. Call the functions in separate Set Values actions or Data Mapper functions. (The `COUNTQUERY` function returns the number of matching rows from a query.)

Signature

```
QUERY (query)
```

Return Value

JSON list

Parameters

Parameter	Data Type	Necessity	Description
<code>query</code>	SOQL Query	Required	A string that contains a valid SOQL query. In the <code>WHERE</code> clause, you can pass a string directly or use ' <code>{0}</code> ' to refer to an additional value passed as a string, a JSON key, or an expression.

Query with Results Example Sample Account object data from database:

```
{
    "Account Name": "Acme Shipping",
    "Account Number": "CD451796",
    "Billing State": "GA",
    "Billing Zip": 27215
},
{
    "Account Name": "GenePoint",
    "Account Number": "CC978213",
    "Billing State": "CA",
    "Billing Zip": 94043
},
{
    "Account Name": "Edge Communications",
    "Account Number": "CD451796",
    "Billing State": "TX",
    "Billing Zip": 78767
},
{
    "Account Name": "Express Logistics",
```

```

    "Account Number": "CD736025",
    "Billing State": "MO",
    "Billing Zip": 63101
},
{
    "Account Name": "Pyramid Construction",
    "Account Number": "CD112262",
    "Billing State": "CA",
    "Billing Zip": 94087
}

```

Sample data:

```

"InputStateWest": "CA",
"InputStateEast": "PA"

```

Formula: `QUERY("SELECT Name FROM Account WHERE BillingState = 'CA'")` **Formula:** `QUERY("SELECT Name FROM Account WHERE BillingState LIKE '{0}'", %InputStateWest%)` **Return value:** ["GenePoint", "Pyramid Construction"] **Formula:** `LISTSIZE(QueryResult)` **Return value:** 2



Query with No Results Example **Formula:** `QUERY("SELECT Name FROM Account WHERE BillingState = 'PA'")` **Formula:** `QUERY("SELECT Name FROM Account WHERE BillingState LIKE '{0}'", %InputStateEast%)` **Return value:** {} **Formula:** `LISTSIZE(QueryResult)` **Return value:** 1 **Formula:** `IF(ISBLANK(QueryResult), 0, LISTSIZE(QueryResult))` **Return value:** 0



Query with Count Query Example **Formula:**

```

IF(COUNTQUERY("SELECT Name FROM Account WHERE BillingState = 'CA'"),
  LIST(QUERY("SELECT Name FROM Account WHERE BillingState = 'CA'")),
  0)

```

Return value: ["GenePoint", "Pyramid Construction"] **Formula:**

```

IF(COUNTQUERY("SELECT Name FROM Account WHERE BillingState = 'PA'"),
  LIST(QUERY("SELECT Name FROM Account WHERE BillingState = 'PA'")),
  0)

```

Return value: 0

Omnistudio GLOBALAUTONUMBER Function

Generates a unique number as per the configurations you define through the Omni Global Auto Number page.

For more information on defining your own numbering system, see [Set Up Omni Global Auto Numbers](#).

Signature

```
GLOBALAUTONUMBER ("GlobalAutoNumber") , GLOBALAUTONUMBER ( "Name")
```

Return Value

String

Parameters

Parameter	Data Type	Necessity	Description
<code>GlobalAutoNumber</code>	String	Optional	A global auto number name, which you created on the Omni Global Auto Number page. When you call this function without any value as an input, the value <code>GlobalAutoNumber</code> is automatically set. If there's data associated with the name, the function generates a number as per this data. If not, an error is thrown.

 **No Name Specified Example** Formula: `GLOBALAUTONUMBER ()` Return value: `"OS-12345"`

 **Example with the Name "Loan Application Number"** Formula: `GLOBALAUTONUMBER ("Loan Application Number")` Return value: `"LI-6298525"`

Sample Apex Code for Custom Functions

Develop Apex classes to implement custom functions for use with Omnistudio Data Mappers and Integration Procedures. A sample Apex class implements three methods that can be called as custom functions in Omnistudio formulas. The class includes code that passes a list to a custom function.

Data Mappers and Integration Procedures can invoke these functions to convert a list to an array, replace one character with another, or reformat phone numbers. The class and `invokeMethod` of custom functions must be global.

The convert function

The first custom function, `convert`, converts a list of values without keys to an array of values with keys. The function signature is:

```
FUNCTION('MyCustomFunctions', 'convert', LIST(%Inputlist%), %ListKey%)
```

If the list key is `"Item"` and the input list is:

```
{
  "InputList": [ "abc", "def", "ghi" ]
}
```

The result is:

```
{
  [
    {
      "Item": "abc"
    },
    {
      "Item": "def"
    },
    {
      "Item": "ghi"
    }
  ]
}
```

The replace function

The second function, `replace`, replaces a character in a string with another character. The function signature is:

```
FUNCTION('MyCustomFunctions', 'replace', %InputString%, %ToReplace%, %ReplaceWith%)
```

If the input string is `my-input-string`, the character to replace is `-`, and the character to replace it with is `_`, the result is `my_input_string`.

The formatPhoneNumber function

The third function, `formatPhoneNumber`, converts a phone number to the format `(nnn) nnn-nnnn`. The number must have no country code, a 3-digit area code, a three-digit exchange, and an overall length of at least nine digits. The function signature is:

```
FUNCTION('MyCustomFunctions', 'formatPhoneNumber', %Phone%)
```

If the input is `123.456.7890`, `123-456-7890`, or `1234567890`, the result is `(123) 456-7890`.

Passing a list to a custom function

If input to your custom function is of the type `List<Object>`, Apex saves the input to a `VLOCITY-FORMULA-LIST` key. To retrieve the input, use code that retrieves this key.

```
static List<Map<String, Object>> FUNCTION(List<Object> inputs)
{
    List<Object> listInput = ((Map<String, Object>)inputs[0]).get('VLOCITY-FORMULA-LIST');
    Object variableInput = inputs[1];

    // Rest of function
}
```

For a working example that passes a list to a custom function, see the implementation of the `convert` function in the example code.

The Apex class

The custom functions just described are defined as methods of the class `MyCustomFunctions`. For more information about calling custom functions and examples of the three custom functions defined in the class, see [Omnistudio FUNCTION Function](#).

```
global class MyCustomFunctions implements Callable
{

    /*
     * input - arguments - List<Object> of arguments passed to the function
     * output - result      - The function supports single Object values, List<Object>,
     *                         or Map<String, Object>
     */

    public Object call(String action, Map<String, Object> args)
    {
        Map<String, Object> input = (Map<String, Object>) args.get('input');
        Map<String, Object> output = (Map<String, Object>) args.get('output');
        Map<String, Object> options = (Map<String, Object>) args.get('option')
```

```
s');

    return invokeMethod(action, input, output, options);
}

global Boolean invokeMethod(String methodName, Map<String, Object> inputs,
    Map<String, Object> output, Map<String, Object> options)
{
    if (methodName == 'convert') {
        List<Object> arguments = (List<Object>) inputs.get('arguments');
        output.put('result', convert(arguments));
    } else if (methodName == 'replace') {
        List<Object> arguments = (List<Object>) inputs.get('arguments');
        output.put('result', replace(arguments));
    } else if (methodName == 'formatPhoneNumber') {
        List<Object> arguments = (List<Object>) inputs.get('arguments');
        output.put('result', formatPhoneNumber(arguments));
    }
    return true;
}

private LIST<Map<String, Object>> convert(List<Object> arguments)
{
    try {
        LIST<Map<String, Object>> result = new LIST<Map<String, Object>>();
        Map<String, Object> inputlist = (Map<String, Object>) arguments[0];
        String key = (String) arguments[1];
        List<Object> listofElements = (List<Object>) inputlist.get('VLOCITY-FORMULA-LIST');
        for (Object str : listofElements) {
            result.add(new Map<String, Object>{key => str});
        }
        return result;
    } catch(Exception e) {
        return new LIST<Map<String, Object>>();
    }
}

private String replace(List<Object> arguments)
{
    String result = '';
    String inputlist = (String) arguments[0];
    String toreplace = (String) arguments[1];
```

```
String replaceby = (String) arguments[2];
result = inputlist.replaceAll(toreplace, replaceby);
return result;
}

public string formatPhoneNumber(List<Object> arguments)
{
    String Phoneno = arguments[0] + '';
    String formattedPhone = Phoneno.replaceAll('[^0-9+]', '');
    String NewFormat;
    if (String.isNotBlank(formattedPhone) && formattedPhone.length() >= 9)
    {
        String first = formattedPhone.subString(0, 3);
        String second = formattedPhone.subString(3, 6);
        String third = formattedPhone.subString(6);
        NewFormat = '(' + first + ')' + ' ' + second + '-' + third;
    }
    return NewFormat;
}

}
```

