Custom Lightning Types

Developer Preview, TDX

Salesforce, Spring ’25

@salesforcedocs

Last updated: March 05, 2025

Important: Custom Lightning Types are available as a **developer preview**. Custom Lightning Types aren't generally available unless or until Salesforce announces its general availability in documentation or in press releases or public statements. All commands, parameters, and other features are subject to change or deprecation at any time, with or without notice. Don't implement functionality developed with these commands or tools in production environments.

CONTENTS

CONTENTS 3
Overview 5
Introduction to Lightning Types 5
Building Blocks of a Lightning Type 5
Standard Lightning Types 5
Lightning Types in Agent Action 6
Example: Standard Lightning Types in Agent Action 7
Custom Lightning Types 11
UI Comparison: Before and After 11
LightningTypeBundle Metadata Type 12
Understand the LightningTypeBundle Structure 12
Use Metadata API to Deploy LightningTypeBundles 13
The Schema.json File 14
The Editor.json File 15
Full Editor Override 15
Example 15
LWC Attribute Mapping 16
The Renderer.json File 17
Full Renderer Override 17
Example 17
LWC Attribute Mapping 18
Example: Customizing user interface with Custom Lightning Types 19
Before You Begin 20
Apex Class 20
Create Agent Action by Using Apex Class 22
Agent Action Execution Input 23
Agent Action Execution Output 24
Result Data 25
Customize UI for Output 26
Override Default UI for Output With Custom Lightning Types 26
Build Output Components with Lightning Web Components 27
Integrate Custom Lightning Type into Agent Action Output 30
Customized Output UI 31
Customize UI for Input 32
Override Default UI for Input with Custom Lightning Types 32
Build Input Components with Lightning Web Components 33

Integrate Custom Lightning Type into Agent Action Input....36
Customized Input UI.... 37
Known Limitations....38
Important Notices.... 38
References.... 39
Lightning Types....39
lightning__objectType.... 39
lightning__booleanType....39
lightning__dateType....40
lightning__dateTimeType....40
lightning__integerType....42
lightning__numberType....42
lightning__richTextType.... 43
lightning__textType....44
lightning__multilineTextType....45
lightning__urlType....45
LWC Target Types for Agentforce Components....46
lightning__AgentforceInput.... 46
Syntax.... 46
targetConfigs....46
targetConfig....47
Property....47
lightning__AgentforceOutput.... 48
Syntax.... 48
targetConfigs....48
targetConfig....48
Property....49

Overview

Discover how to use custom Lightning types to improve the user interface of custom agent actions for Agentforce (Default) in Lightning Experience. Custom Lightning types are particularly effective for handling complex inputs and outputs.

This guide helps you:

- Understand the steps involved in creating a LightningTypeBundle.
- Define the JSON schema for Apex-based custom Lightning types.
- Build editor and renderer components by using Lightning web components.
- Deploy the bundle by using Metadata API.
- Integrate custom Lightning types into custom agent actions to override the default UI for input and output in Agentforce (Default) in Lightning Experience.

Introduction to Lightning Types

Lightning Types are JSON-based data types to structure, validate, and display data for the default Agentforce agent in Lightning Experience.

With Lightning Types, you can manage the representation and shape of data types. You can use Lightning types to create consistent and flexible data interactions in Lightning Experience.

Salesforce provides standard Lightning types, such as text and multiline text, to structure your data type. Additionally, you can create Lightning types to customize the UI experience for Agentforce (Default) in Lightning Experience based on your business requirements.

Building Blocks of a Lightning Type

Lightning Type consists of these artifacts.

1. **Schema** defines the structure of data and the rules for its validation, such as maximum length, type, and format.
2. **Editor** defines the input UI component that you use to enter or edit data.
3. **Renderer** defines the output UI component that displays data.

Standard Lightning Types

Salesforce provides some Lightning types out of the box that act as the basic types that you can reference to structure a more complex schema.

To understand how a Lightning type is validated, you must identify the underlying type used for each Lightning type.

Each standard Lightning type includes a default editor and renderer, so there's no need for you to create those components.

Similar to JSON Schema types, each Lightning type has its own type-specific keywords that apply only to that type.

You can use these standard Lightning types.

- lightning__objectType
- lightning__booleanType
- lightning__dateType
- lightning__dateTimeType
- lightning__dateTimeStringType
- lightning__integerType
- lightning__numberType
- lightning__richTextType
- lightning__textType
- lightning__multilineTextType
- lightning__urlType

For information about the keywords available and the default editor and renderer associated with each of the Lightning types, see Lightning Types.

Lightning Types in Agent Action

Agent actions use standard Lightning types to define the structure, validate, and display of data in Salesforce when an action is triggered.

Here's how Lightning types are used in the context of agent actions.

- **Mapping Data Types to Lightning Types**

  In Salesforce, Apex classes are often used to handle business logic, such as processing inputs and returning results. For example, consider an agent action called Flight Booking that searches for available flights. When you trigger this agent action, the inputs and outputs from the Apex class are mapped to standard Lightning types. So if an Apex class accepts inputs like dates, strings, and numbers, these data types are mapped to the corresponding standard Lightning types, such as `lightning__dateType`, `lightning__stringType`, and `lightning__numberType`. This mapping ensures that the data is structured correctly.

- **Schema Definition and Validation**

  Each standard Lightning type has an associated schema that defines the structure of the data and the rules for its validation, such as maximum length and format.

  This schema ensures that the data that you enter to the action conforms to the expected type and format, which helps to avoid errors during execution.

For example, if an action expects you to enter Date data, the schema ensures that you enter only a valid date.

- **Automatic UI Generation**

  When you trigger an action, Salesforce automatically generates the appropriate UI components for the action's inputs and outputs based on the mapped Lightning types. The UI displays relevant input fields, pickers, or tables according to the standard Lightning type.

  For example:

  - If an Apex action expects a multiline text input, a multiline text field appears in the UI, based on the standard Lightning type `lightning__multilineTextType`.
  - For date fields, a date picker automatically appears in the UI, based on the standard Lightning type `lightning__dateType`.

- **Rendering the Data**

  The renderer component associated with each Lightning type displays the data when an action is executed.

  For example:

  - For an Apex class that returns a list of flights, you can use the standard Lightning type `lightning__listType` to display the flight data in an appropriate format.

  This rendering ensures that the output is displayed in a structured and readable way, whether it's a list, table, or simple text.

- **Out-of-the-Box Components**

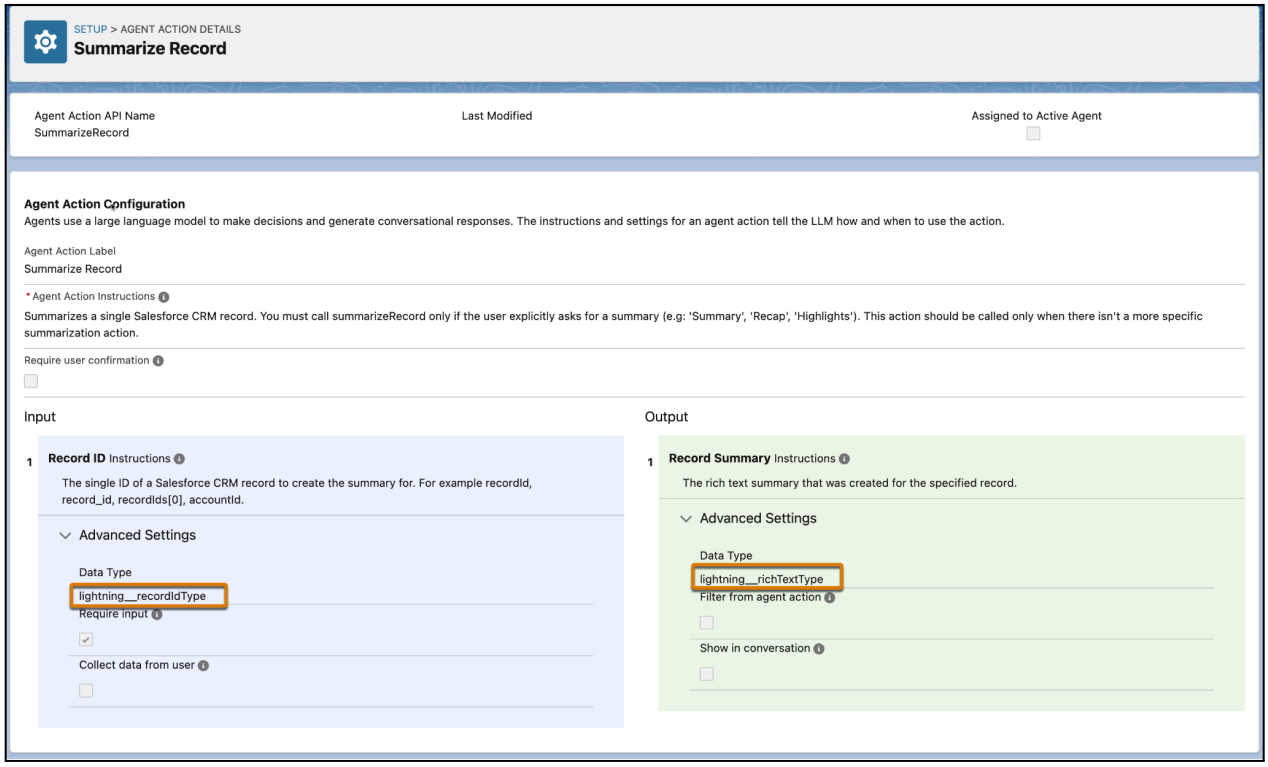
  When you use standard Lightning types, Salesforce provides ready-to-use components for input and output with minimal configuration required. These components handle most use cases, offering a seamless experience when working with standard Lightning types in agent actions.

