Creating Custom Widgets

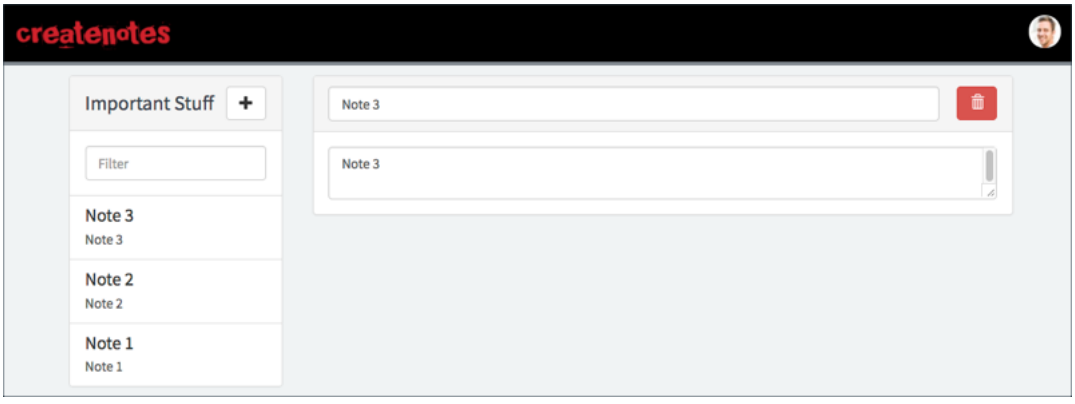
ARTICLE (1 OF 35)

Creating Custom Widget Objectives

In this module you will learn to:

- Clone baseline widgets
- Write widget logic
  - HTML Template
  - Client Script
  - Server Script
- Test Widgets
  - Preview
  - Test Page
  - JavaScript Console
- Use the Widget APIs
- Use the widget global objects
  - *data*
  - *input*
  - *options*
- Create and use directives
- Define and use widget options
- Respond to record changes which occur outside of Service Portal

You will practice widget development skills by creating the *CreateNotes* portal and two widgets.



NOTE: This module assumes familiarity with the technologies that are part of AngularJS and with the ServiceNow client-side and server-side APIs . Students should:

- be able to navigate in Service Portal
- have the ability to write scripts using the ServiceNow APIs for both client-side and server-side scripts
- be able to read and write HTML, CSS, and AngularJS
- understand how to use Bootstrap
- be able to use browser-based debugging tools

For more details on navigating in Service Portal, see the Service Portal Introduction (https://developer.servicenow.com/dev.do#!/learn/courses/utah/app_store_learnv2_serviceportal_utah_service_portal/app_store_learnv2_serv module.

ARTICLE (2 OF 35)

About This Learning Module

IMPORTANT: The content in this learning module was last updated for the ***San Diego*** ServiceNow release and was not updated for the ***Utah*** release. You may see differences between the ***Utah*** release and the content in this learning module.

The *Global* application, *Hello World 1* widget, and *My Currency Widget* widget are used throughout this learning module to introduce and demonstrate the concepts and processes behind creating a widget. You do not build the *Global* application or the *Hello World 1* and *My Currency Widget* widgets.

You will develop the *CreateNotes* application and portal in the hands-on exercises.

Exercises are indicated in three ways:

- *Exercise* icon in the Navigation pane.
- *Exercise* icon and the word *Exercise* at the top of the page.
- The word *Exercise* or the word *Challenge* in the page title.

The *CreateNotes* application and portal allows users to create note records. You will use source control to begin with all *CreateNotes* application files needed for this learning module.