Omnistudio

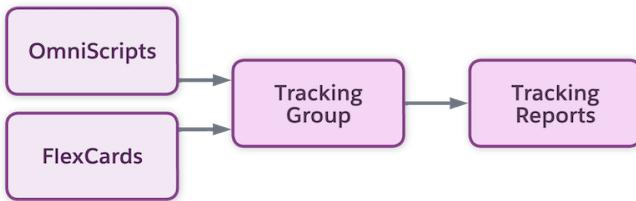
Use OmniAnalytics to track user interactions to answer questions such as which advertisements get the most responses and how many users who begin a purchasing process complete it. OmniAnalytics can track user interactions with Omniscripts and Flexcards with or without a third-party system such as Google Analytics.

Omnistudio OmniAnalytics is supported with the Omnistudio standard runtime and standard objects.

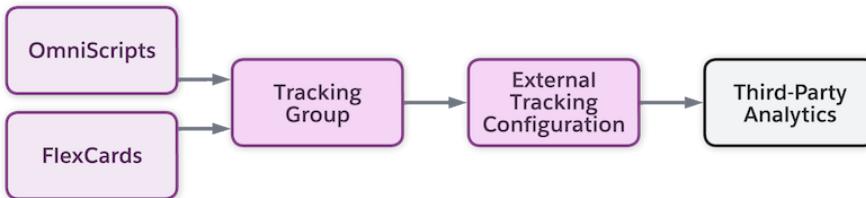
OmniAnalytics is available in Spring '24 and later releases and integrates with Google analytics 4.

Data Flows

In OmniAnalytics, user interaction data flows from Omniscripts and Flexcards, through tracking components, and into OmniAnalytics dashboards.



User interaction data from OmniAnalytics can also flow from Omniscripts and Flexcards, through tracking components, and into a third-party system such as Google Analytics.



You can send the same data to OmniAnalytics dashboards, a third-party system, or both.

You can use OmniAnalytics to control the data that you send, how it's structured, and where it goes without editing templates or writing a script.

Types of Data Collected

OmniAnalytics collects data that helps answer questions.

- Which items are users most interested in?
- Where are we losing users in the flow?
- Are some steps too slow to load or too hard to understand?

Specifically, OmniAnalytics collects data about user activity:

- Clicks per product, offer, or promotion impression
- Clicks per UI action impression
- Percent of Omniscripts completed
- Percent of Omniscript steps completed
- Load time for Flexcards, Omniscripts, and Omniscript steps
- Viewing time for Flexcards, Omniscripts, and Omniscript steps

Next Steps

To use OmniAnalytics, enable it and decide how to store tracked data. After enabling OmniAnalytics, configure it to track without a third-party analytics provider or with Google Analytics.

[Enable OmniAnalytics and Store Tracking Data](#)

To enable OmniAnalytics, configure settings that determine whether OmniAnalytics is enabled and whether OmniAnalytics data is stored in OmniTrackingEvent platform events, Decision Explainer Service stores, or both.

[Configure Internal OmniAnalytics](#)

OmniAnalytics can track user interactions with Omniscripts and Flexcards without the use of a third-party analytics provider.

[Configure OmniAnalytics with Google Analytics](#)

OmniAnalytics can track user interactions with Omniscripts and Flexcards in combination with Google Analytics and Google Tag Manager.

[Create Google Analytics Trusted URLs](#)

For OmniAnalytics and Google Analytics to communicate, your Salesforce org must trust the Google Analytics websites.

[Adding Ecommerce Data to the Example Omniscript for OmniAnalytics](#)

Google Tag Manager Ecommerce requires specific kinds of data. You must add this data to the Messaging Framework Omniscript properties. When the Omniscript runs, it sends this data to Google Analytics.

[Setting Up Third-Party Tracking and Event Types](#)

An External Tracking Definition specifies a third-party analytics account and tag manager that the

OmniAnalytics tracking data is sent to. External Tracking Event Types specify data formats for specific events.

Enable OmniAnalytics and Store Tracking Data

To enable OmniAnalytics, configure settings that determine whether OmniAnalytics is enabled and whether OmniAnalytics data is stored in OmniTrackingEvent platform events, Decision Explainer Service stores, or both.

1. Go to Setup.
 - In Lightning Experience, click the gear icon, , and select **Setup**.
 - In Salesforce Classic, click the user menu and select **Setup**.
2. From Setup, in the Quick Find box, enter *omni*.
3. Expand **OmniAnalytics** and click **Settings**.
4. Select **Enabled** or **Disabled** for each setting.

To follow the examples in the workflow, enable Omnistudio Analytics and Enable Event Internal Writes. You can either enable or disable Enable Event Notifications.

Name	Default Value	Description
Omnistudio Analytics	Disabled	If enabled, OmniAnalytics can track user interaction data for Omniscripts and Flexcards.
Enable Event Notifications	Disabled	If enabled, tracking data is published to OmniTrackingEvent standard platform events.
Enable Event Internal Writes	Disabled	If enabled, tracking data is logged in Decision Explainer Service (DES) stores. OmniAnalytics automatically creates DES Business Process Types – their names begin with OA_ . You can't edit or delete DES data, but you can retrieve it by using Decision Explainer Business APIs .



Note To ensure that data for Tracking Groups with a Type of Internal is saved, set Enable Event Internal Writes to true. For Tracking Groups with a Type of External, if you don't need to save data

to Salesforce, you can set Enable Event Internal Writes, Enable Event Notifications, or both to false. See [Create a Tracking Group and Add Components to Track](#).

After you have enabled OmniAnalytics, see the [Configure Internal OmniAnalytics](#).

Configure Internal OmniAnalytics

OmniAnalytics can track user interactions with Omniscripts and Flexcards without the use of a third-party analytics provider.

 **Note** Example Omniscript and Flexcard The tasks in this workflow use two examples throughout:

- An Omniscript that illustrates a simplified purchasing process for a smartphone.
- A Flexcard that redirects the user to the Google website.

If you aren't following the examples, only tasks 3, 4, and 6 of this workflow are required.

1. [Import and Activate the Example Omniscript](#).

2. [Import and Activate the Example Flexcard](#).

3. [Create a Tracking Group and Add Components to Track](#).

Specify the example Omniscript and Flexcard as components to be tracked.

4. [Add Tracked Components to a Lightning App Page](#).

Create an example page that includes the example Omniscript and Flexcard.