Example: Standard Lightning Types in Agent Action

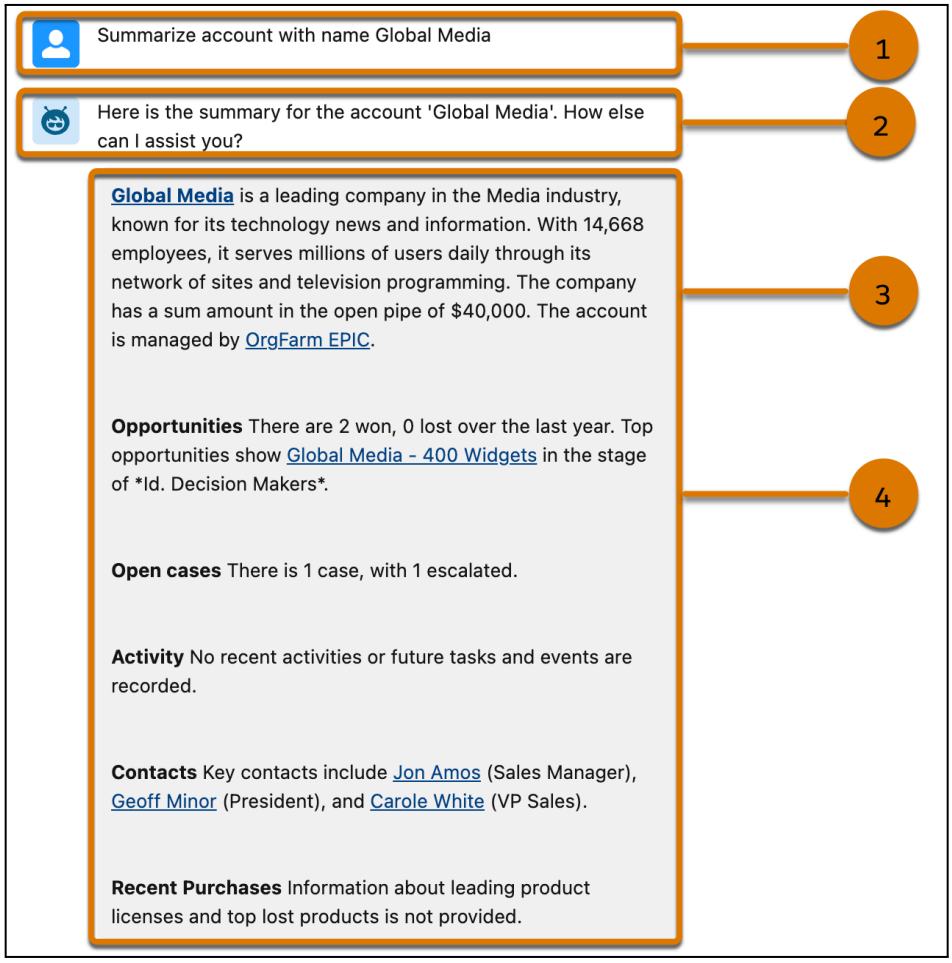
Here's how agent actions output is displayed by using standard Lightning types.

Agent Action: Summarize Record

This image shows the standard Lightning types that the agent action uses to display the records summary.



Here’s how a record summary appears in the action output.

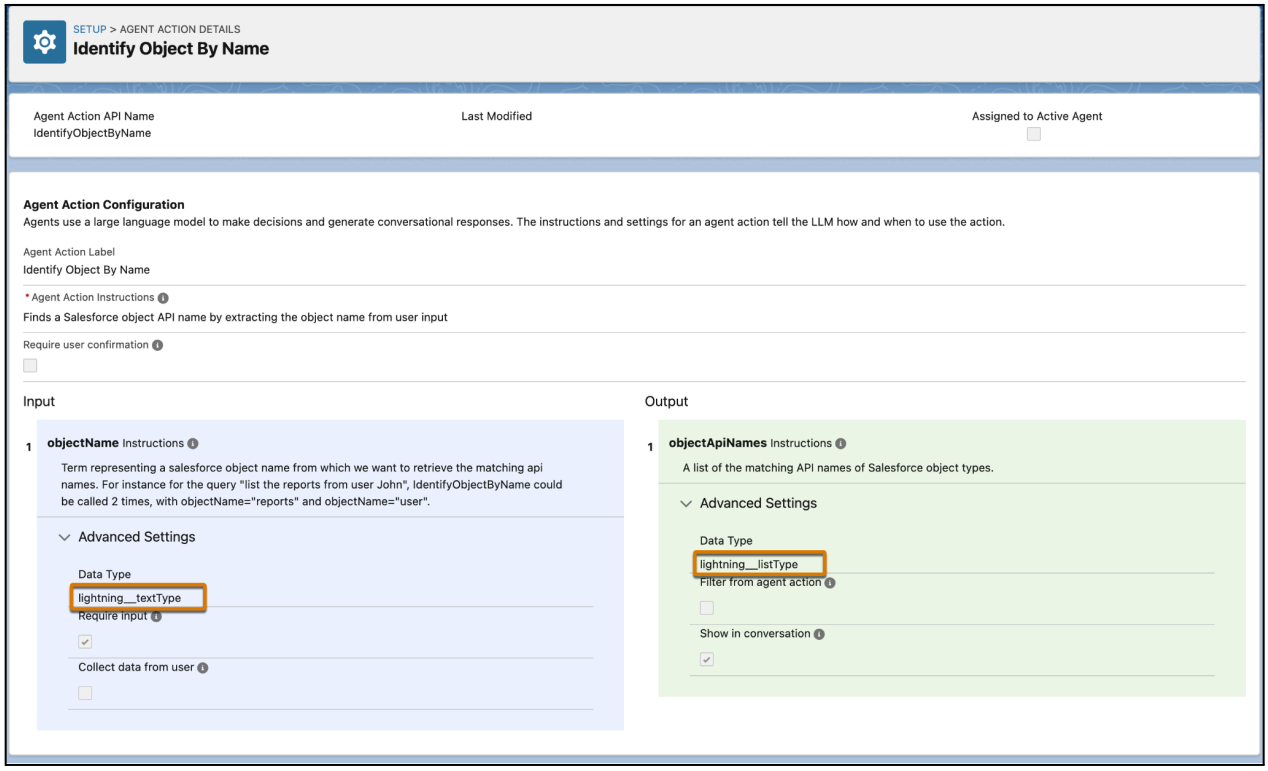


The agent action output has these elements.

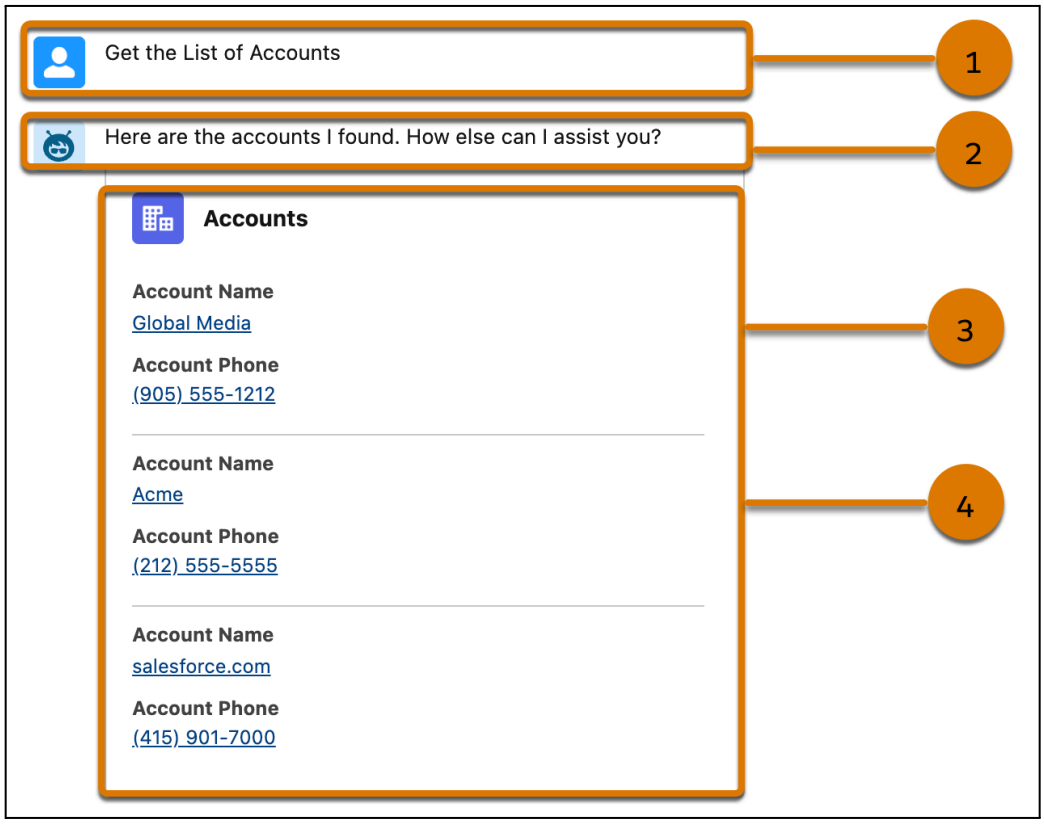
1. Utterance
2. Agent Response
3. Agentforce Component (LWC)
4. Agentforce Action Output/Input Type (Lightning Type)

Agent Action: Identify Objects By Name

This image shows the standard Lightning types that the agent action uses to show the list of accounts.



Here's how the list of accounts appears in the action output.



The agent action output has these elements.

1. Utterance
2. Agent Response
3. Agentforce Component (LWC)
4. Agentforce Action Output /Input Type (Lightning Type)

Custom Lightning Types

Create custom Lightning types to customize the appearance of the UI for Agentforce (Default) in Lightning Experience. With custom Lightning types, you override the default user interface to manage complex interactions within Salesforce.

To create custom lightning types in Salesforce, use the LightningTypeBundle metadata component. For information about how to create custom Lightning types, see LightningTypeBundle Metadata API.

Benefits:

- **Enhanced UI Customization**

  Standard Lightning types have predefined UI components. However, they don't always fit your design needs or the user experience. Custom Lightning types give you full control over the UI. You can create tailored components that match your specific styling and behavior requirements. This customization ensures that the interface looks and functions exactly as you need for your application.

