Custom Property Types and Property Editors

Beta

Salesforce, Summer ’23, Version 58.0

@salesforcedocs

Last updated: May 30, 2023

Important: This feature is a Beta Service. Customer may opt to try such Beta Service in its sole discretion. Any use of the Beta Service is subject to the applicable Beta Services Terms provided at Agreements and Terms.

CONTENTS

Introduction

Add custom property editors and property types to your custom Lightning web components to make component configuration more intuitive in Experience Builder. Without custom property editors and types, property editing options are limited to a few field types, such as text boxes and dropdown menus. Custom property editors let you create color selectors, sliders, and more.

This guide introduces you to property types and property editors, and demonstrates how to use them in custom Lightning web components. First it outlines planning steps and design decisions and shows how they determine the steps you take to build a component. Next it offers step-by-step instructions, including a sample component, for creating a custom property type and property editor. Detailed sections discuss

- Creating an ExperiencePropertyTypeBundle and deploying it with the Metadata API
- Defining the JSON schema and design of a custom property type
- Creating a custom property editor, including HTML and CSS code samples and XML configuration
- Linking the custom property editor and property type to the component to test it

Background

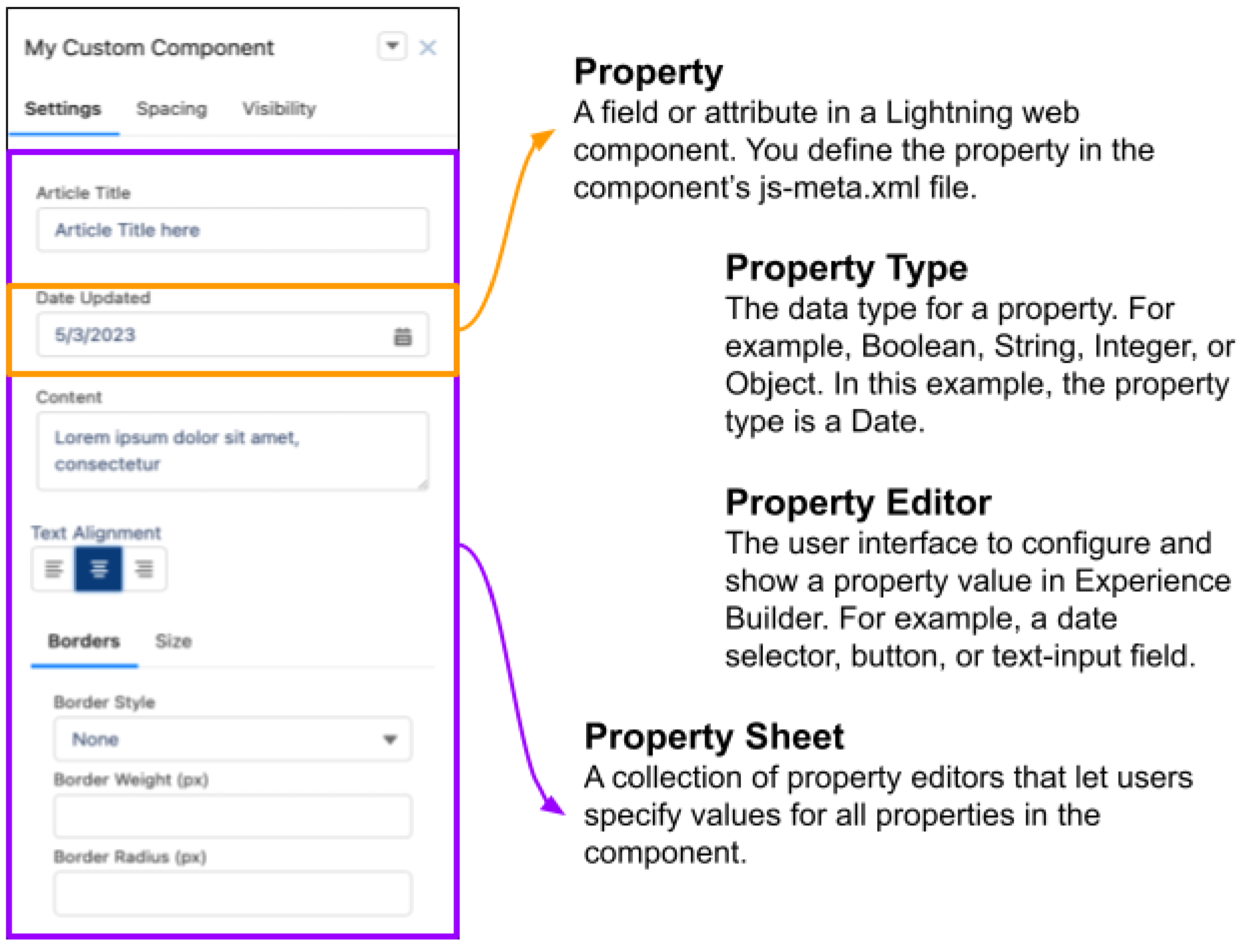
Let's start with definitions of some key terms: **property**, **property type**, **property editor**, and **property sheet**.

To an Experience Builder user, a **property** is a field in a component. Every Lightning web component has a set of exposed properties, or attributes, that site builders can configure in Experience Builder. When Experience Builder users assign a value for font family, color, size, margin, or any other field in a component property panel, they're configuring a property. (For more information about Lightning web components, see the Lightning Web Components Developer Guide.)

When you create a Lightning web component, you specify the property type, description, and title for every property in the component. The **property type** determines the kind of value that a property can hold. Another term for *property type* is *data type*. Salesforce supports five out-of-the-box property types for Lightning web components in Experience Builder: String, Integer, Boolean, ContentReference, and Color.

After you define a property's data type, you can specify its **property editor**–that is, the user experience for configuring the property. In Experience Builder components, users enter a property value in a property editor such as a picklist, checkbox, button, or text-input field.

Property editors are gathered into a **property sheet**–a collection of all the property editors in the property panel for a given Experience Builder component. The property panel is where Experience Builder users enter values for the component properties.



Article Title: Article Title here

Date Updated: 5/3/2023

Content: Lorem ipsum dolor sit amet, consectetur

Border Style: None

Border Weight (px)

Border Radius (px)

Property
A field or attribute in a Lightning web component. You define the property in the component's js-meta.xml file.

Property Type
The data type for a property. For example, Boolean, String, Integer, or Object. In this example, the property type is a Date.

Property Editor
The user interface to configure and show a property value in Experience Builder. For example, a date selector, button, or text-input field.

Property Sheet
A collection of property editors that let users specify values for all properties in the component.

Considerations and Limitations for Custom Property Types and Editors

Before you create a custom property type or editor, review these limitations and known issues for the beta release.

In all editions, you can create custom property types and property editors only for Lightning web components in Lightning Web Runtime (LWR) sites in Experience Builder. You can't create them for Aura sites or Aura components. (For more information about Experience Builder, see Customize Sites with Experience Builder in Salesforce Help. For more information about LWR sites, see the LWR Sites for Experience Cloud developer guide.)

When you include a custom property type or custom property editor in a Lightning web component, the component's js-meta.xml file has these limitations.

- You can use custom property types and property editors only in Lightning web components that target `lightningCommunity__Default`. You can't use them in components that target

`lightningCommunity__Page_Layout`, `lightningCommunity__Theme_Layout`, or other targets that are unrelated to Experience Builder. If you do, deployment of the component fails.
- If a property references a custom property type or lightning type (for example, `<property name="myProperty" type="c__customType" />`), you can't specify any of these attributes in the property. If you do, deployment fails.
    - datasource
    - max
    - min
    - placeholder
- If a property references a custom property editor, you can specify screenResponsive="true" for that property. However, in the component property panel, the screen-responsive icon doesn't appear next to that property's label.

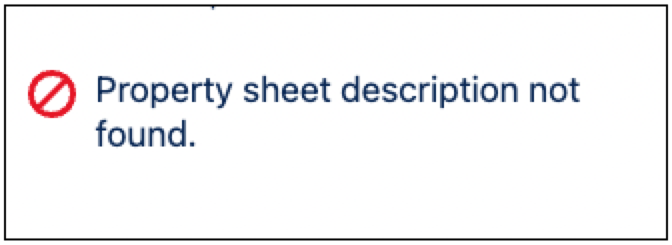
See [XML Configuration File Elements]() in the Lightning Web Components Developer Guide for more information on the js-meta.xml file.

The ExperiencePropertyTypeBundle metadata type has these limitations.

- You can't update an existing custom property type. You can only create and delete one. This limitation also applies to updating a package that contains custom property types.
- Custom property types don't support the JSON schema array data type, so you can't deploy a custom property type of the data type array. (For more information, see [Understanding JSON Schema: array]()).
- You can't nest object properties in a custom property type schema. In a given schema, a custom property type of the data type object can't contain child properties of the data type object. Instead, create a custom property type for the child object property and reference it in the current schema using the `lightning:type` keyword. Here's an example of an invalid schema. For more information, see [The Schema.json File]().

```
{
 "title": "Invalid Nested ObjectType",
 "lightning:type": "lightning__objectType",
 "properties": {
   "postalCode": {
     "title": "Invalid Nested ObjectType Property",
     "lightning:type": "lightning__objectType",
     “properties”: {
        “propertyName”: { … }
     }
   }
 }
}
```

When a custom property editor or type is invalid, an error message appears instead in the component property panel in Experience Builder.


Property sheet description not found.

You can find more details on the error cause in the browser console.

We recommend that you don't save a property type that contains a circular reference. A circular reference occurs when property type A refers to property type B, and property type B refers back to property type A. You aren't prevented from saving a property type with a circular reference, but these references cause gacks when they render a property editor. You also can't delete circularly referenced property types. For example, you can't delete property type A when property type B refers to A, and you can't delete property type B when property type A refers to B.

Important Changes from the Developer Preview

Some components with custom property types or editors that you built in Spring '23 can be incompatible with the metadata in Summer '23. We recommend that you delete these custom components, property types, and editors and redeploy them so that the components work as intended. Otherwise, when you try to open an Experience Builder page with these custom components, the page doesn't render correctly, and an error window can appear. If the page fails to render, open the Page Structure panel in Experience Builder, find the custom component, and click the trash icon beside it. Then reload the page.

- Packaging is supported for the ExperiencePropertyTypeBundle metadata type, which describes property types, and for Lightning web components that reference a custom property type or property editor. Now you can easily distribute custom components that contain custom property types or editors to other users in your own or other companies.
  - Updates of managed packages containing custom property types are not supported. This includes deleting types. If you upgrade a package that contains a custom property type or editor, users who installed this package must uninstall it and reinstall the updated package. We recommend that you don't include ExperiencePropertyTypeBundles in non-beta managed packages.
- You can add the translatable attribute to properties with a custom property type whose underlying JSON schema type is string. You can translate the values in these properties into all your site languages. (See Supported Attributes for the Property Tag in the js-meta.xml File.)