5. [Interact with the Example Omniscript and Flexcard](#).

6. [View Tracked Data in the Tracking Group](#).

Import and Activate the Example Omniscript

To help you learn about OmniAnalytics, Salesforce provides an example Omniscript that you can download, import, and activate.

In later tasks, you add the Omniscript to a Lightning App Page and interact with it to generate OmniAnalytics data.

For a sneak preview of how the example Omniscript and Flexcard work, see [Interact with the Example Omniscript and Flexcard](#).

 **Important** All Omniscript steps must have **Step Label** values. If these values are missing, the steps don't appear when you [View Tracked Data in the Tracking Group](#).

The example Omniscript includes multiple steps to select and purchase a product. In the Preview tab of the Omniscript Designer, the example looks like this:

Select Device

Choose a device

Save for later

Next

Steps

- Select Device
- View Quote
- Cart Details
- Review Cart
- Shipping Address
- Provide Payment
- Order Receipt

For simplicity, and unlike a production Omniscript, the example Omniscript uses hard-coded data.

1. To download a DataPack of the example Omniscript, click [here](#), then click **Download**.
The DataPack automatically downloads to your machine.
2. From the App Launcher, find and select **Omnistudio**, then click **Omniscripts**.
3. Click **Import**, then click **Browse**, and select the Ecommerce Buyflow OmniScript.json file that you downloaded.
4. Click **Next** and respond to the import wizard prompts until the file is imported.
5. In the list of Omniscripts, Find and expand **Documentation/OmniAnalytics**, then click **E-commerce Buyflow Omniscript (Version 1)**.
The Omniscript opens in the Omniscript Designer.
6. If you see an **Activate Version** button, click it to activate the Omniscript.
7. Click **Setup**, expand **Messaging Framework**, and make sure the Window postMessage, Pub/Sub, and Session Storage boxes are selected.
These properties are required for OmniAnalytics.
8. Click each Step component in the Omniscript, expand **Messaging Framework**, and make sure the Window postMessage, Pub/Sub, and Session Storage boxes are selected.
These properties are required for OmniAnalytics.

After you import and activate the example Omniscript, [Import and Activate the Example Flexcard](#).

Import and Activate the Example Flexcard

To help you learn about OmniAnalytics, Salesforce provides an example Flexcard that you can download, import, and activate.

In later tasks, you add the Flexcard to a Lightning App Page and interact with it to generate OmniAnalytics data.

For a sneak preview of how the example Omniscript and Flexcard work, see [Interact with the Example Omniscript and Flexcard](#).

The example Flexcard includes two buttons for opening the Google website either in the same tab or in a new tab. In the Vlocity Flexcard Designer, it looks like this:

The screenshot shows a component configuration screen. At the top, there are fields for 'Id' (001SB000002d2jGYAQ) and 'Name' (Global Media). Below these are two action buttons: a blue one labeled 'NavigateAction' and a green one labeled 'NavigateActionNewWindow'. The green button is highlighted with a green border.

1. To download a DataPack of the example Flexcard, click [here](#), then click **Download**.
The DataPack automatically downloads to your machine.
2. From the App Launcher, find and select **Omnistudio**, then click **Omnistudio Flexcards**.
3. Click **Import**, then click **Upload Files**, and then select the DocumentationTestCard.json file.
4. Click **Next** and respond to the import wizard prompts until the file is imported.
5. In the list of Flexcards, find and expand **DocumentationTestCard**, then click **DocumentationTestCard (version 1)**.
The Flexcard opens in the Flexcard Designer.
6. If you see an **Activate** button, click it to activate the Flexcard.

After you import and activate the example Flexcard, [Create a Tracking Group and Add Components to Track](#).

Create a Tracking Group and Add Components to Track

To specify a group of components that you want to track together, create a Tracking Group and add Omniscript and Flexcard components to it.

If you aren't following the Omniscript and Flexcard examples, make sure your Omniscripts and Flexcards meet these criteria:

- All Omniscript Step components have Step Label values.
- The Omniscript Setup and each Step component have all Messaging Framework boxes checked: Window postMessage, Pub/Sub, and Session Storage.
- The Omniscript or Flexcard is active.
- If you originally downloaded the Omniscript or Flexcard from an Omnistudio for Vlocity org, you've clicked the dropdown in the designer next to Edit or Help and selected **Deploy Standard Runtime Compatible LWC**.

If you're following the Omniscript and Flexcard examples, you add the example Omniscript and Flexcard to the Tracking Group in this task.

 **Note** You can export and import Tracking Group and Tracking Component Definition objects as DataPacks. The referenced Omniscripts and Flexcards aren't included.

1. From Setup, in the Quick Find box, enter *omni*.
2. Expand **OmniAnalytics** and click **Tracking Groups**.
3. Click **New**.

4. For the name, enter *AnalyticsTest*.
5. From the Tracking Group Type dropdown list, select **Internal**.

Internal specifies that you aren't using a third-party Analytics system. External specifies that you're using a third-party system such as Google Analytics.



Note A specific Omniscript or Flexcard can belong to two Tracking Groups if one is Internal and the other is External.

6. Make sure **Is Active** is on for the Tracking Group.

New Omni Tracking Group

*Name <input type="text" value="AnalyticsTest"/>	IsActive <input checked="" type="checkbox"/> Active
*Tracking Group Type <input type="button" value="Internal"/>	Start Date <input type="text"/>
External Tracking Config Definition <input type="button" value="Select an Option"/>	End Date <input type="text"/>
Description <input type="text"/>	Max Age In Days <input type="text"/>

7. Save your changes.

The new tracking group is added to the list.

8. Click **AnalyticsTest**.

The tracking group page opens.

9. Click **Add Component**.

The Add Component window opens.

10. Select the **DocumentationTestCard** from the Available list and move it to the Selected list.

11. Collapse the **Flexcard** list and expand the **Omniscript** list.

12. Select the **Ecommerce Buyflow Omniscript** from the Available list and move it to the Selected list.

Add Component

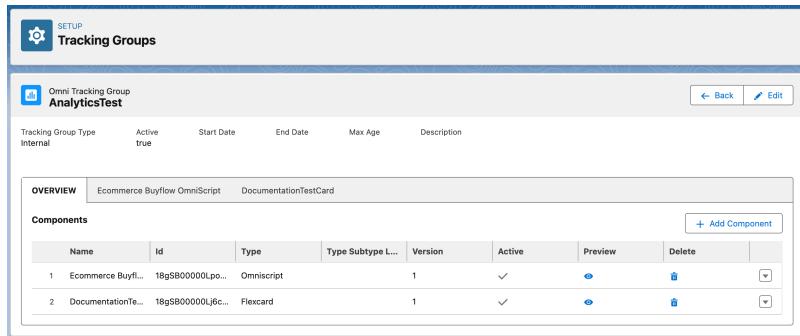
Search Components <input type="text"/>	Available <input type="button" value="☰"/> <input type="button" value="✖"/> > <input type="checkbox"/> Flexcard (8) > <input type="checkbox"/> OmniScript (70)	Selected <input type="button" value="▶"/> <input type="button" value="◀"/> ✓ <input type="checkbox"/> Flexcard (1) DocumentationTestCard ✓ <input type="checkbox"/> OmniScript (1) E-commerce Buyflow Omniscript
---	---	---

13. Save your changes.

You return to the Tracking Groups list.