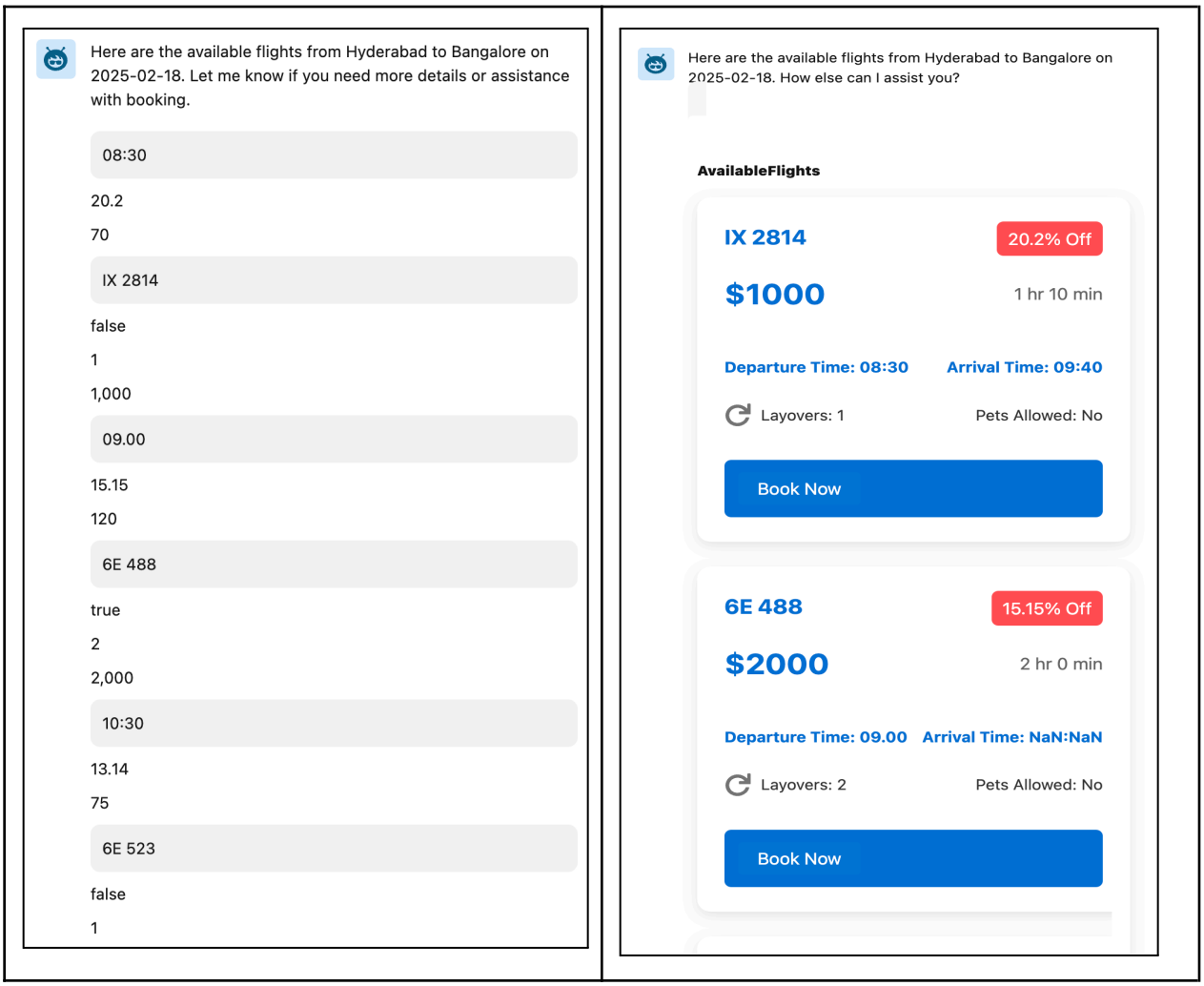
- **Handling Complex Data Structures**

  Standard Lightning types sometimes can't handle complex data, but custom Lightning types can manage and render complex data structures. For example, they can handle deeply nested objects, complex arrays, and dynamic fields that change based on user input. With custom Lightning types, you can build UIs that display complex data smoothly. Customize your UI to handle complex data structures, and you ensure that your Salesforce actions can accommodate even the most detailed and dynamic workflows.

UI Comparison: Before and After

These screenshots demonstrate the improvements in the agent action output UI achieved through custom Lightning types.

The default UI for output in an agent action	The customized UI for output in an agent action



LightningTypeBundle Metadata Type

The LightningTypeBundle metadata type describes the custom Lightning types. It's available in API version 63.0 and later.

To get a list of the custom and standard Lightning types deployed in your org, make a call to the `connect/lightning-types` resource.

For more information about the resources available in the Type System Connect REST API, see the [Type System Resources](#).

Understand the LightningTypeBundle Structure

LightningTypeBundle components are stored in the lightningTypes folder.

Here's an example of the LightningTypeBundle structure.

```
+--myMetadataPackage
    +--lightningTypes (1)
        +--TypeName (2)
           +--schema.json (3)
           +--lightningDesktopGenAi (4)
              +--editor.json (5) OR +--renderer.json (6)
```

The bundle includes these resources.

- The `lightningTypes` folder (1) contains a folder for each custom Lightning type created in the format `TypeName` (2).
- Each custom Lightning type folder contains a `schema.json` file (3) that defines the JSON schema that drives the custom Lightning type validation.
- If applicable, the custom Lightning type folder also contains a `lightningDesktopGenAi` folder (4) with two files that indicate the optional artifacts needed for the `lightningDesktopGenAi` channel. Configure these files to override the default UI of a custom Lightning type when it's used in an agent action.
    - The `editor.json` file (5) has custom user interface and editor information.
    - The `renderer.json` file (6) has custom user interface and renderer information.

Use Metadata API to Deploy LightningTypeBundles

To deploy a LightningTypeBundle to your Salesforce org, use Metadata API. The Metadata API uses a manifest file that defines the metadata that you want to deploy.

Here's an example `package.xml` manifest file for a LightningTypeBundle that includes the custom Lightning type `myFlight`.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>myFlight</members>
        <name>LightningTypeBundle</name>
    </types>
    <version>63.0</version>
</Package>
```

To delete a custom Lightning type, you must deploy a destructiveChanges package to your org that lists the types to delete.

See Also

- *Metadata API Developer Guide*: Deploying and Retrieving Metadata with the Zip File
- *Metadata API Developer Guide*: Deleting Components from an Organization

The Schema.json File

The `schema.json` file uses the JSON Schema Specification to define your custom Lightning type. The schema consists of a specific set of keywords that apply constraints to the data.

This table describes the keywords that you can specify in a `schema.json` file.

Keyword	Required or Optional	Type	Description
title	Required	String	Name for the Lightning type
description	Optional	String	Description for the Lightning type
lightning:type	Required	String	Refers to `@apexClassType` types by using fully qualified names. This keyword is syntactic sugar for the `$ref` keyword in JSON Schema, which links together schemas. For information about `$ref`, see Understanding JSON Schema: The Sref keyword.

Unless noted otherwise, the keywords follow the JSON Schema specification.

Here's a sample code that shows the contents of the schema.json file for a custom Lightning type `flightResponse`.

```
{
  "title": "My Flight Response",
  "description": "My Flight Response",
  "lightning:type": "@apexClassType/c__FlightAgent$AvailableFlight"
}
```

The Editor.json File

An optional file that you can configure for a custom Lightning type that you create. With this file, you can customize how user input is collected by defining the input component.

Full Editor Override

You can define a single editor for your entire custom Lightning type.

Let's say that you have a custom Lightning type named `flightFilter` with a `schema.json` file. For information about how to create custom lightning types, see LightningTypeBundle Metadata API.

You configure `editor.json` to define an editor for your entire custom Lightning type. As a result, the same editor applies whenever you use the instance of this type as input.

Here's an example of what the `editor.json` file can look like for the custom Lightning type `flightFilter`.

```
+--lightningTypes
        +--flightFilter
            +--schema.json
            +--lightningDesktopGenAi
               +--editor.json
```

Example

To illustrate the concept, this example uses the Apex class.

The Filter class contains two fields: `price` and `discountPercentage`.

By default, individual out-of-the-box editors are displayed to collect user input. Use `c/myFilterComponent` to collect user input related to the flight search filter criteria.

For example, you use a slider to select a value for `Discount  Percentage`, and you apply minimum and maximum limits for `Price`.

Let's create a custom LWC component, `myFilterComponent`, that contains the fields `price` and `discountPercentage`.

Here's a sample code that shows the LWC component `myFilterComponent`.

```
import { LightningElement } from 'lwc';
export default class MyFilterComponent extends LightningElement {
    @api
    price = 0;
    @api
```

```
    discountPercentage = 0;
   …
   …
}
```

Reference this component in the `editor.json` file.

Here's how to reference the LWC component `myFilterComponent` to override the editor for the `Filter` input in the `editor.json` file.

```
{
  "editor": {
    "componentOverrides": {
      "$": {
        "definition": "c/myFilterComponent"
      }
    }
  }
}
```

Note: To specify full editor override, use the "$" keyword in your `editor.json` file.

LWC Attribute Mapping

Let's say that you built the LWC component `myExistingFilterComponent` that contains the fields with the names `cost` and `discountPercentage`.

```
import { LightningElement } from 'lwc';
export default class MyExistingFilterComponent extends LightningElement {
    @api
    cost = 0;
    @api
    discountPercentage = 0;
   …
   …
}
```

You decide to reuse the `myExistingFilterComponent` instead of creating a new one, `myFilterComponent`, with field names `price` and `discountPercentage`. However, the field name `price` in the Flight class doesn't match the field name `cost` in `myExistingFilterComponent`.

To map the fields from `Flight` class to the corresponding fields in the LWC component `myExistingFilterComponent`, use attribute mapping.

Here's a sample code that shows how to reference `myExistingFilterComponent` to override the editor for the `Filter` input in the editor.json file with attribute mapping.