- Custom labels are now supported for a custom property type's `title` and `description` fields. You can use custom labels to add translations of the title and description into all your site languages.
- Custom property types with an underlying JSON schema type of type string, integer, or number can specify the exposedTo and screenResponsive attributes for a property. Now you can configure values for a custom property that are tailored to different screen sizes.
- In a component with properties that reference custom property types, you can reference the Color and the ContentReference types. No need to create a selector from scratch for color or content references.
- In the Salesforce CLI, you can now use source commands to deploy an ExperiencePropertyTypeBundle. No more creating a .zip file and using mdapi commands in the CLI–you can use fewer steps and more convenient commands.
- ExperienceBundle and DigitalExperienceBundle deployments validate property values for custom property types. When a site builder enters an invalid value for a custom property type, a message flags the error.
- The syntax to reference a custom property editor has changed. In Spring '23, you specified editor = "c__myCustomEditor". In Summer '23, you specify editor = "c/myCustomEditor".
- The syntax to define a property renderer override for an entire property is updated. Now it's clearer to developers which lines in design.json files are property editor overrides. (See Full Editor Override)

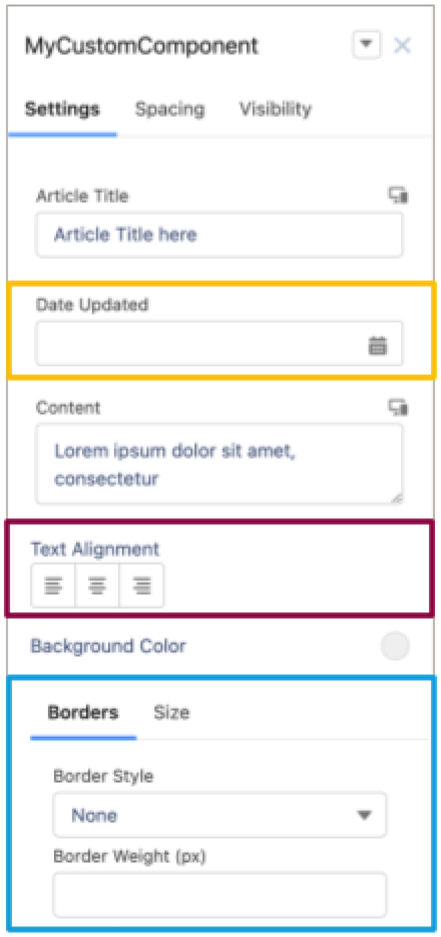
See Also
Create an ExperiencePropertyTypeBundle
Lightning Web Components Developer Guide: XML Configuration File Elements
Metadata API Developer Guide: ExperienceBundle
Metadata API Developer Guide: DigitalExperienceBundle (Beta)

Overview: Creating Custom Property Types and Editors

Let's consider some data and design decisions to make before you create a custom Lightning web component. Then we can go through an overview of the steps to create custom property types and editors.

Before you start to build the component, consider the data validation and user experience needs for the properties in the component. Those needs can establish which custom property types and property editors to create, and whether you can use an out-of-the-box solution instead of creating a custom one.

Let's say that you want your Lightning web component, called MyCustomComponent, to include properties for title, date updated, content, text alignment, background color, and layout borders and size. Here's how you want the component to look.



Our desired component

Property panel for the component

1. **articleDate**
   Has a custom property type with validation to ensure that users enter a date
2. **textAlignment**
   Has a custom property editor for text alignment
3. **layoutProperties**
   Has a custom property type with multiple properties and a tab layout

This code sample shows a custom component that's configured for Experience Builder with these properties: `articleDate`, `textAlignment`, `layoutProperties`.

```
class MyCustomComponent extends Lightning Element {

   @api articleDate;
   @api textAlignment;
   @api layoutProperties;
   ...
}
```

Here's an overview showing how you can develop this custom Lightning web component for Experience Builder.

Decide what kind of validation you want for each property in the component.

Usually when you create a custom Lightning web component, you include validations to verify that the data values that users enter into each component property are acceptable. For example, an address property doesn't work with integer values, so you want to validate that an integer value isn't entered in an address field.

Consider each property's data type and how to validate it, and whether you need to create a custom property type for that validation. (For a list of out-of-the-box property types available for use in a custom property type, and descriptions of their validations, see Lightning Property Types.)

In this example, these are the validations that you want for the properties in MyCustomComponent.

- **articleDate –**You want to ensure that users enter a date. Looking at the list of Lightning property types, you see that lightning_dateType provides the validation that you need.

- **textAlignment**– You don't need any special data validations for this property, so you don't need a custom property type. You can just specify `<property type = "String">` in MyCustomComponent's js-meta.xml file.

- **layoutProperties–**You want to ensure that borders always include border style, weight, and radius. On the list of Lightning property types, you find lightning__objectType, which you can use to create a custom property type with multiple properties.

In summary, the validations you want require you to use a lightning property type for articleDate, and create a custom property type for layoutProperties.

Decide what kind of property editors to give your Experience Builder users to configure each property in the component.

Think about the user experience for entering values into the properties of MyCustomComponent in Experience Builder. For example, for a text alignment property, do you want the property editor to be a text input field, dropdown menu, or icon group? Look at the default property editor that comes with the property type for each property in your custom component, and decide if that editor offers the user experience that you want. If it doesn't, you must create a custom property editor. (For a list of out-of-the-box property types, including their default property editors, see Lightning Property Types.)

In this example, these are the property editors that you want for the properties in MyCustomComponent.

- **articleDate**–The default property editor for lightning_dateType is a date picker. That's the user experience you want, so you don't need to create a custom property editor.
- **textAlignment**–The default property editor for this property is a combobox. You want an icon group to visually display the alignment options possible. You need to create a custom property editor.
- **layoutProperties**–You want to arrange the layout properties on separate tabs, but the default property editor for lightning__objectType is a vertical layout. You need to create a custom property editor.

In summary, you want a date picker, an alignment icon group, and a border combobox selector for MyCustomComponent. The color selector and search-selection fields require you to create custom property editors. (For more information, see Create a Custom Property Editor.)

Create the necessary custom property types and property editors.

After you decide which validations and property editors you want for the properties in your component, it's time to start building. This process requires you to create a new metadata type, called `ExperiencePropertyTypeBundle`, and new Lightning web components for the property editors.

To recap, these are the custom elements that you want for MyCustomComponent.

- **articleDate**, which requires a lightning type
- **textAlignment**, which requires a custom property editor
- **layoutProperties**, which requires a custom property type and custom property editor

High-Level Guide to Creating a Custom Property Type and Editor

Here's a step-by-step overview of how you can build the component. To follow along, download the sample component, which are shared throughout this documentation.

1. Create a custom property editor.
   a. Create a Lightning web component to act as the property editor. (See Create a Custom Property Editor.)
   b. To show the custom property editor in MyCustomComponent, reference it by editor attribute in the MyCustomComponent js-meta.xml file. (See Reference a Custom Property Editor by Editor Attribute.)

2. Create a custom property type.
   a. Create an `ExperiencePropertyTypeBundle` folder and define the schema.json file for the property in the folder. (See Structure of the ExperiencePropertyTypeBundle and The Schema.json File.)
   b. Deploy the `ExperiencePropertyTypeBundle` to your org. (See Use SFDX or Metadata API to Deploy ExperiencePropertyTypeBundles.)
   c. To show the custom property type in MyCustomComponent, reference it by type attribute in the MyCustomComponent js-meta.xml file. (See Reference a Custom Property Type by Type Attribute.)

3. Create a custom property type that includes a custom property editor.
   a. Create the custom property type.
   b. Create the custom property editor.

c. Specify the custom property editor for the custom property type.
   i. Create a design.json file. (See The Design.json File.)
   ii. Reference the custom property editor in the `propertyRenderer` object in the design.json file. (See Property Renderers.)

d. Deploy the `ExperiencePropertyTypeBundle`.

e. To show the custom property type in MyCustomComponent, reference it by type attribute in the MyCustomComponent js-meta.xml file.

Now that you have an idea of the process, let's explore the technical details. We introduce the custom property type and its relationship to JSON schema, how to create a custom property editor, and how to link everything together, with examples, throughout the implementation.

Create an ExperiencePropertyTypeBundle

The `ExperiencePropertyTypeBundle` is a new metadata type that describes property types. `ExperiencePropertyTypeBundle` components are available in API version 58.0 and later.

For a list of the `ExperiencePropertyTypeBundles` deployed in your org, use the Connect API resource `connect/experience-model/property-types`. (See Experience Model Resources in the Connect REST API Developer Guide.)

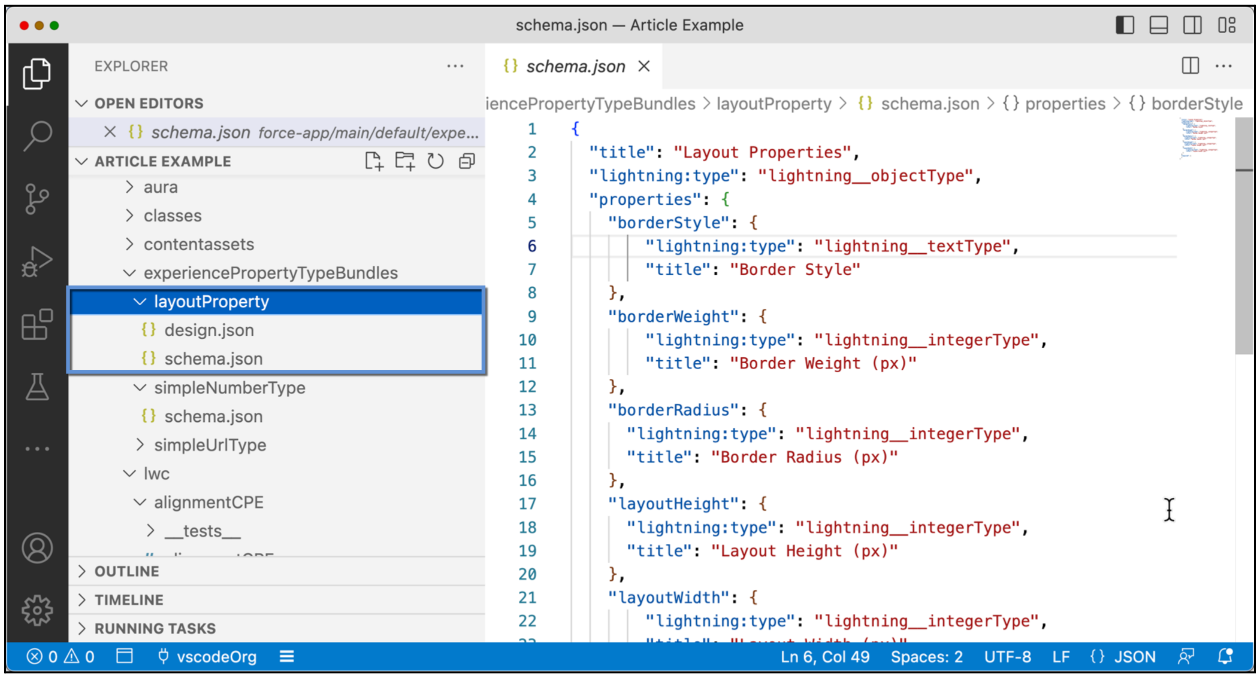
Structure of the ExperiencePropertyTypeBundle

Let's look at how an `ExperiencePropertyTypeBundle` folder is structured. Here's an example that shows an `experiencePropertyTypeBundles` folder for a custom property type, `layoutProperty`.

```
+--myMetadataPackage
     +--experiencePropertyTypeBundles (1)
          +--layoutProperty (2)
              +--schema.json (3)
              +--design.json (4)
```