14. Click **AnalyticsTest** again.

The Tracking Group looks like this:



 **Note** If you change a Tracking Group, you must refresh the browser.

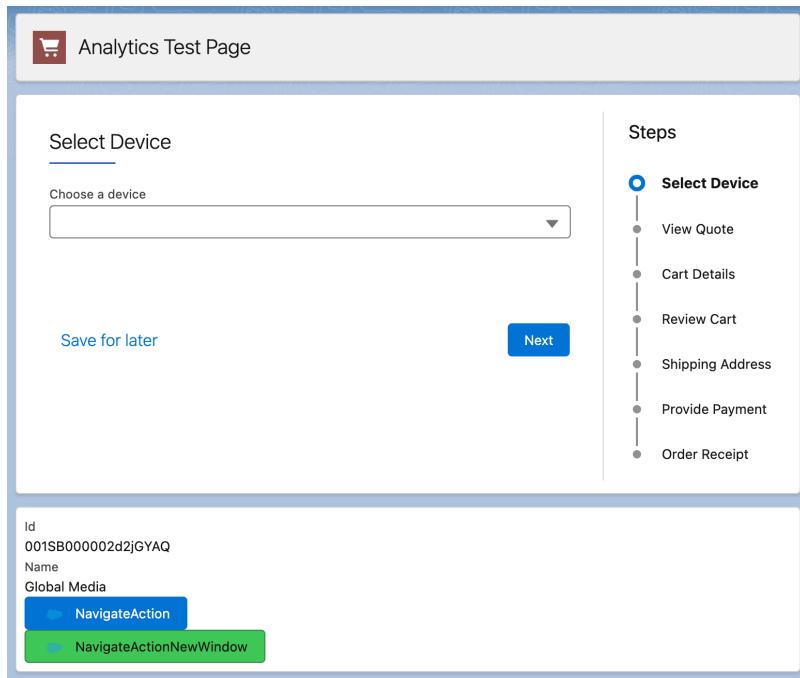
After you create the tracking group, [Add Tracked Components to a Lightning App Page](#).

Add Tracked Components to a Lightning App Page

Users interact with Omniscripts and Flexcards via Lightning App Pages. However, instead of directly adding Omniscripts and Flexcards to pages, you add wrapper components that make the Omniscripts and Flexcards available to OmniAnalytics.

If you're using the example Omniscript and Flexcard, in this task you add them to a Lightning App Page.

1. From Setup, in the Quick Find box, enter *lightning app*, and then select **Lightning App Builder**.
2. Click **New**.
3. Select **App Page**, then click **Next**.
4. In Label, type *Analytics Test Page*, then click **Next**.
5. Click **One Region**, then click **Done**.
The new page opens in Lightning App Builder.
6. Drag the **Omniscript** standard component onto the page canvas.
7. In the component properties, set the Type to **Documentation** and the SubType to **OmniAnalytics**, then select **Enable Omnistudio Analytics**.
8. Drag the **Flexcard** standard component onto the page canvas below the first component.
9. In the component properties, set the Flexcard Name to **DocumentationTestCard** and select **Enable Omnistudio Analytics**.
10. Save your changes and click **Activate**.
11. On the Activation page, click **Lightning Experience**, select **Sales**, click **Add page to app**, then save your changes.
12. To exit Lightning App Builder, click **Save**, then click **Back**.
13. From the App Launcher, go to the Sales app and Analytics Test Page.
14. Verify that the Analytics Test Page looks like this:



After you create the Lightning App Page, [Interact with the Example Omniscript and Flexcard](#) to generate tracking data.

Interact with the Example Omniscript and Flexcard

To generate tracking data, interact with the example Omniscript and Flexcard as if you were an end user.

- (checkmark) **Note** If you click **Save for later** in an Omniscript, subsequent actions in the resumed Omniscript aren't tracked. If a Flexcard opens a new page in the current browser tab, subsequent actions aren't tracked.

1. From the App Launcher, find and select **Sales**, then click **Analytics Test Page**.
 2. In the Omniscript, choose a device and click **Next**.
 3. Click **Add to Cart**.
 4. Click **Proceed**.
 5. Click **Proceed** again.
 6. Click **Pay Now**.
 7. Click **Generate Receipt**. The page looks something like this:
-
8. In the Flexcard, click **NavigateActionNewWindow**. A browser tab opens to the Google website. Don't click **NavigateAction** or subsequent actions aren't tracked. This action is included in case you want to use it for testing.
 9. Close the Google browser tab.
 10. To simulate purchase completion data, refresh the browser, click a few Omniscript steps, and stop before you reach the last step. Repeat this several times, stopping at a different step each time. In some of these repetitions, also click **NavigateActionNewWindow** in the Flexcard.

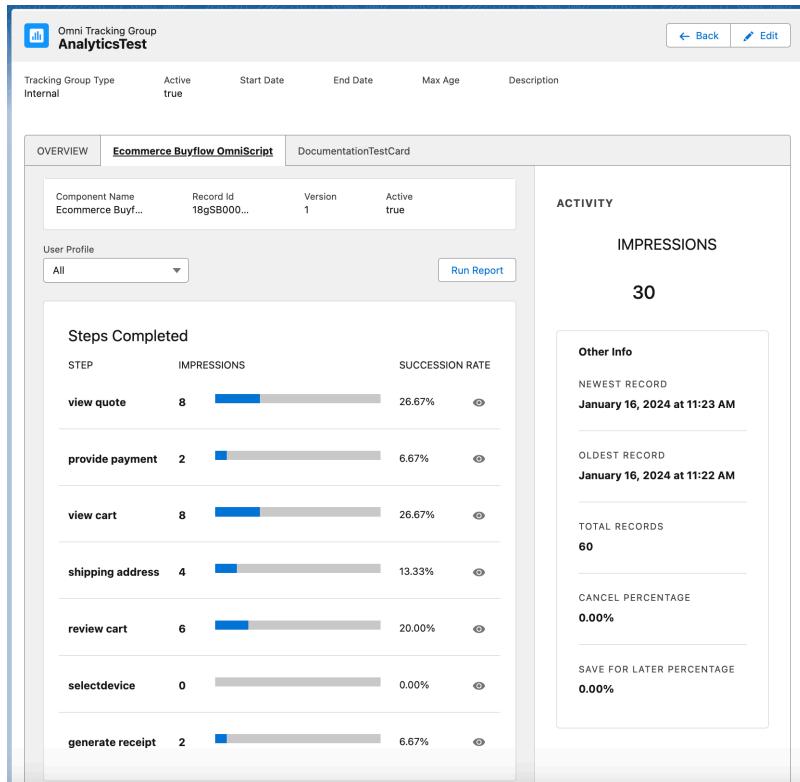
After you generate tracking data, [View Tracked Data in the Tracking Group](#).

View Tracked Data in the Tracking Group

After you interact with the example Omniscript and Flexcard, you generate and view reports of the tracked data in the tabs for these components on the Tracking Group page.

1. From Setup, in the Quick Find box, enter *omni*.
2. Expand **OmniAnalytics** and click **Tracking Groups**.
3. Click **AnalyticsTest**.
The Tracking Group that you created opens.
4. Click the **Ecommerce Buyflow Omniscript** tab and click **Run Report**.