```
{
  "editor": {
    "componentOverrides": {
      "$": {
        "definition": "c/myExistingFilterComponent"
        “attributes”: {
           “cost”: “{!$attr.price}”
        }
      }
    }
  }
}
```

The expression “`cost”: “{!$attr.price}`” indicates:

- `cost:` Field in the LWC component `myExistingFilterComponent`
- `{!$attr}:` Pointer to the `Filter` class field
- `price:` Field in the `Filter` class
- `{!$attr.price}:` Links the `price` field of the `Filter` Apex class to the `cost` field of the LWC component `myExistingFilterComponent` and vice versa.

The Renderer.json File

An optional file that you can configure for a custom Lightning type that you create. With this file you can customize how data is presented to the user by defining the output component.

Full Renderer Override

You can define a single renderer for your entire custom Lightning type.

Let’s say that you have a custom Lightning type named `flightFilter` with the `schema.json` file. For information about how to create custom Lightning types, see LightningTypeBundle Metadata API.

You configure `renderer.json` to define a renderer for your entire custom Lightning type. As a result, the same renderer applies whenever you use the instance of this type as output.

Here’s an example of what the `renderer.json` file can look like for the custom Lightning type `flightFilter`.

```
+--lightningTypes
        +--flightFilter
            +--schema.json
            +--lightningDesktopGenAi
               +--renderer.json
```

Example

To illustrate the concept, this example uses the Apex class.

The `Filter` class contains two fields: `price` and `discountPercentage`.

By default, individual out-of-the-box renderers are displayed as output to the user. Use `c/myFilterRenderer` to display flight details.

Color coding can help represent discount percentages based on their values. For example, the higher the percentage, the more prominent the color used to display the discount.

Let's create a custom LWC component, `myFilterRenderer`, that contains the fields `price` and `discountPercentage`.

Here's a sample code that shows the LWC component `myFilterRenderer`.

```
import { LightningElement } from 'lwc';
export default class MyFilterRenderer extends LightningElement {
    @api
    price = 0;
    @api
    discountPercentage = 0;
   …
   …
}
```

Reference this component in the `renderer.json` file.

Here's how to reference the LWC component `myFilterRenderer` to override the renderer for the `Filter` output in the renderer.json file.

```
{
  "renderer": {
    "componentOverrides": {
      "$": {
        "definition": "c/myFilterRenderer"
      }
    }
  }
}
```

Note: To specify full renderer override, use the "$" keyword in your renderer.json file.

LWC Attribute Mapping

Let's say that you built the LWC component `myExistingFilterRenderer` that contains fields with the names `cost` and `discountPercentage`.

```
import { LightningElement } from 'lwc';
export default class MyExistingFilterRenderer extends LightningElement {
```

```
    @api
    cost = 0;
    @api
    discountPercentage = 0;
   …
   …
}
```

You decide to reuse the `myExistingFilterRenderer` component instead of creating a new one, `myFilterRenderer`, with field names `price` and `discountPercentage`. However, the field name `price` in the Flight class doesn't match the field name `cost` in `myExistingFilterRenderer`.

To map the fields from `Flight` class to the corresponding fields in the LWC component `myExistingFilterRenderer`, use attribute mapping.

Here's a sample code that shows how to reference the LWC component `myExistingFilterRenderer` to override the renderer for the `Filter` output in the renderer.json file with attribute mapping.

```
{
  "renderer": {
    "componentOverrides": {
      "$": {
        "definition": "c/myExistingFilterRenderer"
        “attributes”: {
           “cost”: “{!$attr.price}”
        }
      }
    }
  }
}
```

The expression `“cost”: “{!$attr.price}”` indicates:

- `cost:` Field in the LWC component `myExtistingFilterRenderer`
- `{!$attr}:` Pointer to the `Filter` class field
- `price:` Field in the `Filter` class
- `{!$attr.price}:` Links the `price` field of the `Filter` Apex class to the `cost` field of the LWC component `myExtistingFilterRenderer` and vice versa

Example: Customizing user interface with Custom Lightning Types

Here's an example that explains how to override the default user interface to create a customized appearance of responses on the custom agent's action input and output with custom lightning types.

Before You Begin

Download these sample data files.

- apexClass.zip
- flightDetailsLWC.zip
- flightResponseCLT.zip
- flightFiltersCLT.zip
- flightFiltersLWC.zip

Apex Class

This section includes the Apex class `FlightAgent` that retrieves flight information based on your request.

Use this Apex class to create a custom agent action.

```
public class FlightAgent {

    @InvocableMethod(label='Find Flights' description='Finds available flights')
    public static List<FlightResponse> findFlights(List<FlightRequest> req) {
             List<FlightResponse> flightResponses = new List<FlightResponse>();

        // Hardcoding the data for example and not focusing on how we retrieve it.
        // However, consider that we are receiving available flights from a service,
        // and then iterating through the data to generate the final response.

        List<Flight> flights = new List<Flight>();
        Flight f1 = new Flight('IX 2814', 1, false, 1000l, 20.20d, 70);
        Flight f2 = new Flight('6E 488', 2, false, 2000l, 15.15d, 120);
        Flight f3 = new Flight('6E 523', 1, false, 3000l, 13.14d, 75);
        Flight f4 = new Flight('6E 6166', 2, false, 4000l, 14.14d, 130);
        flights.add(f1);  flights.add(f2); flights.add(f3); flights.add(f4);
        AvailableFlight availableFlights = new AvailableFlight();
        availableFlights.flights = flights;

        FlightResponse fr = new FlightResponse();
        fr.aFlight = availableFlights;
        flightResponses.add(fr);

        return flightResponses;
    }

    public class FlightRequest {

        @InvocableVariable
        public String originCity;

        @InvocableVariable
        public String destinationCity;

        @InvocableVariable
        public Date dateOfTravel;

        @InvocableVariable
```

```
        public Filter filters;
    }

    public class Filter {

        @InvocableVariable
        public Long price;

        @InvocableVariable
        public Double discountPercentage;
    }


    public class FlightResponse {
        @InvocableVariable
        public AvailableFlight aFlight;
    }

    public class AvailableFlight {
        public List<Flight> flights;
    }

    public class Flight {

        @InvocableVariable
        public String flightId;

        @InvocableVariable
        public Integer numLayovers;

        @InvocableVariable
        public Boolean isPetAllowed;

        @InvocableVariable
        public Long price;

        @InvocableVariable
        public Double discountPercentage;

        @InvocableVariable
        public Integer durationInMin;