- The `experiencePropertyTypeBundles` folder (1) contains a folder for each custom property type.
- Each custom property type in the bundle folder has a `propertyTypeName` folder that's named for the property type. In this example, the custom property type is `layoutProperty` (2).
- Each `propertyTypeName` folder contains a JSON file or files that define the property type.
    - A schema.json file, which is a JSON schema that drives the property type validation (3)
    - An optional design.json file, which provides the user experience and property editor information for that property type (4)

Here's how this experiencePropertyTypeBundles folder looks in practice.


schema.json — Article Example
EXPLORER
OPEN EDITORS
schema.json force-app/main/default/expe...
ARTICLE EXAMPLE
aura
classes
contentassets
experiencePropertyTypeBundles
layoutProperty
design.json
schema.json
simpleNumberType
schema.json
simpleUrlType
lwc
alignmentCPE
__tests__
OUTLINE
TIMELINE
RUNNING TASKS
schema.json
iencePropertyTypeBundles > layoutProperty > schema.json > properties > borderStyle
0 0 vscodeOrg
Ln 6, Col 49 Spaces: 2 UTF-8 LF JSON

```
{
  "title": "Layout Properties",
  "lightning:type": "lightning__objectType",
  "properties": {
    "borderStyle": {
        "lightning:type": "lightning__textType",
        "title": "Border Style"
    },
    "borderWeight": {
        "lightning:type": "lightning__integerType",
        "title": "Border Weight (px)"
    },
    "borderRadius": {
      "lightning:type": "lightning__integerType",
      "title": "Border Radius (px)"
    },
    "layoutHeight": {
      "lightning:type": "lightning__integerType",
      "title": "Layout Height (px)"
    },
    "layoutWidth": {
        "lightning:type": "lightning__integerType",
```

See Also

Connect REST API Developer Guide: Experience Model Resources

Understanding JSON Schema

The Schema.json File

The Design.json File

Use SFDX or Metadata API to Deploy ExperiencePropertyTypeBundles

To deploy or delete an `ExperiencePropertyTypeBundle` to your org, use the Metadata API or SFDX.

In Metadata API, a manifest file defines the metadata that you want to deploy. Here's a package.xml manifest file for an `ExperiencePropertyTypeBundle` that contains one custom property type, called `layoutProperty`.

```
<xml version="1.0" encoding="UTF-8">
<package xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>layoutProperty</members>
        <name>ExperiencePropertyTypeBundle</name>
    </types>
    <version>58.0</version>
</package>
```

To delete a custom property type, 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

SFDX and Packaging Support

`ExperiencePropertyTypeBundle` is supported by SFDX commands for retrieve, deploy, and tracking source control of this metadata. For a reference on SFDX commands, see source Commands in the Salesforce CLI Command Reference developer guide.

You can include ExperiencePropertyTypeBundles in first- and second-generation packages. For more information, see Use Managed Packages to Develop Your AppExchange Solution.

See Also
Salesforce CLI Command Reference
Considerations and Limitations for Custom Property Types and Editors
Use Managed Packages to Develop Your AppExchange Solution

The Schema.json File

For custom property types, Salesforce uses JSON Schema to define the data type and data structure of Lightning web component properties. We use JSON Schema so we can ensure that the structure of the property values that you set for the component is valid. For example, a complex component property can be expressed in different ways: It can be a string that describes all the properties in a list. It can be an object with each property, such as borders and layout size, separated into respective sub-properties. The schema lets you expect a specific data structure and type for that property and build around that information.

The schema.json file follows the JSON Schema specification to define the custom property type. The schema is composed of a set of specific keywords. Each keyword carries meaning to apply constraints to the structure of the data.

These are the keywords that you can specify in a schema.json file. Unless otherwise noted, the keywords follow the JSON Schema specification.

Keyword	Required?	Type	Description
title	Yes	String	Text used as the display name for the property
description	No	String	Explanatory text about the property type

const	No	JSON	Specifies a constant value that's supported for a property. Equivalent to setting a default value with `readOnly` set to `true`
enum	No	Array	Specifies a list of values that are supported for a property
lightning:type	Yes	String	Refers to other property types by fully qualified name. This keyword is specific to `ExperiencePropertyTypeBundle` but is syntactic sugar for the `$ref` keyword in JSON Schema, which links together schemas. (See [Understanding JSON Schema: The $ref keyword](#).)

The `lightning:type` property can reference only standard Lightning property types and the custom property types that you create. As in JSON Schema, each Lightning property type has additional type-specific keywords that apply only to that type. For example, `lightning__objectType` requires the keyword `properties` to specify the object's sub-properties, each with its own type.

A property type that contains other property types is a **complex type**. Only object and array types are considered complex. **Primitive types** don't contain other property types.

Example: Let's say that you want to create an `ExperiencePropertyTypeBundle` for `Property`, a complex type that includes sub-properties for `borderStyle`, `borderWeight`, `borderRadius`, `layoutHeight`, and `layoutWidth`. Each of the sub-properties is a primitive type. Here's what the json.schema file looks like.

```
{
 "title": "Layout Properties",
 "lightning:type": "lightning__objectType",
 "properties": {
   "borderStyle": {
       "lightning:type": "lightning__textType",
       "title": "Border Style"
   },
   "borderWeight": {
       "lightning:type": "lightning__integerType",
       "title": "Border Weight (px)"
   },
   "borderRadius": {
     "lightning:type": "lightning__integerType",
     "title": "Border Radius (px)"
   },
   "layoutHeight": {
     "lightning:type": "lightning__integerType",
     "title": "Layout Height (px)"
   },
   "layoutWidth": {
       "lightning:type": "lightning__integerType",
       "title": "Layout Width (px)"
   }
 },
 "required": []
}
```

See Also
Understanding JSON Schema: What is a schema?
Understanding JSON Schema: Generic keywords
Lightning Property Types
Create an ExperiencePropertyTypeBundle

Include Custom Labels

You can use custom labels for the title and description values for a custom property type. With custom labels, you can add translations of the title and description into all your site languages. Experience Builder users can see the title and description labels in their chosen language.

Follow these steps to create and use a translated custom label in a custom property type:

1. In Setup, enter `Labels` in the Quick Find box, and select **Custom Labels**.
2. Click **New Custom Label** and create the label.
3. In the schema.json file for the custom property type, use the expression {!$Label.c.MyLabel1} for the title and description values. "c" is the namespace prefix, and "MyLabel1" is the API name of the custom label.

See Also
Translate Custom Labels

The Design.json File

Design.json is an optional file where you can specify the layout of your property type–for example, as an accordion section or on tabs. You can also specify an editor override–a property editor that overrides the default editor for any `lightning:type` that the current property type references.

The JSON **design specification** encompasses the keywords that make up the design JSON file. The root of the JSON must have the `propertySheet` key, a JSON object that represents the design configuration for the property type. The `propertySheet` key determines the representation of the property in the Experience Builder user interface.

The propertySheet JSON object contains these keys.

Keyword	Required?	Type	Description
propertyRenderers	No	PropertyRenderers	Defines the editor overrides for the property type's schema. (See Property Renderers)
view	No	View	Defines the visual representation for the property type via an abstract component tree (See The View Specification)

The View Specification

The view is a specification for a standard way to represent the metadata of pages in Salesforce. It represents a page, or a fragment of a page, as an abstract tree of components expressed in a JSON object. For design.json files, view represents the visual representation for editing that property. For primitive property types, such as string, view could define a single property editor.

For a complex property type, such as an object, you can use view to define the layout of this property type's sub-properties. The sub-properties of a complex type are laid out in a property sheet, a collection of editors for each sub-property. Let's say that you have a complex type that contains three properties, `borderStyle`, `borderWeight`, `borderRadius`. Here are some examples of what you can do using view.

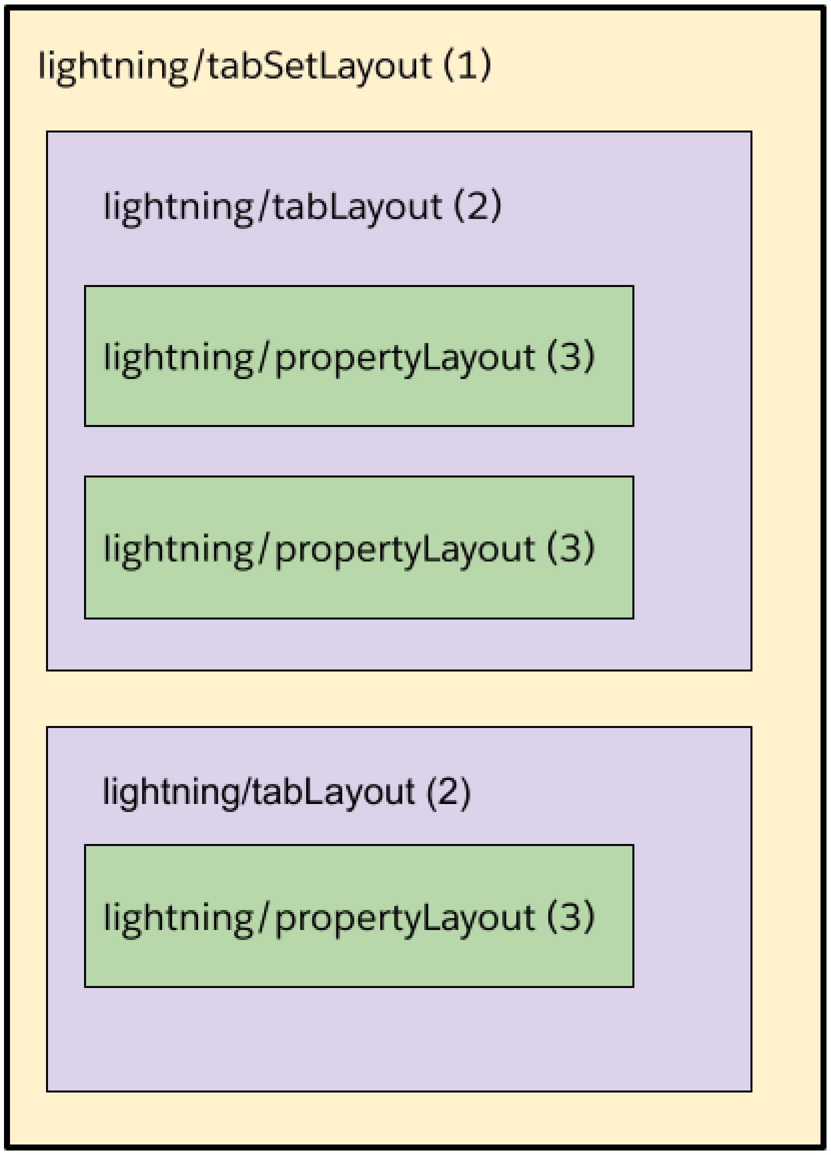
- Specify the order in which the property editors appear in the property sheet (and, therefore, in the component property panel in Experience Builder). For example, you can specify that the property editor for `borderWeight` is the first one shown in the property sheet, then `borderStyle`, then `borderRadius`.
- Specify which properties of the object property type to show. For example, you can opt not to show the property editor for `borderStyle` in the property sheet. This can be useful when certain properties of the component that you're configuring aren't relevant in Experience Builder.
- Specify the layout of the complex type, such as whether to arrange the properties in an accordion section or on tabs.

A view JSON object contains these keys.