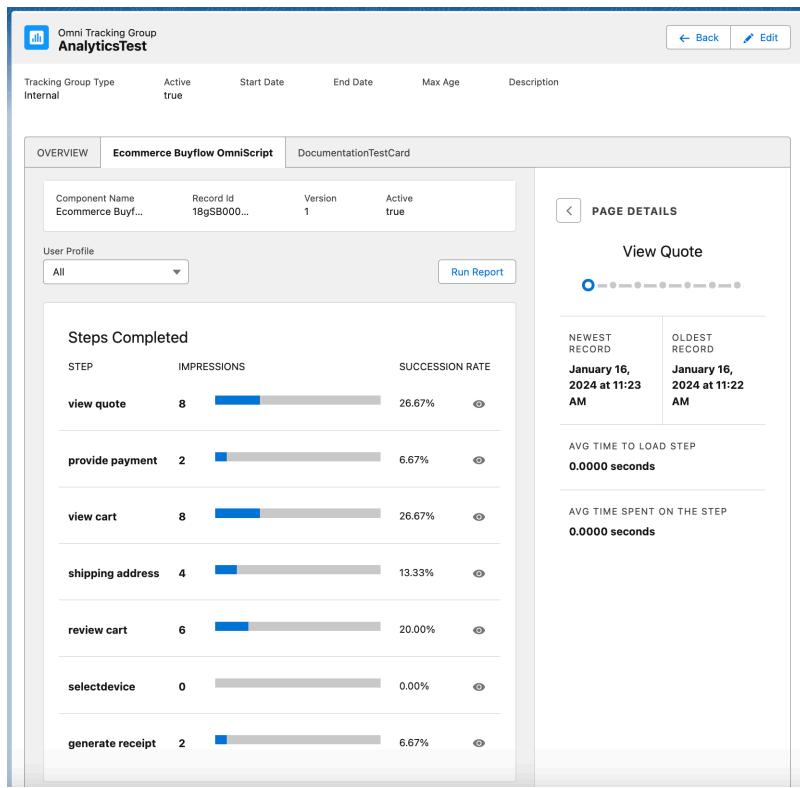
A dashboard similar to this appears:



Analytics report data is updated at 30-minute intervals.

! **Important** If some steps are missing, make sure all Omniscript steps have Step Label values.

5. Click the eye icon, , for any step to display details about that step in the right panel.



To return to the general information, click <.

- Click the **DocumentationTestCard** tab and click **Run Report**. A report appears that's similar to the Omniscript report.

If you don't see report data after 30 minutes, make sure that:

- All Omniscript Step components have Step Label values.
- The Omniscript Setup and each Step component have all Messaging Framework boxes checked: Window postMessage, Pub/Sub, and Session Storage.
- The Omniscript or Flexcard is active.
- If the Omniscript or Flexcard was originally downloaded from an Omnistudio for Vlocity org, you've clicked the dropdown in the designer next to Edit or Help and selected Deploy Standard Runtime Compatible LWC.
- An activated Lightning App Page includes the Omniscript or Flexcard standard component, which references the Omniscript or Flexcard, and has Enable Omnistudio Analytics checked.

Configure OmniAnalytics with Google Analytics

OmniAnalytics can track user interactions with Omniscripts and Flexcards in combination with Google Analytics and Google Tag Manager.

Note Example Omniscript and Flexcard The tasks in this workflow use two examples throughout:

- A Omniscript that illustrates a simplified purchasing process for a smartphone.
- A Flexcard that redirects the user to the Google website.
- The example Omniscript and Flexcard communicate with Google Analytics 4 by using the [Google Tag Manager Ecommerce APIs](#).

If you're familiar with Google Analytics and don't need the examples, perform step 1, make sure your Omniscripts meet these criteria, then skip to step 5.

- Enable Tracking in the Omniscript Setup is checked.
 - The Omniscript has JSON data to pass to Google Analytics and Google Tag Manager. In Setup or Step Properties, expand Messaging Framework and add Message key/value pairs.
1. Complete the [Configure Internal OmniAnalytics](#) as a prerequisite.
 2. [Create Google Analytics Trusted URLs](#).
 3. Set up Google Analytics at analytics.google.com and the Google Tag Manager at tagmanager.google.com.
 4. Complete the [Adding Ecommerce Data to the Example Omniscript for OmniAnalytics](#).
 5. Complete the [Setting Up Third-Party Tracking and Event Types](#).

Create Google Analytics Trusted URLs

For OmniAnalytics and Google Analytics to communicate, your Salesforce org must trust the Google Analytics websites.

1. From Setup, in the Quick Find box, enter *trust*, then click **Trusted URLs**.
2. Click **New Trusted URL**.
The Trusted URL page opens.
3. In API Name, enter *GoogleAnalytics1*.
4. In URL, enter *https://www.google-analytics.com*.
5. In the CSP Directives section, select every box.
6. Click **Save & New**.
7. Repeat steps 4–7 but specify an API Name of *GoogleAnalytics2* and a URL of *https://www.analytics.google.com*.
8. Repeat steps 4–6 but specify an API Name of *GoogleTagManager* and a URL of *https://www.googletagmanager.com*.
9. Save your changes.

After you create trusted URLs, [Adding Ecommerce Data to the Example Omniscript for OmniAnalytics](#)..

Adding Ecommerce Data to the Example Omniscript for OmniAnalytics

Google Tag Manager Ecommerce requires specific kinds of data. You must add this data to the Messaging Framework Omniscript properties. When the Omniscript runs, it sends this data to Google Analytics.

-  **Note** Flexcards send a default payload to the Google Tag Manager. For details, see [Create Flexcard Event Types](#).

The example Omniscript and Flexcard communicate with Google Analytics 4 by using the [Google Tag Manager Ecommerce APIs](#). For simplicity, and unlike a production Omniscript, the example Omniscript uses hard-coded data.

After you add Google Tag Manager data to the example Omniscript, return to the parent workflow and perform the [Setting Up Third-Party Tracking and Event Types](#).

[Add Google Tag Manager Data to the Example Omniscript Setup](#)

The Messaging Framework of the Script Setup stores the complete list of product impressions that's sent to Google Analytics Ecommerce and the Google Tag Manager.

[Add Google Tag Manager Data to the Example Omniscript Steps](#)

The Messaging Framework of each step stores the product data that's sent to Google Analytics Ecommerce and the Google Tag Manager.

Add Google Tag Manager Data to the Example Omniscript Setup

The Messaging Framework of the Script Setup stores the complete list of product impressions that's sent to Google Analytics Ecommerce and the Google Tag Manager.

1. From the App Launcher, find and select **Omnistudio**, then click **Omniscripts**.
2. In the Omniscript list, expand **Documentation/OmniAnalytics** and click **E-commerce Buyflow Omniscript (Version 1)**.
3. Click **New Version**.
The new version is Version 2. The previous version without the Google Tag Manager data is Version 1.
4. In the Properties pane, click **Setup**.
5. Select **Enable Tracking**.
6. Expand the Messaging Framework section.
7. Make sure the Window postMessage, Pub/Sub, and Session Storage boxes are selected.
8. Under Message, add these Key/Value pairs:

Key	Value
eventAction	view_item_list

Key	Value
products	(leave blank)

9. Scroll to the bottom and click **Edit Properties as JSON**.
10. Replace the empty quotes after the `products` node with this JSON array. Make sure a colon precedes the opening bracket and a closing brace follows the closing bracket.

```
[
  {
    "item_name": "Apple iPhone 7",
    "item_brand": "Apple",
    "item_category": "Mobile",
    "item_variant": "Black",
    "price": 699,
    "quantity": 1
  },
  {
    "item_name": "Apple iPhone 8",
    "item_brand": "Apple",
    "item_category": "Mobile",
    "item_variant": "Black",
    "price": 699,
    "quantity": 1
  },
  {
    "item_name": "Apple iPhone X",
    "item_brand": "Apple",
    "item_category": "Mobile",
    "item_variant": "Black",
    "price": 699,
    "quantity": 1
  },
  {
    "item_name": "Apple iPhone 11",
    "item_brand": "Apple",
    "item_category": "Mobile",
    "item_variant": "Black",
    "price": 699,
    "quantity": 1
  },
  {
    "item_name": "Apple iPhone XR",
    "item_brand": "Apple",
    "item_category": "Mobile",
    "item_variant": "Black",
    "price": 699,
    "quantity": 1
  }
]
```

```
    "item_category": "Mobile",
    "item_variant": "Black",
    "price": 699,
    "quantity": 1
}
]
```

11. Click **Close JSON Editor**.

After you add Google Tag Manager data to the example Omniscript Setup, [Add Google Tag Manager Data to the Example Omniscript Steps](#).