        public Flight(String flightId, Integer numLayovers, Boolean isPetAllowed,
                      Long price, Double discountPercentage, Integer durationInMin) {
            this.flightId = flightId;
            this.numLayovers = numLayovers;
            this.isPetAllowed = isPetAllowed;
            this.price = price;
            this.discountPercentage = discountPercentage;
            this.durationInMin = durationInMin;
        }
    }
}
```

This Apex class accepts the flight search criteria, including the origin city, destination city, and date of travel, and then returns a list of available flights.

Note: For this example, flight availability data is already included in the Apex class. However, in a real-time scenario, flight information is fetched from an external service, and the Apex class processes that data to generate the final response.

Create Agent Action by Using Apex Class

For information about how to create a custom action by using Apex class, see Create a Custom Agent Action.

Inputs and outputs for the agent action are defined by using standard Lightning types and Apex classes.

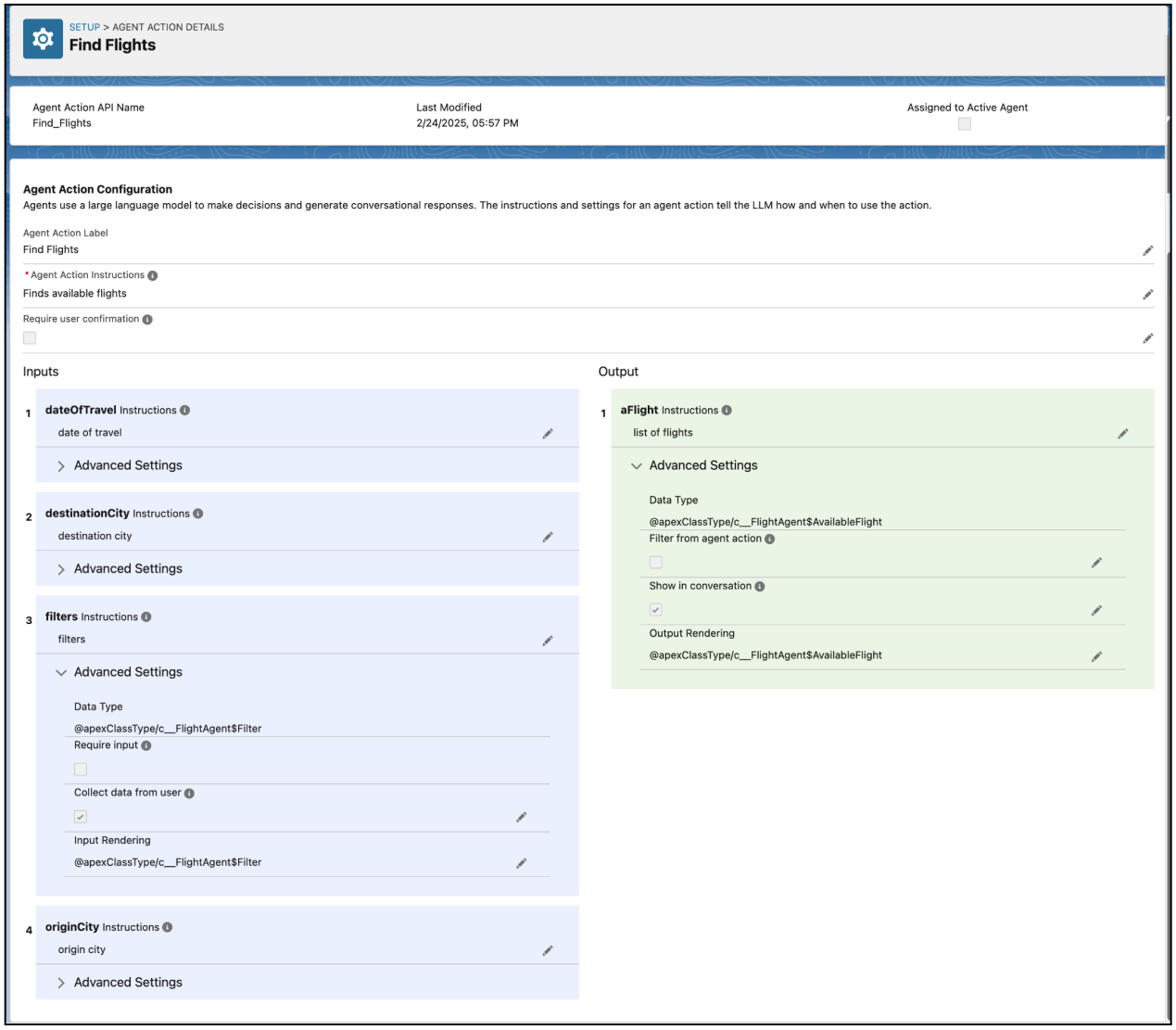
Input:

- `dateOfTravel`, `destinationCity`, and `originCity` use standard Lightning types such as `lightning__dateType` and `lightning__textType`.
- The `filters` input is a complex type that references an Apex class.

Output:

- The output `aFlight` for the agent action is a complex type that references an Apex class.

Here's an image that shows the custom agent action created.



The available flight information is retrieved by using `@apexClassType/c__FlightAgent$AvailableFlight` in the agent action output, where:

- `apexClassType` is the bundle name.
- `FlightAgent` is the parent class.
- `AvailableFlight` is a nested Apex class within `FlightAgent`.

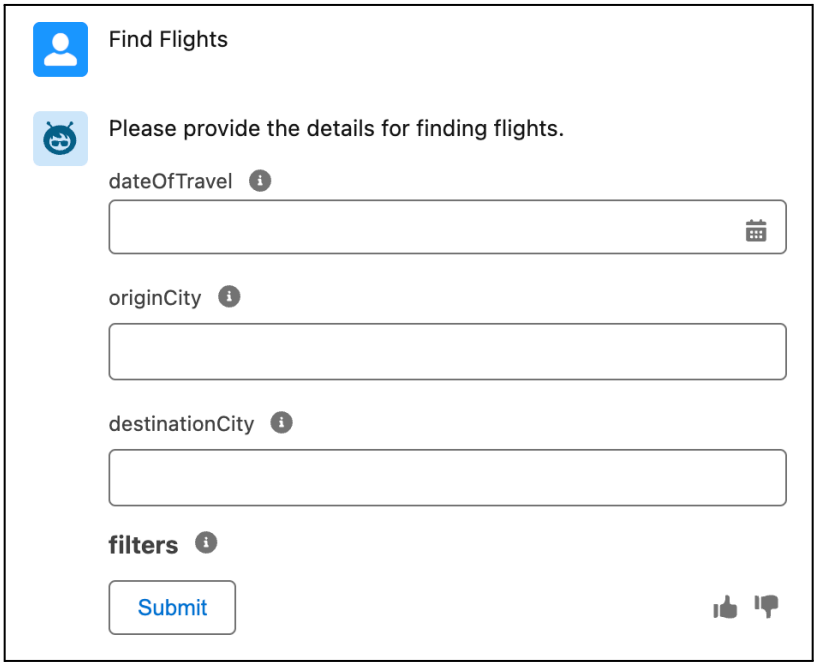
When you execute this agent action, it prompts you to provide input and then generates the output.

Agent Action Execution Input

The agent's action UI collects these details to find available flights.

- Origin city
- Destination city
- Date of travel

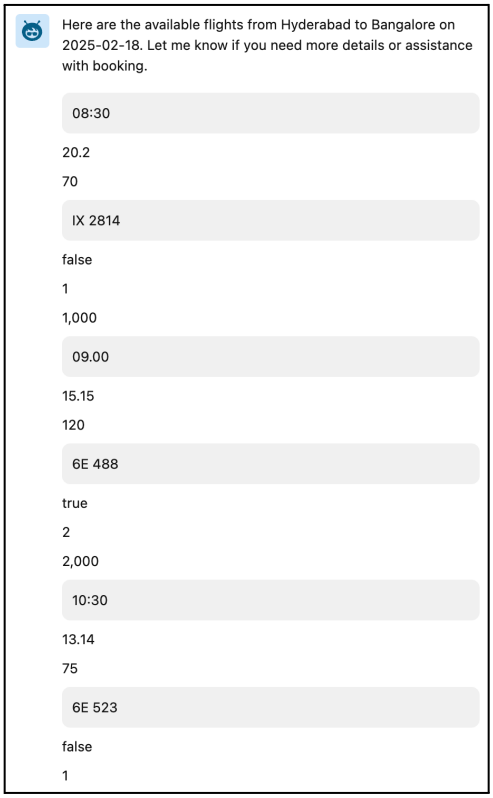
Here's the image that shows how the custom agent action input appears in an agent conversation.



Agent Action Execution Output

The agent's action UI returns the available flight details.

Here's the image that shows how the custom agent action's output appears in an agent conversation.



Result Data

The agent displays the flight data in the response.

Here's the sample code that shows the available flight data.

```
{
  "aFlight": {
    "flights": [
      {
        "price": 1000,
        "numLayovers": 1,
        "isPetAllowed": false,
        "flightId": "IX 2814",
        "durationInMin": 70,
        "discountPercentage": 20.2,
        "departureTime": "08:30"
      },
      {
        "price": 2000,
        "numLayovers": 2,
        "isPetAllowed": true,
        "flightId": "6E 488",
```

```
        "durationInMin": 120,
        "discountPercentage": 15.15,
        "departureTime": "09.00"
      }
    ]
  }
}
```

Customize UI for Output

Create a custom Lightning type named `flightResponse` to enhance the visibility of the information in the output UI.

Override Default UI for Output With Custom Lightning Types

Override the agent's action UI for output to enhance the user experience by using Custom Lightning Types (CLTs). With CLTs, you can add your own Lightning Web Components (LWC) to present data in a more structured and intuitive format.

Configure the renderer.json file to override the default UI of a custom Lightning type in the agent action.

Here's an example showing a lightningTypes folder for a custom Lightning type named `flightResponse`.

```
+--lightningTypes
        +--flightResponse
            +--schema.json
            +--lightningDesktopGenAi
               +--renderer.json
```

The custom Lightning type `flightResponse` includes a `schema.json` file and a `renderer.json` file. The `renderer.json` file controls how the data is displayed to the user in the agent action output.

This sample code shows the contents of the `schema.json` file.

```
{
  "title": "My Flight Response",
  "description": "My Flight Response",
  "lightning:type": "@apexClassType/c__FlightAgent$AvailableFlight"
}
```

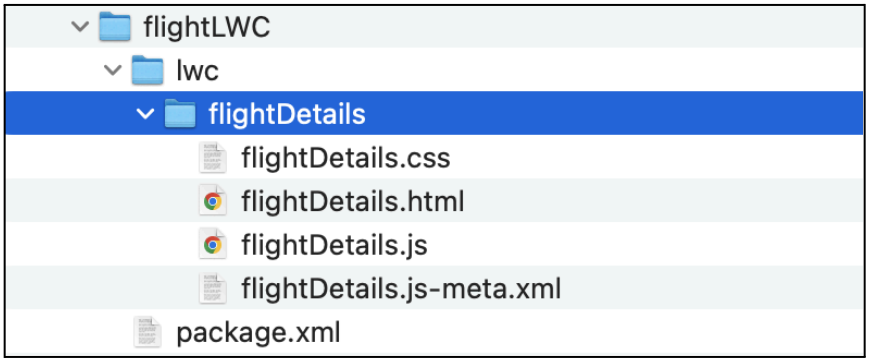
This sample code shows the contents of the `renderer.json` file.

```
{
  "renderer": {
    "componentOverrides": {
      "$": {
        "definition": "c/flightDetails"
      }
    }
  }
}
```

Build Output Components with Lightning Web Components

This section explains how the components are created and deployed for agent action output.

This image shows the Lightning Web Component (LWC) folder structure.



The LWC component includes HTML markup designed to represent the data that the agent returns for `@apexClassType/c__FlightAgent$AvailableFlight`. This HTML markup ensures that the data is displayed in an intuitive and customized format.

This sample code shows the contents of the `flightDetails.js-meta.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
   <apiVersion>63.0</apiVersion>
   <isExposed>true</isExposed>
   <masterLabel>Flight LWC</masterLabel>
   <targets>
       <target>lightning__AgentforceOutput</target>
   </targets>
</LightningComponentBundle>
```

Note: When you create an LWC component to override the UI for action input, use `lightning__AgentforceInput` as the target. For output, use

`lightning__AgentforceOutput`. For information about LWC target types, see [LWC Target Types for Agentforce Components](#).