Keyword	**Required**	**Type**	**Description**
definition	Yes	String	Specifies the layout of a complex property type. Can reference the out-of-the-box definitions or the `fullyQualifiedName` of any Lightning web component
attributes	No	Object	The key-value pairs for the properties of the view component
children	No	View[]	A list of child views

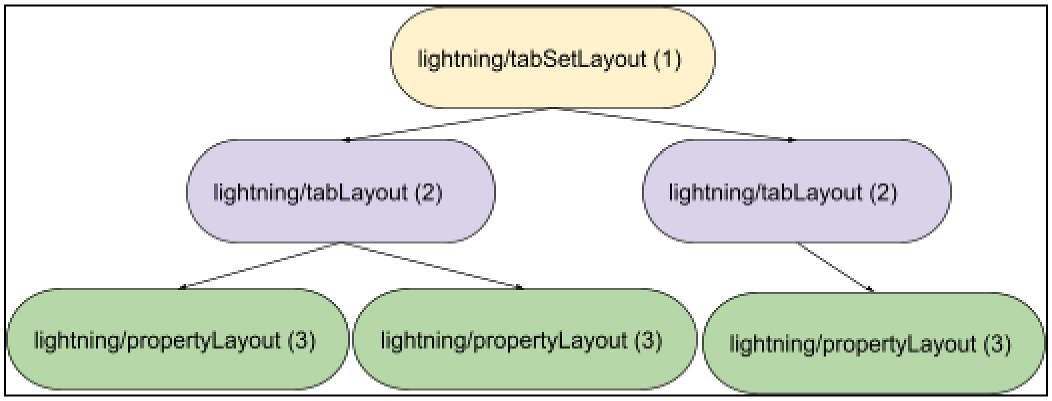
To help demonstrate the view, here's an example of a view layout for an `objectType` with three sub-properties, which are arranged on two tabs. We illustrate the same view in different formats.

This representation of the object layout is in the form of a user interface component, with each component and sub-component labeled with its definition.


lightning/tabSetLayout (1)
lightning/tabLayout (2)
lightning/propertyLayout (3)
lightning/propertyLayout (3)
lightning/tabLayout (2)
lightning/propertyLayout (3)

- The yellow section (1) represents the entire `tabset` component.
- The purple sections (2) represent each tab component.
- The green sections (3) represent each property layout component.

Here's the same view, represented as a tree. This is how the view will be represented in the DOM. The numbers and text colors correspond to the same ones described in the preceding object layout format.


lightning/tabSetLayout (1)
lightning/tabLayout (2)
lightning/tabLayout (2)
lightning/propertyLayout (3)
lightning/propertyLayout (3)
lightning/propertyLayout (3)

Finally, here's the same view represented as a JSON object, the format that we use to define a view for property sheets. The text colors correspond to the same colors in the preceding tree and user interface formats.

```
{
 "definition": "lightning/tabsetLayout",
 "children": [
   {
     "definition": "lightning/tabLayout",
     “attributes”: {
        “label”: “First Tab”
     },
     “children”: [
       {
         "definition": "lightning/propertyLayout",
         "attributes": {
           "property": "aProperty"
         }
       },
       {
         "definition": "lightning/propertyLayout",
         "attributes": {
           "property": "bProperty"
         }
       },
    ]
   },
     {
     "definition": "lightning/tabLayout",
     “attributes”: {
        “label”: “Second Tab”
     },
     “children”: [
       {
         "definition": "lightning/propertyLayout",
         "attributes": {
           "property": "cProperty"
         }
       },
     ]
   },
 ]
}
```

Out-of-the-Box Definitions for Views

When you use a view to specify the layout for a complex property type, you can leverage a few out-of-the-box definitions of layout components. Each layout component can contain a `lightning/propertyLayout` to specify where the property editor is expected to appear within the property sheet.

These are the out-of-the-box definitions.

- `lightning/propertyLayout`
- `lightning/verticalLayout`
- `lightning/tabSetLayout`
- `lightning/accordionLayout`

lightning/propertyLayout

The `lightning/propertyLayout` is a component that renders a property editor. Its definition takes a required attribute called “property,” the value of which is a reference to a sub-property in the schema. The property editor that’s rendered is based on the `lightning:type` that it references and whether the property is overridden by the `propertyRenderers`. (See The Editor Resolution Algorithm for more details on how the property editor is chosen. See the other layouts listed here for examples of how `propertyLayout` is used in accordance with other layouts.)

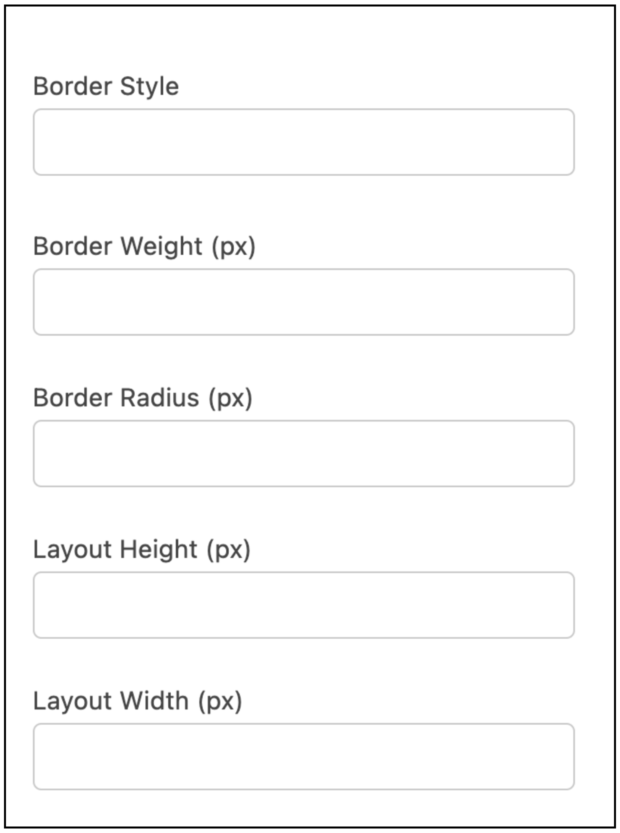
lightning/verticalLayout

To arrange your property sheet in a flat list, use the `lightning/verticalLayout` component. Properties appear in the order in which they’re defined in the schema. This layout can take only `lightning/propertyLayouts` as children.

Here’s a code snippet of a vertical layout view with properties for `borderStyle`, `borderWeight`, `borderRadius`, `layoutHeight`, and `layoutWidth` arranged in vertical order.

```
{
 "propertySheet": {
   "view": {
     "definition": "lightning/verticalLayout",
     "children": [
       {
         "definition": "lightning/propertyLayout",
         "attributes": {
           "property": "borderStyle"
         }
       },
       {
         "definition": "lightning/propertyLayout",
         "attributes": {
           "property": "borderWeight"
         }
       },
       {
         "definition": "lightning/propertyLayout",
         "attributes": {
           "property": "borderRadius"
         }
       },
       {
         "definition": "lightning/propertyLayout",
         "attributes": {
           "property": "layoutHeight"
         }
       },
       {
         "definition": "lightning/propertyLayout",
         "attributes": {
           "property": "layoutWidth"
         }
       }
     ]
   }
 }
}
```

Here’s how the layout looks in the component property panel in Experience Builder.



lightning/accordionLayout

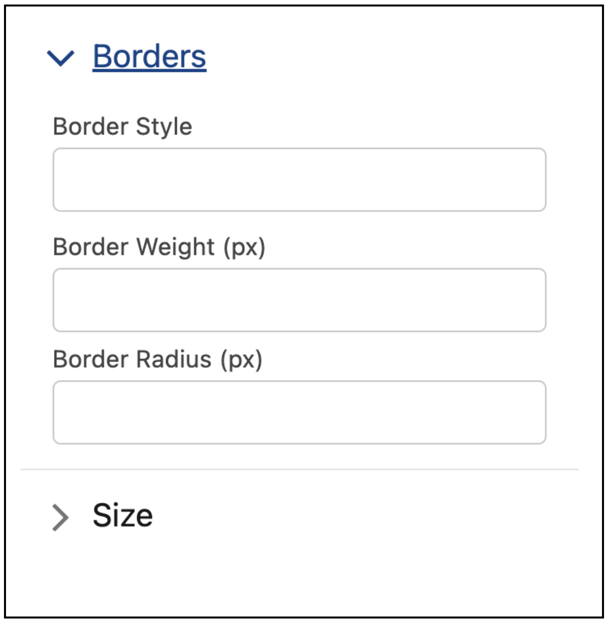
To arrange your property sheet in accordion sections, use the `lightning/accordionLayout` component. This layout can take only `lightning/accordionSectionLayout` as its children. The `lightning/accordionSectionLayout` component, in turn, can use only `lightning/propertyLayout` components as children.

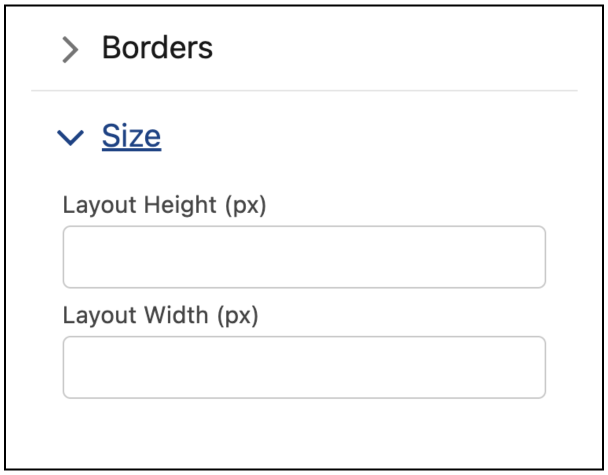
Here's an example of the properties for `borderStyle`, `borderWeight`, `borderRadius`, `layoutHeight`, and `layoutWidth` laid out in accordion sections for Borders and Size.

```
{
 "propertySheet": {
   "view": {
     "definition": "lightning/accordionLayout",
     "children": [
       {
         "definition": "lightning/accordionSectionLayout",
         "attributes": {
           "label": "Borders"
         },
         "children": [
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "borderStyle"
             }
```

```
          },
          {
            "definition": "lightning/propertyLayout",
            "attributes": {
              "property": "borderWeight"
            }
          },
          {
            "definition": "lightning/propertyLayout",
            "attributes": {
              "property": "borderRadius"
            }
          }
        ]
      },
      {
        "definition": "lightning/accordionSectionLayout",
        "attributes": {
          "label": "Size"
        },
        "children": [
          {
            "definition": "lightning/propertyLayout",
            "attributes": {
              "property": "layoutHeight"
            }
          },
          {
            "definition": "lightning/propertyLayout",
            "attributes": {
              "property": "layoutWidth"
            }
          }
        ]
      }
    ]
  }
 }
}
```

Here’s how the accordion layout looks in the component property panel in Experience Builder.