Add Google Tag Manager Data to the Example Omniscript Steps

The Messaging Framework of each step stores the product data that's sent to Google Analytics Ecommerce and the Google Tag Manager.

If you aren't in the Omniscript, from the App Launcher, find and select **Omnistudio**, then click **Omniscripts**. In the Omniscript list, expand **Documentation/OmniAnalytics** and click **E-commerce Buyflow Omniscript (Version 2)**.

Perform these steps for each Step element in the Omniscript:

1. Click the Step element.
2. In the Properties pane, click **Properties** to display the Step element properties.
3. Expand the Messaging Framework section.
4. Make sure the Window postMessage, Pub/Sub, and Session Storage boxes are checked.
5. Under Message, add a Key/Value pair with *eventAction* as the key. The value depends on the Step element you're editing:

Step Element	Value of eventAction Key
Select Device	view_item_list
View Quote	select_item
View Cart	add_to_cart
Review Cart	begin_checkout
Shipping Address	add_shipping_info

Step Element	Value of eventAction Key
Provide Payment	add_payment_info
Generate Receipt	purchase

6. Add a Key/Value pair with *transID* as the key. You can leave the value blank for all the steps except the Generate Receipt step. For that step, you can use any value. For example, you can use a timestamp.

If your Google Analytics configuration processes all events in the same way, all events require the same parameters, even if some events don't use them.

7. Add a Message Key/Value pair with *products* as the key, and leave the value blank.
 8. Click **Edit Properties as JSON**.
 9. Replace the empty quotes after the *products* node with one of these JSON arrays. Make sure a colon precedes the opening bracket and a closing brace follows the closing bracket.

For the Select Device step, use this JSON array:

```
[
  {
    "item_name": "Apple iPhone 7",
    "item_brand": "Apple",
    "item_category": "Mobile",
    "item_variant": "Black",
    "price": 699,
    "quantity": 1
  },
  {
    "item_name": "Apple iPhone 8",
    "item_brand": "Apple",
    "item_category": "Mobile",
    "item_variant": "Black",
    "price": 699,
    "quantity": 1
  },
  {
    "item_name": "Apple iPhone X",
    "item_brand": "Apple",
    "item_category": "Mobile",
    "item_variant": "Black",
    "price": 699,
    "quantity": 1
  }
],
```

```
[  
  {  
    "item_name": "Apple iPhone 11",  
    "item_brand": "Apple",  
    "item_category": "Mobile",  
    "item_variant": "Black",  
    "price": 699,  
    "quantity": 1  
  },  
  {  
    "item_name": "Apple iPhone XR",  
    "item_brand": "Apple",  
    "item_category": "Mobile",  
    "item_variant": "Black",  
    "price": 699,  
    "quantity": 1  
  }  
]
```

For all other steps, use this JSON array:

```
[  
  {  
    "item_name": "%DeviceName%",  
    "item_brand": "Apple",  
    "item_category": "Mobile",  
    "item_variant": "Black",  
    "price": 699,  
    "quantity": 1  
  }  
]
```

10. Click **Close JSON Editor**.

This is the final task in the current workflow. For the next task in the parent workflow, see [Setting Up Third-Party Tracking and Event Types](#).

Setting Up Third-Party Tracking and Event Types

An External Tracking Definition specifies a third-party analytics account and tag manager that the OmniAnalytics tracking data is sent to. External Tracking Event Types specify data formats for specific events.

-  **Note** You can export and import External Tracking Configuration and External Tracking Event Type objects as DataPacks. The corresponding Omniscripts and Flexcards aren't included.

Before You Begin

- Complete the [Configure Internal OmniAnalytics](#).
- Configure the third-party analytics account and tag manager. See the [Google Analytics](#) and [Google Tag Manager](#) documentation.
- See the [Adding Ecommerce Data to the Example Omniscript for OmniAnalytics](#).

1. Create an External Tracking Configuration

Third-party analytics vendors require an External Tracking Configuration in addition to a Tracking Group. The steps in this task are for Google Analytics.

2. Create Omniscript Event Types

External Tracking Event Types arrange Omniscript data in the format that a third-party analytics provider expects. The steps in this task are for Google Analytics and Google Tag Manager Ecommerce.

3. Create Flexcard Event Types

External Tracking Event Types arrange Flexcard data in the format that a third-party analytics provider expects. The steps in this task are for Google Analytics Ecommerce.

4. Create Inclusion Rules for Event Types

Add an Inclusion Rule to an External Tracking Event Type to determine whether the event is sent to a third-party analytics vendor.

5. Connect the Tracking Group and External Tracking Configuration

You can edit a Tracking Group to send its data to a third-party analytics provider without changing the components that the group includes.

Create an External Tracking Configuration

Third-party analytics vendors require an External Tracking Configuration in addition to a Tracking Group. The steps in this task are for Google Analytics.

1. From Setup, in the Quick Find box, enter *omni*.
2. Expand **OmniAnalytics** and click **External Tracking Configurations**.
3. Click **New**.
The New External Tracking Configuration window opens.
4. For the name, enter *GoogleAnalyticsTest*.
5. In Tracking Vendor, click in the text box and select **Google**.

Under Replacement Values, the `id` key and the **Script URL** property are automatically added.

6. In the value field for the `id` key, paste the Google Tag Manager ID that you received when you created your Google Tag Manager account. It has the format `GTM-XXXXXXX`.

If you didn't copy this ID when you created your account, log in to Google Tag Manager, open your container if you have more than one, and copy the `GTM-XXXXXXX` text.

New External Tracking Configuration

*Name: GoogleAnalyticsTest

Is Active: Active

Tracking Framework Information

*Tracking Vendor: Google

Script URL: https://www.googletagmanager.com/gtm.js?id=\${id}

Replacement Values

id	GTM-XXXXXXX	
----	-------------	--

+ Add New Key/Value Pair

Description:

Cancel Save

7. Save your changes.

The new External Tracking Configuration is added to the list.

Note If you change an External Tracking Configuration or External Tracking Event Type, you must refresh the browser.

After you create the External Tracking Configuration, [Create Omniscript Event Types](#).

Create Omniscript Event Types

External Tracking Event Types arrange Omniscript data in the format that a third-party analytics provider expects. The steps in this task are for Google Analytics and Google Tag Manager Ecommerce.

1. From Setup, in the Quick Find box, enter *omni*.
2. Expand **OmniAnalytics** and click **External Tracking Event Types**.
3. Click **New**.
4. For the External Tracking Configuration, select **GoogleAnalyticsTest**.
5. For the Component Type, select **Omniscript**.
6. Click in the **Name** text box and select **OS Invoke**.

Note You can select a name from the list or enter a custom name. If you enter a custom name, this name isn't added to the Name list next time you create an Event Type. You must enter the custom name each time.

7. Remove the entire contents of the Payload Template text area and replace it with this JSON code:

```
{
  "event": "%eventAction%",
  "ecommerce": {
    "currency": "USD",
    "label": "Product A"
  }
}
```

```
        "value": 0,  
        "items": "%products%"  
    }  
}
```

The `%products%` merge field here references the JSON structure assigned to the `products` key in the example Omniscript's Setup.