This sample code shows the contents of the `flightDetails.html` file.

```
<template>
    <lightning-card icon-name="standard:flight" class="flight-card-container">
    <span class="flightTitle">AvailableFlights</span>
        <!-- Flight Cards List -->
        <div class="flight-list-container">
            <template for:each={flightData} for:item="flight">
                <div key={flight.flightId} class="flight-card">
                    <!-- Flight Info Section -->
                    <div class="flight-info">
                        <h2 class="flight-id">{flight.flightId}</h2>
                        <div class="discount-tag">{flight.discountPercentage}%
Off</div>
                    </div>

                    <!-- Flight Price, Duration, Departure and Arrival -->
                    <div class="price-duration">
                        <div class="price">
                            <strong>${flight.price}</strong>
                        </div>
                        <div class="duration">
                            {flight.durationInHr}
                        </div>
                    </div>

                    <!-- Timeline for Departure, Duration and Arrival -->
                    <div class="flight-timeline">
                        <div class="timeline">
                            <div class="time-point departure">
                                <span>Departure Time: {flight.departureTime}</span>
                            </div>
                            <div class="time-point arrival">
                                <span>Arrival Time: {flight.arrivalInHr}</span>
                            </div>
                        </div>
                    </div>

                    <!-- Additional Info Section (Layovers, Pets, etc.) -->
                    <div class="additional-info">
                        <div class="layovers">
                            <lightning-icon icon-name="utility:loop"
size="small"></lightning-icon>
                            <span>Layovers: {flight.numLayovers}</span>
                        </div>
```

```
                        <div class="pets">
                            <lightning-icon icon-name="utility:paw"
size="small"></lightning-icon>
                            <span>Pets Allowed: {flight.petAllowedStatus}</span>
                        </div>
                    </div>

                    <!-- Book Now Button -->
                    <div class="card-footer">
                        <lightning-button variant="brand" label="Book Now"
onclick={handleBookFlight} data-flight-id={flight.flightId}></lightning-button>
                    </div>
                </div>
            </template>
        </div>
    </lightning-card>
</template>
```

This sample code shows the contents of the `flightDetails.js` file.

```
import { LightningElement, api } from 'lwc';

export default class FlightDetails extends LightningElement {

   flightData = [];

    @api
    get value() {
        return this._value;
    }
    /**
     * @param  {} value
     */
    set value(value) {
        this._value = value;
    }

    // Method to convert duration from minutes to hours and minutes
    formattedDuration(durationInMin) {
        if (durationInMin) {
            const hours = Math.floor(durationInMin / 60); // Get whole hours
            const minutes = durationInMin % 60; // Get remaining minutes
            return `${hours} hr ${minutes} min`
        }
        return;
    }
```

```
    // Method to calculate arrival time based on departure time and duration
    arrivalTime(departureTime, durationInMin) {
        const [hours, minutes] = departureTime.split(':').map(num => parseInt(num));
        const departureDate = new Date(2025, 0, 1, hours, minutes); // Sample date
for calculation

        const arrivalDate = new Date(departureDate.getTime() + durationInMin *
60000); // Add duration to departure time

        const arrivalHours = String(arrivalDate.getHours()).padStart(2, '0');
        const arrivalMinutes = String(arrivalDate.getMinutes()).padStart(2, '0');

        return `${arrivalHours}:${arrivalMinutes}`;
    }

    connectedCallback() {
        if (this.value) {
           this.updatedValue = []
           this.value.flights.map((flight) => {
            this.updatedValue.push({...flight,
arrivalInHr:this.arrivalTime(flight.departureTime, flight.durationInMin),
                petAllowedStatus:this.value.isPetAllowed ? 'Yes' : 'No',
                durationInHr:this.formattedDuration(flight.durationInMin)
           })
           });
           // this.value.updatedFlights = this.updatedValue;
          this.flightData = this.updatedValue;
        }
    }

}
```

See Also

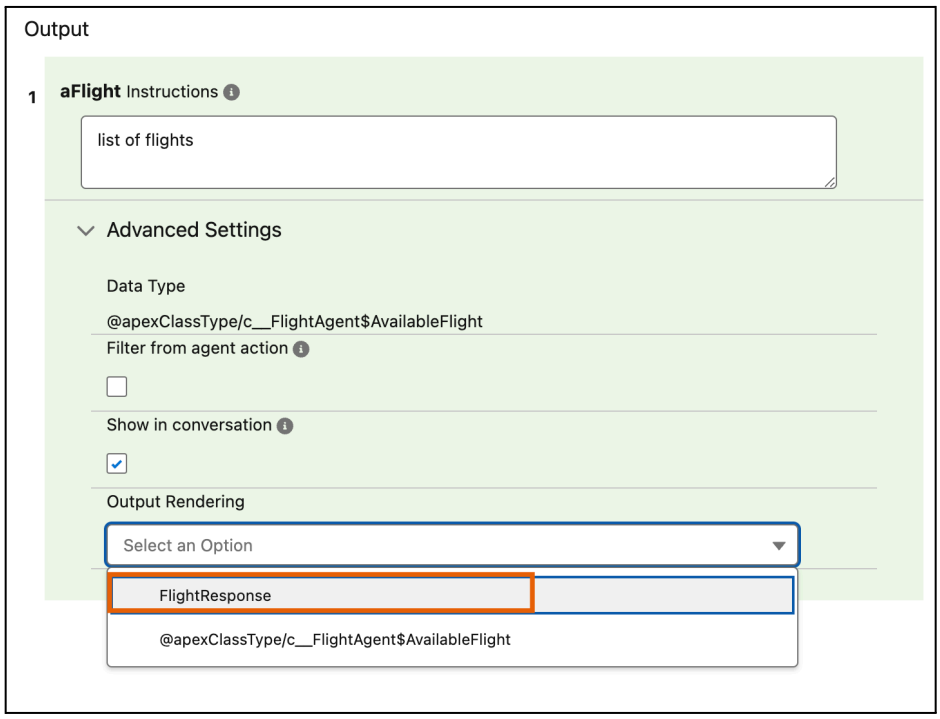
- *Lightning Web Components Developer Guide*: Get Started with Lightning Web Components

Integrate Custom Lightning Type into Agent Action Output

To add a custom Lightning type to the agent action, complete these steps.

1. Open the agent action.
2. Edit the `Output Rendering` parameter of the agent action output for `aFlight`.
3. Select the custom lightning type `flightResponse`.
4. Save the agent action.

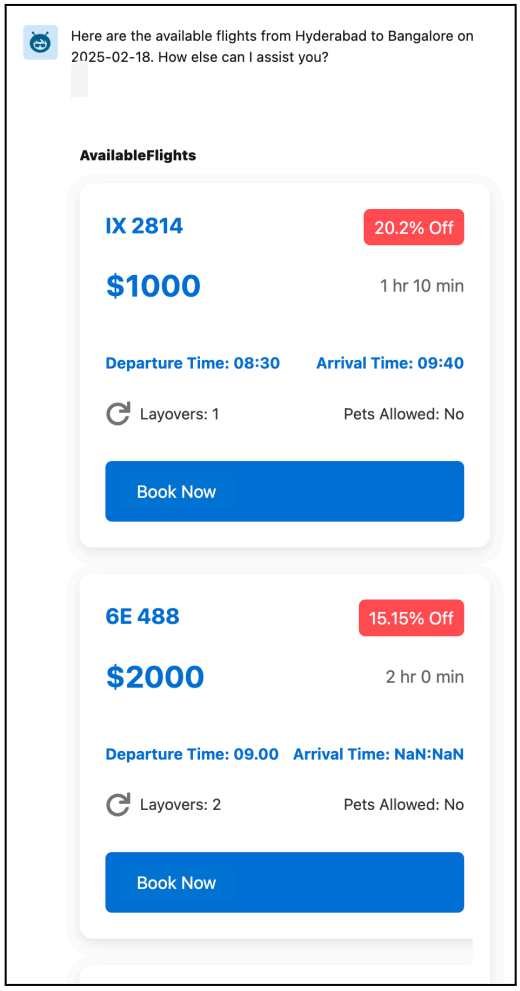
This image shows the custom Lightning type that you created.



Customized Output UI

Before executing the agent action that you modified, reload the agent page. The agent prompts you to provide input and then generate the output. The output provides a new UI experience.

This image shows how the custom agent action's output appears in an agent conversation.



Customize UI for Input

Create a custom Lightning type named `flightFilter` to show filters in the input UI that suits your business needs.

Override Default UI for Input with Custom Lightning Types

Override the agent's action UI for input to enhance the user experience by using Custom Lightning Types (CLTs). With CLTs, you can add your own Lightning Web Components (LWC) to present data in a more structured and intuitive format

Configure the editor.json file to override the default UI of a custom Lightning type in the agent action.

Here's an example that shows a lightningTypes folder for a custom Lightning type named `flightFilter`.

```
+--lightningTypes
        +--flightFilter
            +--schema.json
            +--lightningDesktopGenAi
               +--editor.json
```

The custom Lightning type `flightFilter` includes a `schema.json` file and an `editor.json` file. The `editor.json` file controls how the data is displayed to the user in the agent action input.

This sample code shows the contents of the `schema.json` file.

```
{
  "title": "Flight Filter",
  "description": "Flight Filter",
  "lightning:type": "@apexClassType/c__FlightAgent$Filter"
}
```

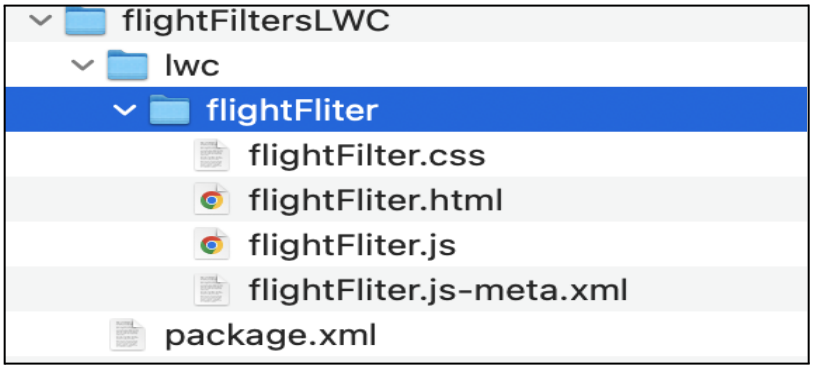
This sample code shows the contents of the `editor.json` file.

```
{
  "editor": {
    "componentOverrides": {
      "$": {
        "definition": "c/flightFilter"
      }
    }
  }
}
```

Build Input Components with Lightning Web Components

This section explains how the components are created and deployed for agent action input.

This image shows the Lightning Web Component (LWC) folder structure.



The LWC component includes HTML markup designed to accept input for `@apexClassType/c__FlightAgent$Filter`. This HTML markup ensures that the data is displayed in an intuitive and customized format.

This sample code shows the contents of the `flightFilter.js-meta.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
   <apiVersion>63.0</apiVersion>
   <isExposed>true</isExposed>
   <masterLabel>Flight Request Filters</masterLabel>
   <targets>
       <target>lightning__AgentforceInput</target>
   </targets>
</LightningComponentBundle>
```

Note: When you create an LWC component to override the UI for action input, use `lightning__AgentforceInput` as the target. For output, use `lightning__AgentforceOutput`. For information about LWC target types, see [LWC Target Types for Agentforce Components](#).

This sample code shows the contents of the `flightFilter.html` file.