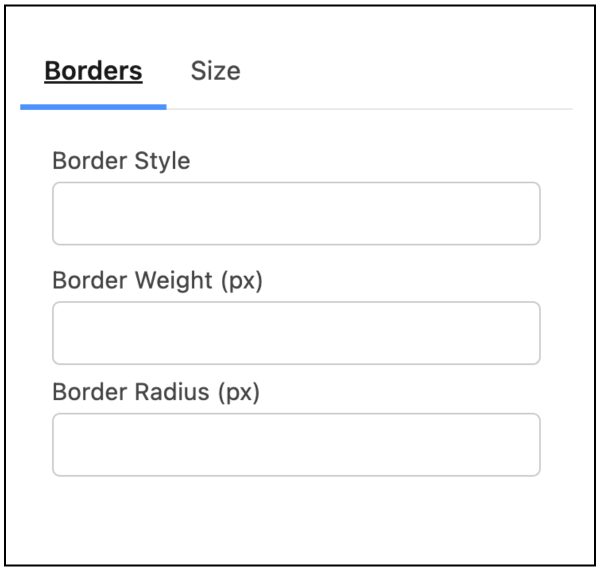
lightning/tabSetLayout

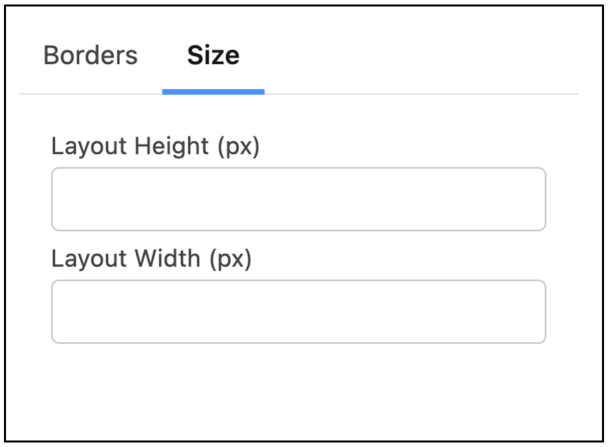
To arrange your property sheet in tabs, use the `lightning/tabSetLayout` component. This layout can take only `lightning/tabLayout` as its children. The `lightning/tabLayout` component, in turn, can take only `lightning/propertyLayouts` as children.

Here's an example of the properties `borderStyle`, `borderWeight`, `borderRadius`, `layoutHeight`, and `layoutWidth` laid out on tabs for Borders and Size.

```
{
 "propertySheet": {
   "view": {
     "definition": "lightning/tabSetLayout",
     "children": [
       {
         "definition": "lightning/tabLayout",
         "attributes": {
           "label": "Borders"
         },
         "children": [
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "borderStyle"
             }
           },
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "borderWeight"
             }
           },
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "borderRadius"
             }
           }
         ]
       },
       {
         "definition": "lightning/tabLayout",
         "attributes": {
           "label": "Size"
         },
         "children": [
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "layoutHeight"
             }
           },
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "layoutWidth"
             }
           }
         ]
       }
     ]
   }
 }
}
```

Here’s how the tab set layout looks in the component property panel in Experience Builder.





Custom Definitions

If you don't want to use one of the out-of-the-box view definitions, you can provide your own custom Lightning web component and reference it using the `definition` property. Use the `attributes` property on the view to pass values into the exposed `@api` properties of your layout component. Use the `children` property of the view to put any child components inside the default `<slot>` of your component.

Any view definitions with children are considered layout components. Layout components are used to arrange your property editors in the property sheet however you like. For example, like arranging property editors into tabs or accordions.

Any view definitions that aren't lightning/propertyLayout and have no children are considered info components. Info components aren't considered property editors and aren't required to implement the property editor contract (see Property Editor Contract). Info components are purely for informational purposes and don't contribute to editing property values the way property editors do.

When you're ready to place a property editor inside the view, use the lightning/propertyLayout definition to indicate where you want that editor to appear.

Here is an example of a view with a custom layout component and an info component.

```
{
 "propertySheet": {
   "view": {
     "definition": "c/myCustomCarouselLayout",
     "children": [
       {
         "definition": "c/infoComponent",
         "attributes": {
           "text": "Here is my display text with Helpful information."
         }
       },
       {
         "definition": "lightning/propertyLayout",
         "attributes": {
           "property": "propertyName"
         }
       }
     ]
   }
 }
}
```

Property Renderers

You can use the propertyRenderers object to override the property editors for custom types that are referenced in your own custom property type. Each lightning type provides a default property editor that's used when there are no property renderer overrides. The propertyRenderers JSON object is a map where each key-value pair maps a sub-property to a view override for that property. This view serves as a property editor override for that property.

Example: Let's take an example of a complex property type, layoutProperty, that has the sub-properties borderStyle, borderWeight, borderRadius, layoutHeight, and layoutWidth. You want to use a custom property editor for the borderStyle sub-property and use default property editors for the rest.

In this code snippet of a design.json file, a propertyRenderer overrides the default property editor for postal code by changing the definition to c/borderStyleDropdownCPE. c/borderStyleDropdownCPE is a custom property editor that you created. (For details on how to create a custom property editor, see Create a Custom Property Editor.)

Because you didn't override the property renderers for borderWeight, borderRadius, layoutHeight, or layoutWidth, the property editor shown in the property sheet for each of those properties is the default editor.

```
{
 "propertySheet": {
   "propertyRenderers": {
     "postalCode": {
       "definition": "c/borderStyleDropdownCPE"
     },
   }
 }
}
```

Full Editor Override

To use a single custom property editor for your entire complex property type, use the "$" keyword in your `propertyRenderer`.

Let's say you have a complex property type for map with two properties, `latitude` and `longitude`. You can create a custom property editor, `customMapEditor`, that plots a map. Then, in the design.json file, you can use "$" to define `customMapEditor` as the editor for all the properties in the type—in this case, `latitude` and `longitude`.

```
{
 "propertySheet": {
   "propertyRenderers": {
     "$": {
       "definition": "c/myCustomMapEditor"
     }
   }
 }
}
```

Include the Namespace Prefix for Metadata References

When you reference any external metadata in your custom property type, always use "c" as the namespace prefix for the reference to metadata in the current org. This includes referencing other custom property types, custom property editors, and labels. For example, a postal code custom property editor reference in your current org should be `c/postalCodePicker`. If your org has a namespace defined, still use "c" as the prefix. The exception is if the referenced custom property editor or property type is from a managed package. In that case, use the package's namespace as the prefix. For example, `isv/postalCodePicker`.

Here's an example schema.json file using local org metadata references for label and custom property types.

```
{
 "title": "Address Type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "postalCode": {
     "lightning:type": "c__postalCodeType",
     "title": "{!$Label.c.PostalCodeTitle}"
   }
 }
}
```

Here's an example design.json file using local org metadata for a custom property editor.

```
{
 "propertySheet": {
   “propertyRenderers”: {
      “postalCode”: {
         “definition”: "c/postalCodePicker"
       }
   }
}
```

See Also
Create a Custom Property Editor
The Editor Resolution Algorithm

The Editor Resolution Algorithm

To understand which property editor appears in the property sheet for a given property, it can help to know the algorithm that the property sheet uses to choose the correct property editor to render. Here is a representation of the algorithm that the property sheet uses to show the correct editor in Experience Builder.

```
1. If the design file is defined for a property type:
      a. If propertyRenderers is present and contains a full editor
         override ($), use the given definition as the property editor.
      b. Else if the current type is a primitive lightning:type, use the
         property editor defined by the property type schema’s
         lightning:type referenced. Repeat algorithm for the
         lightning:type referenced to find the lightning:type’s property
         editor.
      c. Else if the current type is a complex lightning:type (object),
         render the complex property editor as a layout of sub-property
         editors.
          i.   Calculate the layout of the object properties:
                 1. If a view is not defined, the layout of the object
                    properties defaults to a vertical layout of all of
                    the properties’ editors, in the order the properties
                    are defined in the property type schema.
                 2. Else if a view is defined, use the layout specified.
         ii.   Calculate the property editors for each property in the
               layout:
                 1. For each object sub-property:
                       a. If propertyRenderers contains the property,
                          use the property editor override defined.
                       b. Else then use the property editor from the
                          lightning:type in the schema for that
```

```
                              sub-property. Repeat algorithm for the
                              lightning:type referenced to find that
                              lightning:type's property editor.
   2. Else if the design file is not defined:
         a. If the current type is a primitive type, use the property editor
            from the lightning:type. Repeat the algorithm for the
            lightning:type referenced to find its property editor.
         b. Else if the current type is a complex type, all the properties
            are laid out in vertical order as defined in the schema. For
            each sub-property:
             i.   Use the property editor found at its lightning:type
                  referenced.
```

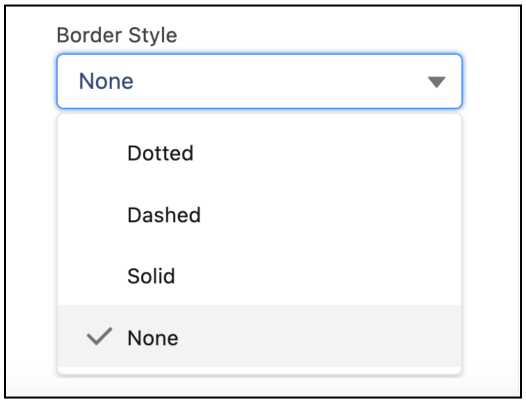
Design Example

Let's look at a custom component, MyCustomComponent, for which you created an experiencePropertyTypeBundle for the complex property `layoutProperty`. `layoutProperty` has the sub-properties `borderStyle`, `borderWeight`, `borderRadius`, `layoutHeight`, and `layoutWidth`. Now you want to arrange the sub-properties in tabs, and you want to use a custom property editor for the sub-property `borderStyle`.

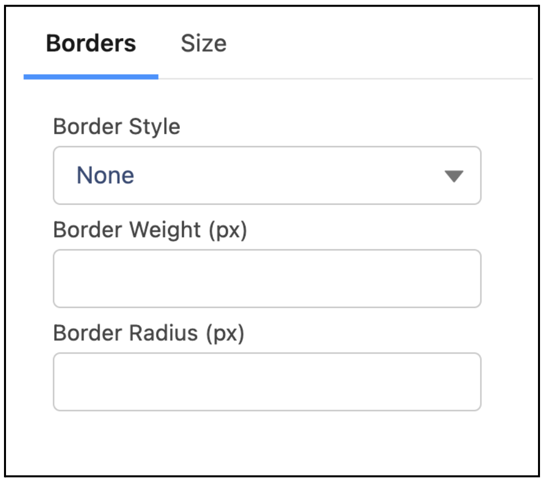
Here's what the design.json file looks like. We use the propertyRenderers object to define a property editor override for borderStyle. To arrange the sub-properties in tabs, we use the lightning/tabSetLayout in the view definition

```
{
 "propertySheet": {
   "propertyRenderers": {
     "borderStyle": {
       "definition": "c/borderStyleDropdownCPE"
     }
   },
   "view": {
     "definition": "lightning/tabSetLayout",
     "children": [
       {
         "definition": "lightning/tabLayout",
         "attributes": {
           "label": "Borders"
         },
         "children": [
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "borderStyle"
             }
           },
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "borderWeight"
             }
           },
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "borderRadius"
             }
           }
         ]
       },
       {
         "definition": "lightning/tabLayout",
         "attributes": {
           "label": "Size"
         },
         "children": [
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "layoutHeight"
             }
           },
           {
             "definition": "lightning/propertyLayout",
             "attributes": {
               "property": "layoutWidth"
             }
           }
         ]
       }
     ]
   }
 }
}
```

Here's how the custom property editor appears in the component property panel in Experience Builder.



And here's how layoutProperty looks in the component property panel in Experience Builder.