8. Save your changes.
The new External Tracking Event Type is added to the list.
9. Click **New** again.
10. For the External Tracking Configuration, select **GoogleAnalyticsTest**.
11. For the Component Type, select **Omniscript**.
12. Click in the **Name** text box and select **OS Step Load**.
13. Remove the entire contents of the Payload Template text area and replace it with this JSON code:

```
{  
    "event": "%eventAction%",  
    "ecommerce": {  
        "currency": "USD",  
        "transaction_id": "%transID%",  
        "value": 699,  
        "items": "%products%"  
    }  
}
```

With merge fields, you can use the same payload format for every step in the Omniscript.

14. Save your changes.
The new External Tracking Event Type is added to the list.

After you create Omniscript Event Types, [Create Flexcard Event Types](#).

Create Flexcard Event Types

External Tracking Event Types arrange Flexcard data in the format that a third-party analytics provider expects. The steps in this task are for Google Analytics Ecommerce.

Unlike Omniscripts, which have configurable Messaging Framework data, Flexcards have default external tracking payloads for their events. However, you can alter the default payloads by using External Tracking Event Types.

The External Tracking Event Type for the example Flexcard maps this default payload to the Google Analytics Ecommerce format for a `view_promotion` event.

1. From Setup, in the Quick Find box, enter *omni*.
2. Expand **OmniAnalytics** and click **External Tracking Event Types**.
3. Click **New**.
4. For the **External Tracking Configuration**, select **GoogleAnalyticsTest**.
5. For the **Component Type**, select **Flexcard**.
6. Click in the **Name** text box and select **UI Action**.



Note You can select a Name from the list or enter a custom name. If you enter a custom name, this name isn't added to the Name list next time you create an Event Type. You must enter it each time.

7. Review the default contents of the Payload Template text area. This JSON structure is the default external tracking payload for the selected or custom Flexcard event. The only merge fields you can use are the merge fields in the default payload.

This JSON structure is the default payload for a **UI Action** event:

```
{  
    "ActionContainerId": "%ActionContainerId%",  
    "ActionContainerComponent": "%ActionContainerComponent%",  
    "InstanceIdentifier": "%InstanceIdentifier%",  
    "ActionContainerGlobalKey": "%ActionContainerGlobalKey%",  
    "ActionElementType": "%ActionElementType%",  
    "TrackingCategory": "%TrackingCategory%",  
    "RequestUrl": "%RequestUrl%",  
    "BusinessEvent": "%BusinessEvent%",  
    "BusinessCategory": "%BusinessCategory%",  
    "ActionElementName": "%ActionElementName%",  
    "ActionElementLabel": "%ActionElementLabel%",  
    "ActionTargetType": "%ActionTargetType%",  
    "ActionTargetName": "%ActionTargetName%"  
}
```

8. Remove the entire contents of the Payload Template text area and replace it with this JSON code:

```
{  
    "event": "view_promotion",  
    "ecommerce": {  
        "promotion_name": "%BusinessEvent%",  
        "items": [  
            {  
                "item_name": "%BusinessEvent%"  
            }  
        ]  
    }  
}
```

This payload specifies literal values and uses the **%BusinessEvent%** merge field from the default

payload.

9. Save your changes.

The new External Tracking Event Type is added to the list.

After you create Flexcard Event Types, optionally [Create Inclusion Rules for Event Types](#) or skip ahead to [Connect the Tracking Group and External Tracking Configuration](#).

Create Inclusion Rules for Event Types

Add an Inclusion Rule to an External Tracking Event Type to determine whether the event is sent to a third-party analytics vendor.

If the Inclusion Rule text area is blank, all events of the specified type are tracked.

1. From Setup, in the Quick Find box, enter *omni*.
2. Expand **OmniAnalytics** and click **External Tracking Event Types**.
3. Click **New**, or click an existing External Tracking Event Type and click **Edit**.
4. In Inclusion Rule, type a condition that evaluates to true or false.

You can use the Field Name, Math Operators, and Logical Operators dropdowns to construct the condition. You can also use Omniscript functions and merge fields.

Here are some examples of possible inclusion rules:

```
%StepSequence% > 1  
%LoadDuration% > 1000 || %StepWaitTime% > 2000  
CONTAINS(%DeviceName%, "Apple iPhone X")
```

5. Save your changes.
6. If you edited an existing External Tracking Event Type, refresh the browser.

After you create Inclusion Rules, [Connect the Tracking Group and External Tracking Configuration](#).

Connect the Tracking Group and External Tracking Configuration

You can edit a Tracking Group to send its data to a third-party analytics provider without changing the components that the group includes.

1. From Setup, in the Quick Find box, enter *omni*.
2. Expand **OmniAnalytics** and click **Tracking Groups**.
3. Click **AnalyticsTest**.

AnalyticsTest is the [Tracking Group that you created](#). If you edit this Tracking Group, you can use the [Lightning App Page that you created](#).

4. Click **Edit**.
5. Change the Tracking Group Type to **External**.



Note A specific Omniscript or Flexcard can belong to two Tracking Groups if one is Internal and the other is External.

6. Click in the External Tracking Config Definition text box and select **GoogleAnalyticsTest** from the list.
7. Save your changes.
8. Refresh the browser.

Omnistudio

Take advantage of Omnistudio's declarative 'clicks not code' approach to building and modifying your applications with the Omnistudio Lightning Web Components. With Omnistudio Lightning Web Components, use standard JavaScript and HTML to modify and extend Vlocity products.

Omnistudio Lightning Web Components are:

- Fast and Lightweight. Runs natively in browsers and is independent from JavaScript frameworks.
- Reusable. Omnistudio Lightning Web Components use [web components](#) to create reusable custom HTML elements. Custom elements wrap functionality, protecting components from other styles and scripts on the page.

While some Omnistudio Lightning web components appear in the Community and App Builders, they differ from the Omnistudio Lightning Components that also appear in the builders. Omnistudio Lightning Web Components follow the same standards as Salesforce's Lightning Web Components. For more information on Lightning Web Component standards, see [Lightning Web Components](#).

Omnistudio Lightning Web Components Reference

Component Type	Description	References
Base	Basic components used by the entire Vlocity platform. Extend or modify base components to customize appearance and behavior.	ReadMes: Base Omnistudio LWC ReadMe Reference
Omniscript	Components specific to Omniscript.	ReadMes: Omniscript ReadMe Reference

[Set Up Lightning Web Components](#)

To manage and develop Omnistudio Lightning web components, use Visual Studio Code with Salesforce DX and the SF CLI. Use to deploy components from one sandbox or dev org to another, such as a production org.

[Extend Omnistudio Lightning Web Components](#)

Customize the behavior and styling of an application by extending Omnistudio Lightning web components. For example, override properties, add other components, or insert HTML.

[Deploy Lightning Web Components](#)

Deploy new Lightning web components or change existing components from your local development environment with Visual Studio and Salesforce DX or IDX Workbench.

[Base Omnistudio LWC ReadMe Reference](#)

View examples, and learn about available attributes, methods, and other functions from the ReadMes of each Base Omnistudio Lightning web component.

Set Up Lightning Web Components

To manage and develop Omnistudio Lightning web components, use Visual Studio Code with Salesforce DX and the SF CLI. Use to deploy components from one sandbox or dev org to another, such as a production org.