```
<template>
   <lightning-card title="Price and Discount Percentage">
       <div class="slds-p-horizontal_medium">
           <!-- Price input -->
           <lightning-input
               label="Enter Price (between 1,000 and 20,000)"
               name="price"
               value={price}
```

```
                type="number"
                min="1000"
                max="20000"
                step="1"
                onchange={handleInputChange}
                read-only={readOnly}>
            </lightning-input>
            <!-- Discount Percentage input -->
            <lightning-input
                label="Enter Discount Percentage (0% to 100%)"
                name="discountPercentage"
                value={discountPercentage}
                type="number"
                min="0"
                max="100"
                step="1"
                onchange={handleInputChange}
                read-only={readOnly}>
            </lightning-input>
        </div>
    </lightning-card>
</template>
```

This sample code shows the contents of the `flightFilter.js` file.

```
import { api, LightningElement } from 'lwc';
export default class FlightFilter extends LightningElement {
  @api
   get readOnly() { return this._readOnly; }

   set readOnly(value) { this._readOnly = value;}
   _readOnly = false;
   _value;
   @api
   get value() { return this._value;}
   set value(value) { this._value = value;}
   price; discountPercentage;

   connectedCallback() {
       if (this.value) {
           this.price = this.value?.price || '';
           this.discountPercentage = this.value?.discountPercentage || '';
       }
   }
   handleInputChange(event) {
       event.stopPropagation();
       const { name, value } = event.target;
```

```
        this[name] = value;
        this.dispatchEvent(new CustomEvent('valuechange', {
          detail: {
            value: {
              price: this.price,
              discountPercentage: this.discountPercentage
            }
          }
        }));
    }
}
```

Note: You must include the `handleInputChange()` function to capture user input, update the component's state, and notify the parent component (planner component) by using the `valuechange` event. The function ensures real-time data binding and prevents unwanted event propagation.

See Also

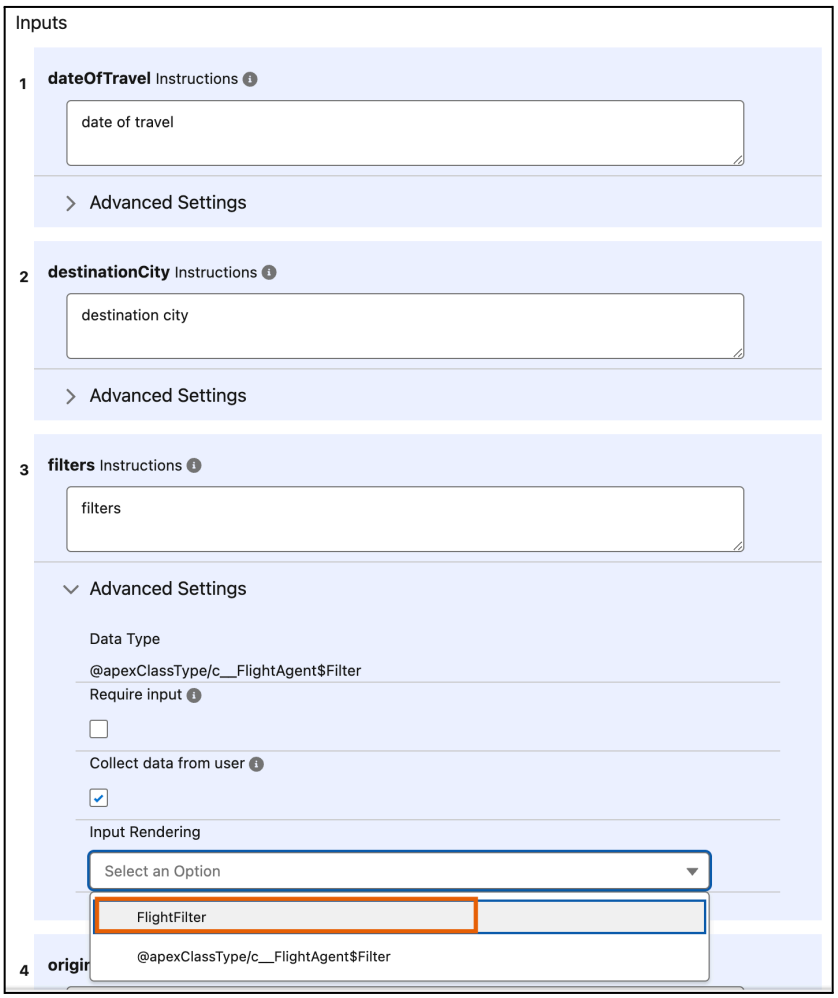
- *Lightning Web Components Developer Guide*: Get Started with Lightning Web Components

Integrate Custom Lightning Type into Agent Action Input

To add a custom Lightning type to the agent action, complete these steps.

1. Open the agent action.
2. Edit the `Input Rendering` parameter of the agent action input for `filters`.
3. Select the custom lightning type `flightFilter`.
4. Save the agent action.

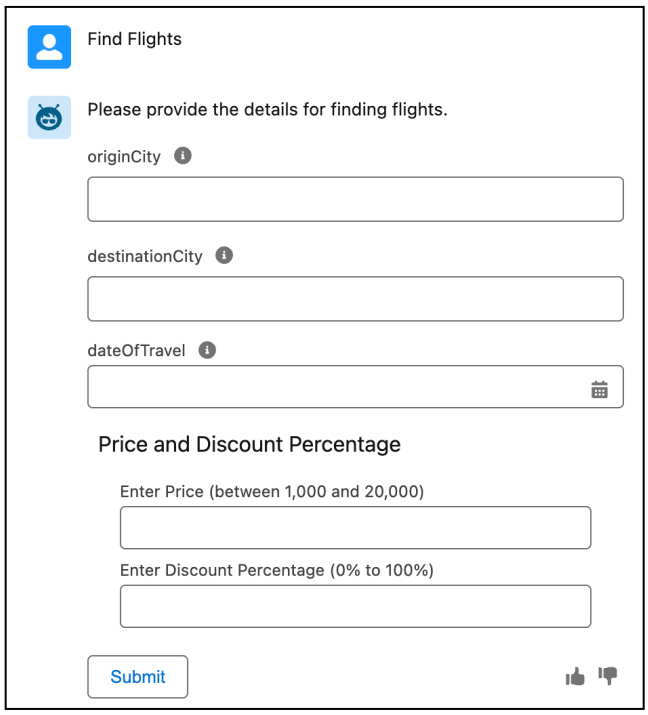
This image shows the custom Lightning type that you created.



Customized Input UI

Before executing the agent action that you modified, reload the agent page. The agent prompts you to provide input and then generate the output. The input provides a new UI experience.

This image shows how the custom agent action's input appears in an agent conversation.



Note: In certain instances the large language model (LLM) requests input as text, so make sure to accurately update the topic instructions for the correct selection of the Override Input component. For example, when you enter a prompt to find flights, the agent executes the Find Flight action. The Find Flight action executes by taking input through a UI form, and not in the form of Text because it includes a price and discount range.

Known Limitations

You can only override the default UI with a custom Lightning type for agent actions that use Apex classes as input or output.

Important Notices

We intend to introduce this changed behavior in an upcoming release.

The introduction of a few mandatory tags after the general availability (GA) of Lightning Types means that the definition of LWC components will change. As a result, components must incorporate these tags from GA onwards.

References

Lightning Types

This section describes the keywords available and the default editor and renderer associated with each of the Lightning types.

lightning__objectType

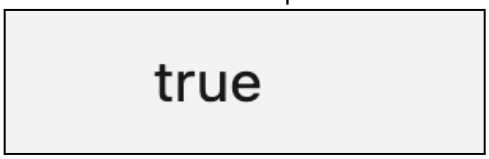
Use the `lightning__objectType` lightning type to create object lightning types. This complex Lightning type can contain sub-properties, each with its own Lightning type.

With `lightning__objectType`, you can group other lightning types. The `lightning__objectType` lightning type corresponds to the object type defined in a JSON schema.

For information about object types in JSON Schema, see [Understanding JSON Schema: object](#).

lightning__booleanType

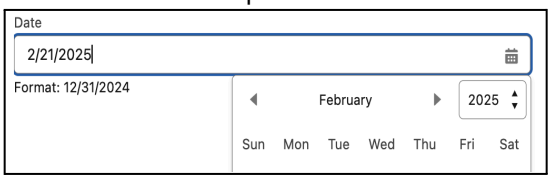
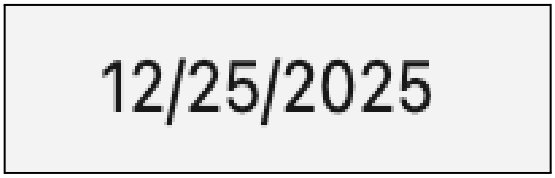
The `lightning__booleanType` Lightning type corresponds to the boolean type in a JSON schema. It produces only two values, true or false.

`lightning__booleanType`	
Editor Description	When the type receives user input, it appears as a toggle in the UI. Default editor example: 
Renderer Description	When the type produces output, it's rendered as a `true` or `false` value. Default Renderer example: 

For information about boolean type, see [Understanding JSON Schema: boolean](#).

lightning__dateType

The `lightning__dateType` Lightning type uses a string type to specify the date data in the format `yyyy-mm-dd`.

`lightning__dateType`	
Editor Description	When the type receives user input, it appears as a date picker in the UI. Default editor example: 
Renderer Description	When the type produces output, it's rendered as a `date` data. Default Renderer example: 

lightning__dateTimeType

The `lightning__dateTimeType` Lightning type describes the complex Lightning type `lightning__objectType`, which contains the `dateTime` and `timeZone` (optional) properties. Use the `Lightning__dateTimeType` Lightning type to specify date and time together.

Because `lightning__dateTimeType` is a standard complex Lightning type, the value of the type is represented as an object.

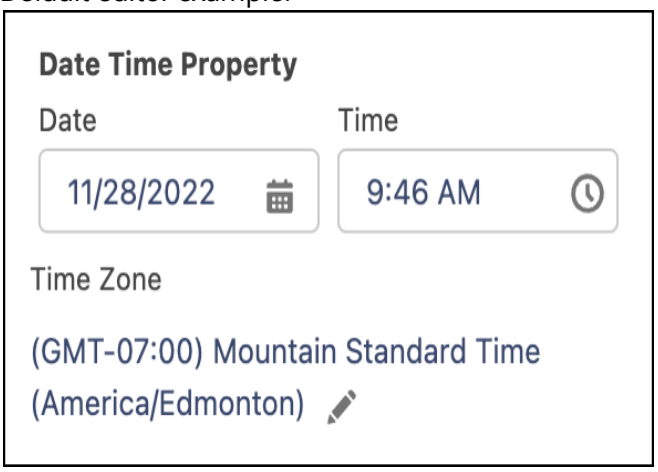
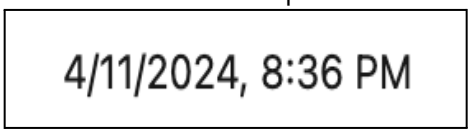
This table shows the properties of the object type that the `lightning_dataTimeType` Lightning type describes.

Property	Required or Optional	Type	Description
dateTime	Required	String	Specify the date value in `yyyy-MM-dd'T'HH:mm:ss.SSSZ` format.

timeZone	Optional	String	Specify the time zone information in IANA time zone database format.

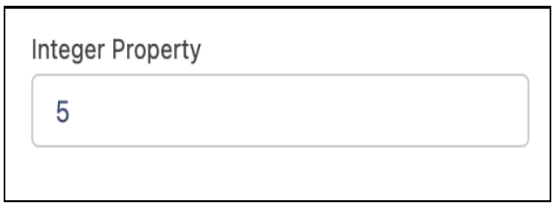
This example shows an object with valid date and time values.