Border Style

None

Border Weight (px)

Border Radius (px)

Lightning Property Types

In the schema.json file of an ExperiencePropertyTypeBundle, you specify a `lightning:type` for each property. The `lightning:type` is syntactic sugar for the JSON schema `$ref` keyword that's used to combine schemas. (See Understanding JSON Schema: $ref). The main difference between `lightning:type` and `$ref` is that you don't have to enter a full URL path for your `lightning:type` reference. Instead, you can just use the fully qualified name of an `ExperiencePropertyTypeBundle`.

Salesforce created several Lightning property types—out-of-the-box property types that act as the basic types that you can reference to structure a more complex schema. For example, you can use the `lightning:type` called `lightning__objectType` to reference a schema with the underlying type "object."

It's important to understand the underlying type used for each `lightning:type` to understand how the Lightning property type is validated. By combining the basic Lightning types, you can construct any property type of higher complexity. Each Lightning property type includes a default property editor, so if you use one of these property types, you don't have to create a property editor yourself.

Like JSON Schema types, each Lightning property type comes with type-specific keywords that apply only to that type. The ensuing subsections describe the keywords available and the default property editor associated with each of these Lightning property types.

- lightning__booleanType
- lightning__dateType
- lightning__dateTimeType
- lightning__integerType
- lightning__multilineTextType
- lightning__numberType
- lightning__objectType

- lightning__richTextType
- lightning__textType
- lightning__urlType
- lightning__booleanType

lightning__booleanType

This property type maps to an underlying JSON schema boolean type. (See Understanding JSON Schema: boolean.) It's expressed as a toggle property editor.

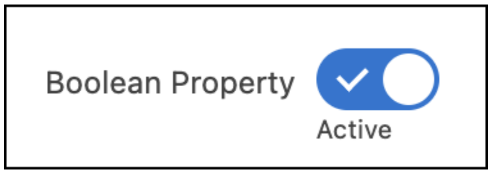
The `lightning__booleanType` property type includes these configurable parameters.

Keyword	Required	Type	Description
title	Yes	String	The text associated with the property label and used as the display label for the property editor
description	No	String	Additional information about the property

Here's a sample custom object property type that contains a property of the `lightning__booleanType` and configures its parameters.

```
{
 "title": "Custom Property with Boolean Property Type",
 "description": "This explains how to refer a OOTB booleanType property type
in a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "booleanProperty": {
     "lightning:type": "lightning__booleanType",
     "title": "Boolean Property",
     "description": "description of this property"
   }
}
```

Here's how the default property editor for `lightning__booleanType` appears in the component property panel in Experience Builder. Notice that the title is used as the property editor label.



lightning__dateType

This property type is a string used to specify a date in yyyy-mm-dd format. (That is, four-digit year, two-digit month, and two-digit date format.)

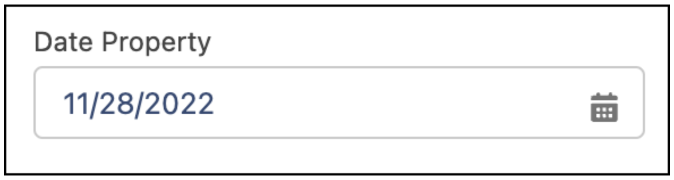
These are the parameters used to configure the `lightning__dateType` property.

Keyword	Required	Type	Description
title	Yes	String	The text associated with the property label and used as the display label for the property editor
description	No	String	Additional information about the property

Here's a sample custom object property type that contains a property of the `lightning__dateType` and configures its parameters.

```
{
 "title": "Custom Property with Date Property Type",
 "description": "This explains how to refer a OOTB dateType property type in
a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "boolean": {
     "lightning:type": "lightning__dateType",
     "title": "Date Property",
     "description": "description of this property"
   }
}
```

Here's how the default property editor for `lightning__dateType` appears in the component property panel in Experience Builder.



lightning__dateTimeType

This property type describes a complex property type that's an object with a `dateTime` and optional `timeZone` properties. Use this property type to specify date and time together.

Because `lightning__dateTimeType` is a predefined complex property type, its value is an object. The following is the schema for the object property value that this type describes.

Keyword	Required	Type	Description
dateTime	Yes	String	This field must be specified in the format yyyy-MM-dd'T'HH:mm:ss.SSSZ
timeZone	No	String	Use the timezone attribute to specify a time zone in IANA time zone database format.

Here are some examples of valid data for a property value of this type. This sample shows an object with both `dateTime` and `timeZone` values set.

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

This sample shows an object with only the `dateTime` value set.

```
{
 "dateTime": "2012-05-31T01:30:05.000Z"
}
```

Here are some examples of invalid data for a property value of this type. This sample shows an object without the required `dateTime` field.

```
{
  "timeZone": "Asia/Kolkata"
}
```

This sample shows an object with an invalid `timeZone` format (without milliseconds).

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

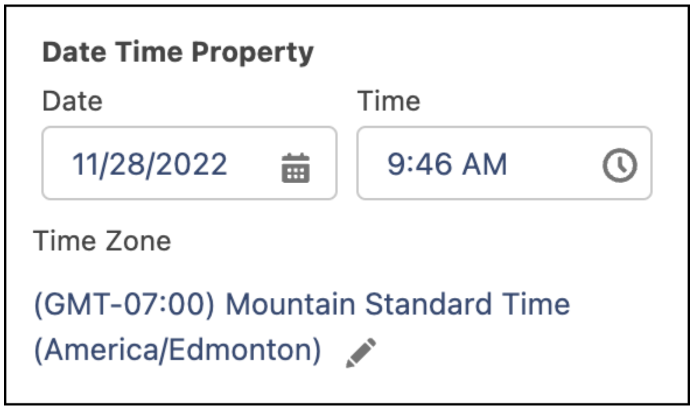
The `lightning__dateTimeType` property type includes these configurable parameters.

Keyword	Required	Type	Description
title	Yes	String	The text associated with the property label and used as the display label for the property editor
description	No	String	Additional information about the property

Here's a sample custom object property type that contains a property of the `lightning__dateTimeType` and configures its parameters.

```
{
 "title": "Custom Property with DateTime Property Type",
 "description": "This explains how to refer a OOTB dateTimeType property
type in a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "dateTime": {
     "lightning:type": "lightning__dateTimeType",
     "title": "Date Time Property",
     "description": "description of this property"
   }
}
```

Here's how the default property editor for `lighting__dateTimeType` appears in the component property panel in Experience Builder.


Date Time Property
Date: 11/28/2022
Time: 9:46 AM
Time Zone
(GMT-07:00) Mountain Standard Time (America/Edmonton)

lightning__integerType

Use this property type to specify integers. This property will be used for integral numbers. This property type maps to an underlying JSON schema integer type. (See Understanding JSON Schema: integer.)

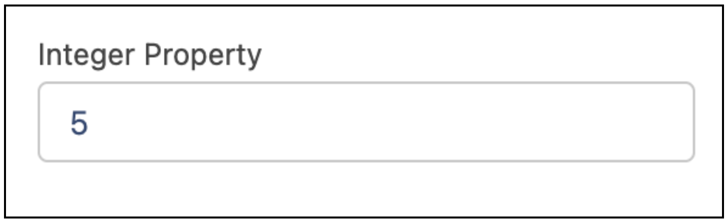
The lightning__integerType property type accepts these configurable parameters.

Keyword	**Required**	**Type**	**Description**
maximum	No	Number	Sets the maximum value that a number can be for this property
minimum	No	Number	Sets the minimum value that a number can be for this property
title	Yes	String	The text associated with the property label and used as the display label for the property editor
description	No	String	Additional information about the property

Here's a sample custom object property type that contains a property of the lightning__integerType and configures its parameters. Notice that this example includes a default, which is set inside the property editor.

```
{
 "title": "Custom Property with Integer Property Type",
 "description": "This explains how to refer a OOTB integerType property type
in a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "integer": {
     "lightning:type": "lightning__integerType",
     "title": "Integer Property",
     "description": "description of this property",
     "minimum": 100,
     "maximum": 200,
     "default": 5
   }
}
```

Here's how the default property editor for lightning__integerType appears in the component property panel in Experience Builder.



lightning__multilineTextType

This property type is similar to `lightning__textType`, but it accommodates a larger maximum character length and a property editor for larger text input. This lightning property type maps to an underlying JSON string type. (See Understanding JSON Schema: string.)

The `lightning__multilineTextType` property type accepts these configurable parameters.

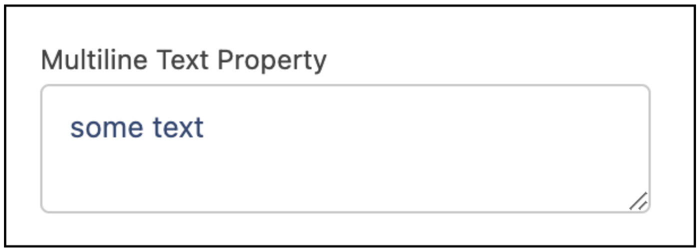
Keyword	Required	Type	Description
maxLength	No	Number	This field sets the maximum length of characters for the property value. Values can range from 0 to 2,000
minLength	No	Number	This field sets the minimum length of characters for the property value. Values can range from 0 to 2,000
title	Yes	String	The text associated with the property label and used as the display label for the property editor
description	No	String	Additional information about the property

Here's a sample custom object property type that contains a property of the `lightning__multilineTextType` and configures its parameters. Notice that this example includes a default, which is set inside the property editor.

```
{
 "title": "Custom Property with MultilineText Property Type",
 "description": "This explains how to refer a OOTB multilineTextType
property type in a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "text": {
     "lightning:type": "lightning__multilineTextType",
     "title": "Multiline Text Property",
     "description": "description of this property",
     "maxLength": 200,
     "minLength": 1,
     "default": "some text",

   }
}
```

Here’s how the default property editor for `lightning__multilineTextType` appears in the component property panel in Experience Builder.


Multiline Text Property

some text

lightning__numberType

Use this property type to specify numbers. Number type is validated as a decimal number, also known as a float in some programming languages. This property type maps to an underlying JSON schema number type. (See Understanding JSON Schema: number.)

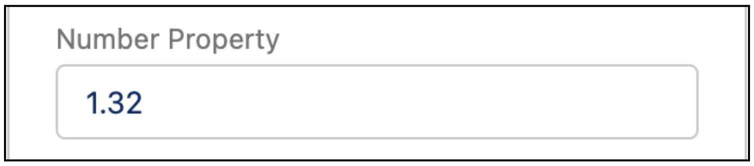
The `lightning__numberType` property type accepts these configurable parameters.

Keyword	Required	Type	Description
maximum	No	Number	Sets the maximum value that a number can be for this property
minimum	No	Number	Sets the minimum value that a number can be for this property
title	Yes	String	The text associated with the property label and used as the display label for the property editor
description	No	String	Additional information about the property

Here’s a sample custom object property type that contains a property of the `lightning__numberType` and configures its parameters. Notice that this example includes a default, which is set inside the property editor.

```
{
 "title": "Custom Property with Number Property Type",
 "description": "This explains how to refer a OOTB numberType property type
in a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "number": {
     "lightning:type": "lightning__numberType",
     "title": "Number Property",
     "description": "description of this property",
     "minimum": .555,
     "maximum": 20.555,
     "default": 5.55,
   }
  }
}
```

Here's how the default property editor for `lightning__numberType` appears in the component property panel in Experience Builder.



lightning__objectType

Use the `lightning__objectType` to create object property types. This property type is a complex type—it can contain sub-properties, each with its own property type. In other words, you can use this property type to group other property types. This property type maps to an underlying JSON schema object type. (See Understanding JSON Schema: object.)