- For recommended deployment tools, see [Deploy Omnistudio Components Between Orgs](#).
- To manage and develop your LWCs with Salesforce DX, see [Set Up Your Development Environment](#).

Extend Omnistudio Lightning Web Components

Customize the behavior and styling of an application by extending Omnistudio Lightning web components. For example, override properties, add other components, or insert HTML.

 **Note** If you turn off Managed Package Runtime in Setup and use the standard runtime, we recommend that you don't extend Lightning web components such as Block or TypeAhead. Changing these components can cause incompatibilities between Omnistudio and Omnistudio for Managed Packages.

In this code example, a custom Lightning web component extends the Button Lightning Web Component. Replace the *namespace* variable in the code example with the namespace of the Omnistudio package you're using. For Omnistudio in standard runtime, the namespace is `omnistudio`.

`//.js`

```
import Button from "namespace/button";

export default class buttonExtended extends Button {
    //override the property here so it gets triggered

    onclickbutton() {
        this.label = "Button clicked";
    }
}
```

`//.js-meta.xml`

```
<?xml version="1.0" encoding="UTF-8"?><LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>45.0</apiVersion>
```

```
<isExposed>true</isExposed>
<masterLabel>button_extended</masterLabel>
<description>Button extended</description>
<targets>
    <target>lightning__RecordPage</target>
    <target>lightning__AppPage</target>
    <target>lightning__HomePage</target>
</targets>
<runtimeNamespace>namespace</runtimeNamespace>
</LightningComponentBundle>
```

//.html

```
<template>
//add HTML here to override the template layout
</template>
```

//_slds.css

```
//add CSS to override or append the SLDS theme css
.slds-button {
    background: #cccccc;
    border-color: #dddddd;
}
```

- Note** Custom Lightning web components built outside of the package can't use any Salesforce Lightning web component that uses Salesforce resources or affects the component at run time. For more information, see [Salesforce Modules](#).
- Note** Custom Lightning web components don't throw errors unless Debug Mode is enabled. For more information, see [Debug Lightning Web Components](#).

1. Ensure you have IDX Workbench, or Salesforce DX set up locally. For information on setting up IDX Workbench or Salesforce DX, see [Set Up Lightning Web Components](#).
2. Choose which component you want to extend. For a list of Lightning web components, see [Omnistudio Lightning Web Components](#).
3. To create a Lightning web component, navigate to your **lwc** folder in your project and run the `lightning generate component` Salesforce CLI command. For example:

```
sf lightning generate component --type lwc --name componentname_extended
```

To learn more about creating a Lightning web component, see [Create Lightning Web Components](#). For a complete list of available Salesforce CLI commands, see [Lightning Commands](#).

4. In your JavaScript file, import and extend the Lightning web component. See code example on this page.
 5. To make the custom Lightning web component compatible with Omnistudio Lightning web components, you must set two metadata tags in your XML configuration file:
 - If [Lightning Web Security \(LWS\)](#) is disabled, add the namespace of your Omnistudio package using the **runtimeNamespace** metadata tag. See the code example on this page. For more information on finding the namespace of your package, see [View the Namespace and Version of Managed Packages](#).
- !** **Important** If LWS is enabled, setting **runtimeNamespace** in your components causes errors similar to "Cannot use runtime namespace 'somenamespace' in module c-someComponentName". To determine if LWS is enabled in your org, see [Enable Lightning Web Security in an Org](#).
- Set the **isExposed** metadata tag to true. See code example on this page.
6. Enable a custom Lightning web component to make remote calls by using the Common Action utility. See [Make Remote Calls Within Omniscripts from Lightning Web Components](#).

Deploy Lightning Web Components

Deploy new Lightning web components or change existing components from your local development environment with Visual Studio and Salesforce DX or IDX Workbench.

To set up your development environment for managing and deploying your components, see [Set Up Lightning Web Components](#).

1. Open your project in Visual Studio.
2. Make sure you have Salesforce DX or IDX Workbench installed. To install Salesforce DX or IDX Workbench, see [Salesforce DX Setup Guide](#).
3. In the terminal, run the following command to deploy changes to your org:

```
sf project deploy start
```

For more on deploying components to your org with Visual Studio, see [Analyze Your Code and Deploy It to Your Org](#).

Base Omnistudio LWC ReadMe Reference

View examples, and learn about available attributes, methods, and other functions from the ReadMes of each Base Omnistudio Lightning web component.

To extend components, see [Extend Omnistudio Lightning Web Components](#).

All Omnistudio component HTML, CSS, and read-me files are available when you [Set Up Your Environment to Customize Omniscript Elements](#). Check the folder for each component to find these files.

Omnistudio

Use Omnistudio to enhance your Experience Cloud sites by showing key contextual information and creating dynamic interactions.

In addition to setting up Omnistudio, you must complete specific tasks before you use Omnistudio on Experience Cloud such as provisioning user access, and designing and deploying Omnistudio components.

This topic gives a holistic view of Omnistudio on Experience Cloud, featuring high-level links. For comprehensive knowledge, follow the detailed tasks in the nested topics.

Learn About Experience Cloud

If you're new to Experience Cloud, follow these links to get started.

- Trailhead: [Experience Cloud Basics](#)
- Trailhead: [Experience Cloud Site Strategy](#)
- Trailhead Project: [Set Up a Customer Site with Experience Cloud](#)
- [Learn Experience Cloud](#)
- Salesforce Help: [Experience Cloud](#)

Set Permissions

Though both authenticated and guest users can use Omnistudio on Experience Cloud, the permissions and sharing-related rules needed for each user type can vary.

- Licenses and permission sets that are required to access Omnistudio on Experience Cloud: [Omnistudio Permission Sets](#)
- Authenticated user access: [Setup Omnistudio Standard Permission Sets for Experience Site Users](#)
- Guest user access: [Grant Digital Experience Guest Users Omnistudio Access](#)

Work with Flexcards

Use Flexcards on Experience Cloud pages to build dynamic UI components and show key information at a glance.

- Learn about publishing Flexcards on Experience Cloud sites: [Activate and Publish a Flexcard](#)

- To get data from an Apex remote, Omnistudio Data Mapper, or Integration Procedure data source, most Flexcards use a context ID represented by the {recordId} context variable in an input map. To learn more, see Access the Context ID of a Flexcard on an Experience Site in the [Examples for Flexcard Settings](#) topic.
- Images used in Flexcards on Experience Cloud pages must be static resources. See [Add an Icon or an Image to a Flexcard](#).
- The Navigate and Omniscript actions don't work in preview. To view them, add them to a Lightning or Experience Builder page. For more information, see [Preview and Debug a Flexcard](#).
- Use Flexcards on LWR sites: [Add a Flexcard to an Experience Builder Page](#)
- [Considerations for Using Flexcards on Lightning Web Runtime Sites](#)

Work with Omniscripts

Use Omniscripts on Experience Cloud to build UI components that can interact with your Experience Cloud users.

- Learn about deploying and embedding Omniscripts on Experience Cloud sites: [Activate and Launch Omniscripts](#)
- Use Omniscripts on LWR sites: [Considerations for Using Omniscripts with Lightning Web Runtime Sites](#)
- [Add Your Omniscript to an Experience Cloud Page](#)
- To use features such as linking between an Omniscript and an Experience Cloud page, use page reference types: [Page Reference Types](#)

Omnistudio

View the number of Omnistudio calls that are available for your org and how many are remaining.

To see the number of Omnistudio calls that you used up, check the Usage-Based Entitlements setup configuration. See [View Your Salesforce Org's Usage-Based Entitlements](#).

To check your usage, in the Resources column, search for Maximum omnistudio calls allowed for an org.