```
{
"dateTime": "2012-05-31T01:30:05.000Z",
"timeZone": "Asia/Kolkata"
}
```

`lightning__dateTimeType`	
Editor Description	When the type receives user input, it appears as a date picker and a time picker in the UI. Default editor example: 
Renderer Description	When the type produces output, it's rendered as a `date` and `time` data. Default Renderer example: 

lightning__integerType

Use the `lightning__integerType` Lightning type to specify integers. The type applies to whole numbers. The `lightning__integerType` Lightning type corresponds to the integer type in a JSON schema.

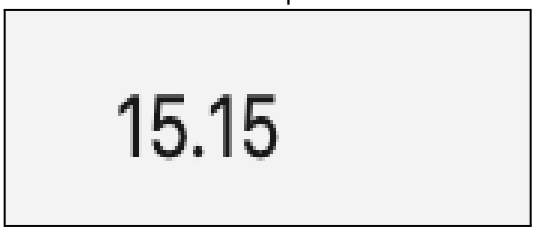
`lightning__intergerType`	
Editor Description	When the type receives user input, it appears as a numeric input field in the UI. Default editor example: 
Renderer Description	When the type produces output, it's rendered as a number display field. Default Renderer example: 

For information about the integer type in JSON Schema, see [Understanding JSON Schema: integer](#).

lightning__numberType

Use the `lightning__numberType` Lightning type to specify numbers. This type is validated as a decimal number, also known as a float in some programming languages. The `lightning__numberType` Lightning type corresponds to the number type in a JSON schema.

`lightning__numberType`	
Editor Description	When the type receives user input, it appears as a decimal number input field in the UI. Default editor example:

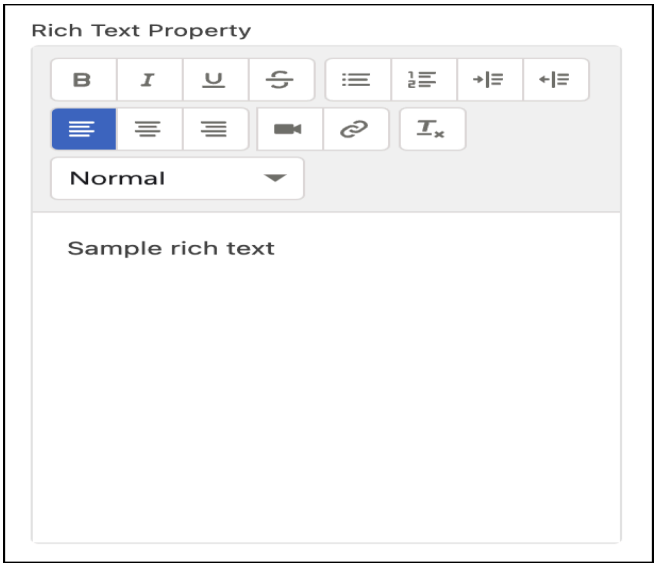
	
Renderer Description	When the type produces output, it's rendered as a decimal number display field. Default Renderer example: 

For information about number type in JSON Schema, see [Understanding JSON Schema: number](#).

lightning__richTextType

Use the `lightning__richTextType` Lightning type to add, edit, and delete rich text data. You can enter input text data of up to 100,000 characters.

`lightning__richTextType`	
Editor Description	When the type receives user input, it appears as a rich text input field in the UI. Default editor example:

	
Renderer Description	When the type produces output, it's rendered as a rich text display field.

For information about rich text editor, see [Rich Text Editor].

lightning__textType

Use the lightning__textType Lightning type for text fields, such as titles and descriptions. You can enter input text data of up to 255 characters. The `lightning__objectType` Lightning type corresponds to the string type in a JSON schema.

`lightning__textType`	
Editor Description	When the type receives user input, it appears as a text input field on the UI. Default editor example: 
Renderer Description	When the type produces output, it's rendered as a text display field. Default Renderer example:

	asda23s

lightning__multilineTextType

The `lightning__multilineTextType` Lightning type is similar to `Lightning__textType`, but it accommodates a larger maximum character length and an editor for larger text input. The `lightning__multilineTextType` Lightning type corresponds to the string type in a JSON schema.

`lightning__multilineTextType`	
Editor Description	When the type receives user input, it appears as a multiline text input field on the UI. Default editor example: Multiline Text Property some text
Renderer Description	When the type produces output, it's rendered as a multiline text display field.

For information about string types in JSON Schema, see [Understanding JSON Schema: string](#).

lightning__urlType

Use the `lightning__urlType` Lightning type for URL values. To specify the url schemes that the type can validate against, configure `lightning:allowedUrlSchemes` parameter of the Lightning type.

`lightning__urlType`	
Editor Description	When the type receives user input, it appears as a URL input field in the UI. Default editor example:

	URL Property https://sampleurl.com
Renderer Description	When the type produces output, it's rendered as a hyperlink.

LWC Target Types for Agentforce Components

This section describes the Agentforce targets.

lightning__AgentforceInput

Enables a component to be used in agent actions. Use this target to configure components that can accept input data from a user in any agent actions.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
   <apiVersion>63.0</apiVersion>
   <isExposed>true</isExposed>
   <targets>
       <target>lightning__AgentforceInput</target>
   </targets>
   <targetConfigs>
       <targetConfig targets="lightning__AgentforceInput">
           <property name="myString" type="String"></property>
           <property name="myBoolean" type="Boolean"></property>
       </targetConfig>
   </targetConfigs>
</LightningComponentBundle>
```

targetConfigs

Configures the component for different action types and defines component properties. For agent actions, only `lightning__AgentforceInput` is supported. The `targetConfigs` tag contains at least 1 `targetConfig` tag.

targetConfig

Configures a component for action input with this attribute.

Attribute	Description	Required
targets	Specify 1 or more action types in the `targets` attribute, such as `<targetConfig targets="lightning__AgentforceInput">` or `<targetConfig targets="lightning__AgentforceOutput,lightning__AgentforceOInput">`. The `targets` attribute value must match 1 or more of the action types that you listed under `<targets>`.	true

Property

Specifies a public property of a component. The component author defines the property in the component's JavaScript class by using the `@api` decorator. See the Usage section.
Use the `property` tag with these attributes.

Attribute	Type	Description	Required
name	String	The attribute name. This value must match the property name in the component's JavaScript class.	Yes
type	String	The attribute's data type. Make sure that this value matches the type assigned to the property in the component's JavaScript module. If the types don't match, the value in the configuration file takes precedence.	Yes

		Supported types are: • `Boolean` • `Integer` • `String`	

lightning__AgentforceOutput

Enables a component to be used in agent actions. Use this target to configure components that can display output data from an agent action.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
   <apiVersion>63.0</apiVersion>
   <isExposed>true</isExposed>
   <targets>
       <target>lightning__AgentforceOutput</target>
   </targets>
   <targetConfigs>
       <targetConfig targets="lightning__AgentforceOutput">
           <property name="myString" type="String"></property>
           <property name="myBoolean" type="Boolean"></property>
       </targetConfig>
   </targetConfigs>
</LightningComponentBundle>
```

targetConfigs

Configures the component for different action types and defines component properties. For agent actions, only `lightning__AgentforceOutput` is supported. The `targetConfigs` tag contains at least 1 `targetConfig` tag.

targetConfig

Configures a component for action output with this attribute.

Attribute	Description	Required
targets	Specify 1 or more action types in the `targets` attribute, such as `<targetConfig`	true

	targets="lightning__AgentforceOutput"> or <targetConfig targets="lightning__AgentforceOutput,lightning__AgentforceOInput">. The targets attribute value must match 1 or more of the action types that you listed under <targets>.	

Property

Specifies a public property of a component. The component author defines the property in the component's JavaScript class by using the `@api` decorator. For more information, see the Usage section. Use the `property` tag with these attributes.

Attribute	Type	Description	Required
name	String	The attribute name. This value must match the property name in the component's JavaScript class.	Yes
type	String	The attribute's data type. Make sure that this value matches the type assigned to the property in the component's JavaScript module. If the types don't match, the value in the configuration file takes precedence. Supported types are: • `Boolean` • `Integer` • `String`	Yes