The `lightning__objectType` accepts these configurable parameters.

Keyword	Required?	Type	Description
title	yes	String	The text associated with the property label and used as the display label for the property editor
description	no	String	Additional information about the property
required	no	String[]	Specifies an array of strings where each string is a property name in the object's sub-properties. All property names included in this array must be present in order for the object to be valid.
properties	yes	PropertyType	Specifies the map of sub-properties for

			the property type. Each key of the map is a property name, and the value is a `PropertyType` schema.

Here's an example of a valid use of the `lightning__objectType` type.

```
{
 "title": "Custom Property with Object Property Type",
 "description": "This explains how to refer a OOTB objectType property type
in a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "url": {
     "lightning:type": "lightning__urlType",
     "title": "URL property",
     "description": "description of this property",
     "lightning:allowedUrlSchemes": ["https"]
   },
   "number": {
     "lightning:type": "lightning__numberType",
     "title": "Number property",
     "description": "description of this property",
     "minimum": 100,
     "maximum": 200
   },
   "boolean": {
     "lightning:type": "lightning__booleanType",
     "title": "Boolean property",
     "description": "description of this property"
   }
 }
}
```

Here's an example of invalid code, declaring properties without using the `lightning__objectType`. This schema would be blocked on save.

```
{
 "title": "Invalid Custom Property Without Object Property Type",
 "description": "This explains how a complex property type needs
objectType property type",
  "properties": {
    "url": {
      "lightning:type": "lightning__urlType",
      "title": "title of this property",
      "description": "description of this property",
    }
  }
}
```

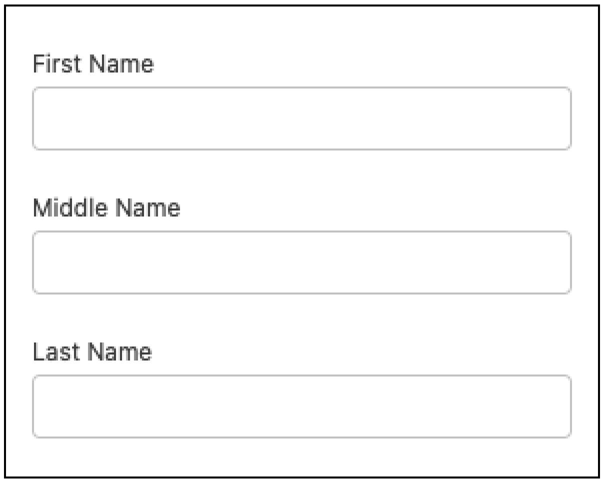
This example of invalid code shows an `objectType` with an empty set of properties.

```
{
  "title": "Invalid Custom Property with Empty Properties",
  "lightning:type": "lightning__objectType",
  "properties": { }
}
```

This example of an invalid use of `lightning__objectType` shows an `objectType` nested inside of itself.

```
{
 "title": "Custom Property with Object Property Type",
 "description": "This explains how to refer a OOTB objectType property type
in a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "nestedProperty": {
     "lightning:type": "lightning__objectType",
      "properties": {
        "url": {
          "lightning:type": "lightning__urlType",
          "title": "title of this property",
          "description": "description of this property",
          "lightning:allowedUrlSchemes": ["https"]
        }
     }
   }
}
```

By default, the object properties are laid out vertically, in the order that the properties are declared in the schema. Here's how an object with the `firstName`, `middleName`, and `lastName` properties in that order appears in the component property panel in Experience Builder.



If you want a different layout or order, you can use view in the design.json file to override the design configuration and arrange the properties in accordion sections or on tabs.

lightning__richTextType

This property type lets users add, edit, and delete rich text. The maximum length of a text field can't exceed 100,000 characters. (See the Rich Text Editor documentation for more information on the editor supports.)

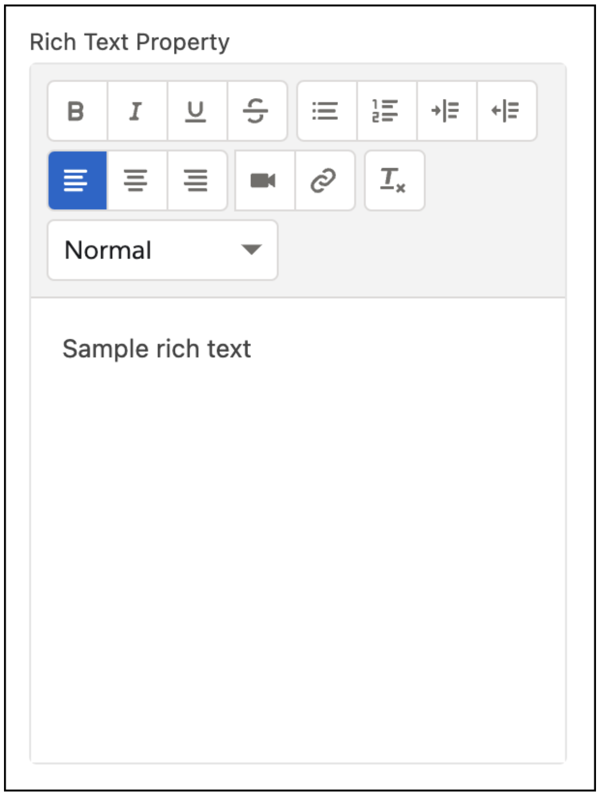
The `lightning__richTextType` property type includes these configurable parameters.

Keyword	Required	Type	Description
maxLength	No	Number	This field sets the maximum length of characters for the property value. Values can range from 0 to 100,000
minLength	No	Number	This field sets the minimum length of characters for the property. Values can range from 0 to 100,000
title	Yes	String	The text associated with the property label and used as the display label for the property editor
description	No	String	Additional information about the property

Here's a sample custom object property type that contains a property of the `lightning__richTextType` and configures its parameters. Notice that this example includes a default, which is set inside the property editor.

```
{
 "title": "Custom Property with RichText Property Type",
 "description": "This explains how to refer a OOTB richTextType property
type in a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "richText": {
     "lightning:type": "lightning__richTextType",
     "title": "title of this property",
     "description": "description of this property",
     "maxLength": 90000,
     "minLength": 100,
     "default": "Sample rich text",
   }
}
```

Here's how the default property editor for `lightning__richTextType` appears in the component property panel in Experience Builder.



Sample rich text

lightning__textType

Use this property type for text fields such as titles and descriptions. The maximum character length of this text field is 255 characters. This lightning property type maps to an underlying JSON schema string type. (See [Understanding JSON Schema: string](#).)

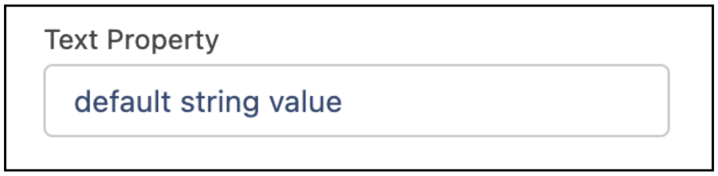
The `lightning__textType` property type includes these configurable parameters.

Keyword	Required	Type	Description
maxLength	No	Number	This field sets the maximum length of characters for the property. Values can range from 0 to 250
minLength	No	Number	This field sets the minimum length of characters for the property. Values can range from 0 to 250
title	Yes	String	The text associated with the property label and used as the display label for the property editor
description	No	String	Additional information about the property

Here's a sample custom object property type that contains a property of the `lightning__textType` and configures its parameters. The example includes a default, which is set inside the property editor.

```
{
 "title": "Custom Property with Text Property Type",
 "description": "This explains how to refer a OOTB textType property type in
a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "text": {
     "lightning:type": "lightning__textType",
     "title": "Text Property”,
     "description": "description of this property",
     "maxLength": 200,
     "minLength": 1,
     "default": "default string value",
   }
}
```

Here's how the default property editor for `lightning__textType` appears in the component property panel in Experience Builder.



lightning__urlType

Use this property type for URL values. This type uses a configurable parameter called `lightning:allowedUrlSchemes` to let you specify the URL schemes that this type can validate against.

The `lightning__url`Type property includes these configurable parameters.

Keyword	**Required**	**Type**	**Description**
lightning:allowedUrlSchemes	No	String[]	Array of the supported schemes for the url type to validate against. Must be one of the following values: `https`, `http`, `relative`, `mailto`, `tel`. If none specified, defaults to [`https`, `http`, `relative`]
title	Yes	String	The text associated with the property label and used as the display label for the property editor
description	No	String	Additional information about the property

Here's a sample custom object property type that contains a property of the `lightning__urlType` and configures its parameters. This example includes `lightning:allowedUrlSchemes` to ensure that the URL entered includes https. In this code, valid data for the `url` property is https://google.com. But http://google.com is invalid, because http isn't included in `lightning:allowedUrlSchemes`.

```
{
 "title": "Custom Property with Url Property Type",
 "description": "This explains how to refer a OOTB urlType property type in a custom property type",
 "lightning:type": "lightning__objectType",
 "properties": {
   "url": {
     "lightning:type": "lightning__urlType",
     "title": "URL Property",
     "description": "description of this property",
     "default": "https://sampleurl.com",
     "lightning:allowedUrlSchemes": ["https"]
   }
}
```

Here's how the default property editor for `lightning__url`Type appears in the component property panel in Experience Builder.

Create a Custom Property Editor

To replace the default property editor for a given property type in a custom Lightning web component, create a custom property editor. First you create a Lightning web component, including HTML and CSS, to serve as the property editor. Then you link your custom property editor back to the component properties that you want to use this editor for.

The Property Editor Contract

To work successfully with the property sheet and function smoothly in Experience Builder, the component that you create for your property editor must satisfy the property editor contract. The contract requires your custom property editor component to include these public properties in its element class: label, description, required, value, errors, and schema. After the contract is satisfied, you have access to all existing Lightning web component tools, such as UI APIs and Apex, to make the component behave the way you want it to. (For in-depth information on creating Lightning web components, see the Lightning Web Component Developer Guide.)

Here's an interface that outlines the required properties of the property editor contract. Each required property is injected into the property editor from the property sheet with the relevant information. Think of these properties as the inputs to the property editor.

```
interface PropertyEditorContract {
 label: string;
 description: string;
 required: boolean;
 value: any;
 errors: PropertyError[];
 schema: JSONSchema
}

interface PropertyError {
 message: string;
}
```

When users enter a value into the property editor, the property sheet expects the editor to throw a single CustomEvent called `valuechange` to send the value to Experience Builder. Here's a code sample that shows an interface for the object payload expected for the `valuechange` event. Think of this interface as the output of the property editor.

```
interface PropertyChangedEventPayload {
 value: any
}
```

After the event fires, the property sheet validates the entered value against the property editor's property type schema and persists the value to the component on the canvas in Experience Builder. If any errors occur during the validation process, they're injected back into the editor via the errors property of the property editor. (See General Guidelines for more information to help you avoid common mistakes when you create the property editor component.)

Let's say you have a custom Lightning web component called MyCustomComponent, and you want to create a custom property editor for the `textAlignment` property in this component. The default editor for `type=String` is a combobox, and you want to replace that with a button group where users can visually toggle the alignment . So you create a custom Lightning web component to act as the property editor, called `alignmentCPE` .

The component class of `alignmentCPE` must follow the property editor contract. Here's a code snippet showing a basic editor component class for the editor. Notice that the public properties of the component follow the property editor contract, and that the editor is firing the `valuechange` event when users blur their changes.

```
export default class AlignmentCPE extends LightningElement {
 @api value;
 @api label;
 @api schema; // the JSON Schema derived from the accompanying property
type.
 @api errors;

 @track buttons = [...]

 handleAlignmentClick(event) {
        const value = event.target.value;
        this.value = value;
        this.dispatchEvent(new CustomEvent("valuechange", {detail: {value:
this.value}}));
    }
}
```

Build the Property Editor HTML and CSS

The HTML markup and CSS for your custom property editor don't have the additional restrictions that the property editor contract imposes on the component class. You can make the editor look and feel however you like. For example, you can assign the font, type size, and text color of your choice. (See General Guidelines for information that can help you avoid common mistakes in editor development.)

Let's return to your custom Lightning web component, MyCustomComponent. You want to create a custom property editor for `textAlignment` so that your Experience Builder users can select alignment from a button group. Here's how the HTML for this custom property editor can look.

```
<template>
   <div>
       <lightning-button-group onclick={handleAlignmentClick}>
           <template for:each={buttons} for:item="button">
               <lightning-button-icon-stateful
                   key={button.value}
                   value={button.value}
                   selected={button.selected}
                   icon-name={button.icon}></lightning-button-icon-stateful>
           </template>
       </lightning-button-group>
   </div>
</template>
```

XML Configuration

The js-meta.xml file of your custom property editor component must include the `lightning__PropertyEditor` target to indicate that this component is a property editor.

Here's how the js-meta.xml for alignmentCPE looks when the `lightning__propertyEditor` target is set.

```
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata"
fqn="alignmentCPE">
   <apiVersion>58.0</apiVersion>
   <isExposed>true</isExposed>
   <masterLabel>Alignment Editor Component</masterLabel>
   <targets>
     <target>lightning__PropertyEditor</target>
   </targets>
</LightningComponentBundle>
```

Link a Custom Property Editor or Property Type to Your Custom Component

After you create your custom property editor or custom property type, link it back to the property of the custom Lightning web component. Link a custom property editor to your component using the `editor` attribute, and link a custom property type to your component using the `type` attribute.

Reference a Custom Property Editor by Editor Attribute

A component property can reference a custom property editor directly through the new `editor` attribute. The `editor` attribute is a reference to a custom property editor by a `fullyQualifiedName` string. This option is best if you want only to override the default user experience, without needing the additional validations of a custom property type.

In this example, you specify an `editor` reference to your custom `alignmentCPE` for the `textAlignment` of MyCustomComponent.

```
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata"
fqn="lwcWithCustomPropertyType">
 <masterLabel>My Custom Component</masterLabel>
 <apiVersion>58.0</apiVersion>
 <targets>
   <target>lightningCommunity__Page</target>
   <target>lightningCommunity__Default</target>
 </targets>
 <targetConfigs>
  <targetConfig targets="lightningCommunity__Default">
    <property  name="textAlignment" type="String" editor="c/alignmentCPE"
label="Text Alignment"/>
  </targetConfig>
 </targetConfigs>
</LightningComponentBundle>
```

Besides referencing a custom property editor by `editor` attribute in the js-meta.xml file, you can also reference a property editor in the design.json file. See Design.json for details.

Reference a Custom Property Type by Type Attribute

A component property can reference a custom property type via the existing `type` attribute. Previously, the `type` property was limited to a specific set of types such as String, Integer, Boolean. But now the `type` property can be a `fullyQualifiedName` string that references an existing `ExperiencePropertyTypeBundle`, whether it is an out-of-the-box Lightning property type or a custom property type.

In this example, you add to the js-meta.xml file for MyCustomComponent and use the `type` attribute to reference the custom property type `layoutProperty`.

```
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata"
fqn="lwcWithCustomPropertyType">
 <masterLabel>My Custom Component</masterLabel>
 <apiVersion>58.0</apiVersion>
 <targets>
   <target>lightningCommunity__Page</target>
   <target>lightningCommunity__Default</target>
 </targets>
 <targetConfigs>
   <targetConfig targets="lightningCommunity__Default">
       <property name="layoutProperties" type="c__layoutProperty"
label="Layout" />
   </targetConfig>
 </targetConfigs>
</LightningComponentBundle>
```

In this example, you use the type attribute to reference a lightning property type for the `articleDate` property.

```
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata"
fqn="lwcWithCustomPropertyType">
 <masterLabel>My Custom Component</masterLabel>
 <apiVersion>58.0</apiVersion>
 <targets>
   <target>lightningCommunity__Page</target>
   <target>lightningCommunity__Default</target>
 </targets>
 <targetConfigs>
   <targetConfig targets="lightningCommunity__Default">
     <property  name="articleDate" type="lightning__dateType" label="Date
Updated" />
   </targetConfig>
 </targetConfigs>
</LightningComponentBundle>
```

Overriding Custom Property Type Attributes

You can override some of a custom property type's attributes in the `<property>` definition. In this example, the layoutProperty label, “Layout,” overrides the “Layout Properties” title specified in the Schema Example.

```
 <property  name="layoutProperties" type="c__layoutProperty"
```

```
label="Layout"/>
```

You can override the attributes `label`, `description`, and `default` in a custom property type. Label overrides a custom property type's `title` attribute.

Escaping Special XML Characters in Default Values

The `default` attribute lets you set the value for the property before an end user has a chance to change the input. When you set the `default` value of a custom property type from the js-meta.xml file, you need to escape any special characters in that value. This is standard practice in XML documents to ensure that special characters in the value don't affect the validity of the whole XML document. This consideration generally isn't applicable to most primitive types. For properties that directly or indirectly reference a custom property type that's defined as `lightning__objectType`, `lightning__dateTimeType`, or `lightning__richTextType` in its schema, you may need to escape any special characters in those values.

For example, if the component has a property tag that uses the `lightning__richTextType`, any HTML or XML usage in the `default` such as "<strong>bold text</strong>" must be specified as:

```
<property name="richTextProperty" type="lightning__richTextType"
default="&gt;&lt;strong&gt;bold text&lt;/strong&gt;" />
```

Similarly, if a component has a property type reference that uses the `lightning__objectType`, a default object value such as "{"state":"California", "city":"San Francisco"}" must be specified as:

```
<property name="addressProperty" type="c__addressType"
default="{&quot;state&quot;:&quot;California&quot;,&quot;city&quot;:&quot;Sa
n Francisco&quot;}" />
```

Supported Attributes for the Property Tag in the js-meta.xml File

In the js-meta.xml file for a custom Lightning web component, you can use these attributes for the property tag.

type

Salesforce supports five out-of-the-box property types for Lightning web components in Experience Builder: String, Integer, Boolean, ContentReference, and Color. You can use all these types together with properties that reference custom property types.

filter

This attribute is supported only if the property has `type=ContentReference`. It specifies the

Salesforce CMS content types to display in the component in Experience Builder. These are the valid values.

- cms_document
- cms_image
- cms_video
- news
- custom_type, where custom_type is the name of a custom content type

screenResponsive

For properties that reference custom property types which have an underlying JSON schema type of string, integer, or number, you can specify this attribute to indicate whether the property is screen responsive.

exposedTo

This attribute is required if you use the screenResponsive attribute. The valid value is `css`.

translatable

When you specify `translatable="true"`, you indicate that this component property can hold different values for each language supported on your Experience Cloud site. You can specify the translatable attribute in a property tag only when the custom property type that's being referenced has an underlying JSON schema type of string. For example, you can add `translatable=”true”` to properties that reference these types.

- `lightning__textType`
- `lightning__multlineTextType`
- `lightning__richTextType`
- `lightning__urlType`
- `lightning__dateType`

You can also add `translatable="true" to` any custom property type that references one of these types in its schema.json file.

When you specify `translatable="true"` for a property of `type="lightning__richTextType"`, the property's value is exported in rich text format in the .xlf file via the Export Content feature in Experience Builder. All other types are exported as plain text.

Testing

After you link your custom property type or editor to your custom Lightning web component, deploy the component to your org and test it to see if it works. Drag the component onto the canvas in Experience Builder, and look at the component property panel. Make sure that the expected property editors appear, and that the validation matches what you expect based on your property type schema. Use your custom property editor to change that property value, and verify that the component property in the canvas is updated accordingly.

General Guidelines

We recommend that you follow these guidelines when you create a custom property editor. They can help you avoid common mistakes and deliver a consistent end-user experience.

Don't use the property editor to validate user input. The property editor is intended to affect only the user interface in Experience Builder. Make sure that the property editor always sends its input value without changing or processing the user input value. The property sheet takes care of the validation of the value. To avoid inconsistent validation, the property sheet ensures that the property value is valid according to the property's type schema. Any validation errors that the property sheet finds are injected into the property editor via the `errors` property.

To illustrate this guideline, this code sample provides an example of what *not* to do. In the sample, the code violates this guideline by performing a validation check on the value entered by the user and setting its own error message. The code also bypasses firing the `valuechange` event when this special validation condition is met.

```
export default class CustomPropertyEditor extends LightningElement {
 @api value;
 @api label;
 @api schema;
 @api errors;
 handleOnBlur(event) {
   // Don't do this
   if (this.value.contains("myCustomValidation")) {
     this.errors = [{ message: "My Special validation is broken"}];
     return;
  }
   this.dispatchEvent(new CustomEvent("valuechange", {detail: {value:
this.value}}));
 }
}
```

Avoid creating flyouts, popouts, and modals. These user experience patterns can interrupt the Experience Builder user's editing experience. Use caution - these patterns don't always behave as expected.

Avoid using CSS to define the absolute position of an element. This pattern also doesn't always behave as expected in various builder scenarios. (See MDN Web Docs: position for more information.)

Avoid setting CSS width properties for your property editor. Don't set CSS properties to determine the width of your property editor. Setting width properties affects the property panel size in Experience Builder and can cause unexpected issues with user experience.

Don't fire the `valuechange` event too often. A canvas refresh is triggered whenever the property sheet handles the `valuechange` event fired from the property editor. Too many refreshes of the canvas per second hinders the performance of Experience Builder. For example, when you bind `valuechange` to events such as `keydown` or `mousemove` for an input, the canvas refreshes many times per second. We recommend binding to a committal events, such as `blur`, instead.

This example shows what *not* to do. Here the `valuechange` event fires on every keypress of an input. An Experience Builder user types rapidly can unknowingly trigger many canvas refreshes and slow down the editing experience of this component.

```
export default class AlignmentCPE extends LightningElement {
 @api value;
 @api label;
 @api errors;
// Use “blur” instead
 handleOnKeyDown(event) {
   this.dispatchEvent(new CustomEvent("valuechange", {detail: {value:
this.value}}));
 }
}
```

Conclusion

Remember long ago, when you set out on your quest to create a custom property editor? You have made it! We hope this guide has been useful. If you have any questions, suggestions for improving the guide, or other feedback, please let us know in the Experience Cloud Trailblazer community. Here's to your success creating custom property editors to develop dynamic, intuitive components for your sites.