ARTICLE (3 OF 35)

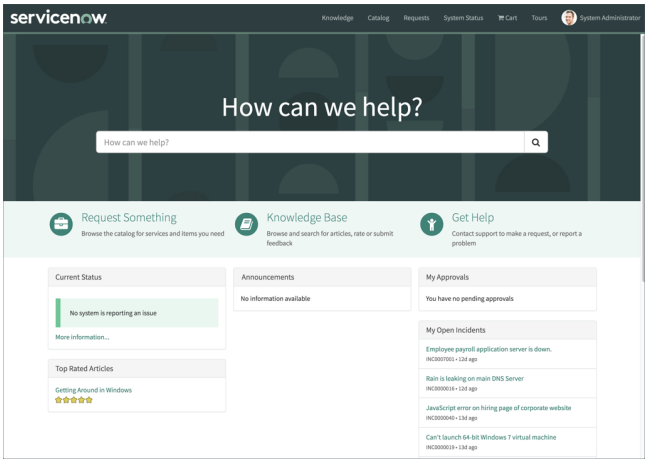
What Are Widgets?

Widgets are reusable components which make up the functionality of a portal page. Widgets define what a portal does and what information a user sees. ServiceNow provides a large number of baseline widgets. Examples include:

- Approvals
- Knowledge Base
- My Requests

- Carousel
- Catalog content
- Popular questions
- Search

Some example widgets:



Widgets are AngularJS directives. When a page is loaded, a directive is created for each widget on the page. Widgets are tightly coupled to the server-side JavaScript code which is powered by the Rhino engine under the Now Platform.

ARTICLE (4 OF 35)

Widget Components

Widgets include both mandatory and optional components.

HTML Template

The widget's HTML accepts and displays data.

- Renders the dynamic view a user sees in the browser using information from the model and controller
- Binds client script variables to markup
- Gathers data from user inputs like input text, radio buttons, and check boxes

HTML is mandatory.

Client Script

The widget's Client Script defines the AngularJS controller.

- Service Portal maps server data from *JavaScript* and *JSON* objects to client objects
- Processes data for rendering
- Passes data to the HTML template
- Passes user input and data to the server for processing

A Client Script is mandatory.

Server Script

The widget's Server Script works with record data, web services, and any other data available in ServiceNow server-side scripts.

- Sets the initial widget state
- Sends data to the widget's Client Script using the *data* object
- Runs server-side queries

A Server Script is mandatory.

Link Function

The Link Function uses AngularJS to directly manipulate the DOM.

A Link Function is optional.

Option Schema

The Option Schema allows a Service Portal Admin (*sp_admin* role) to configure a widget.

- Specifies widget parameters
- Allows admin users to define instance options for a widget instance
- Makes widgets flexible and reusable

An Option Schema is optional.

Angular Providers

An Angular Provider:

- Keeps widgets in sync when changing records or filters
- Shares context between widgets
- Maintains and persists state
- Creates reusable behaviors and UI components then injects them into multiple widgets

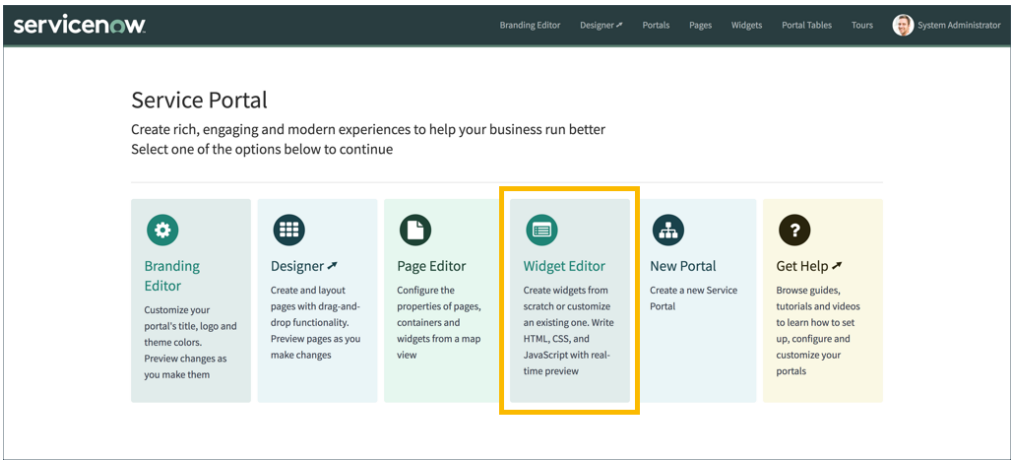
Angular Providers are optional.

ARTICLE (5 OF 35)

Widget Editor

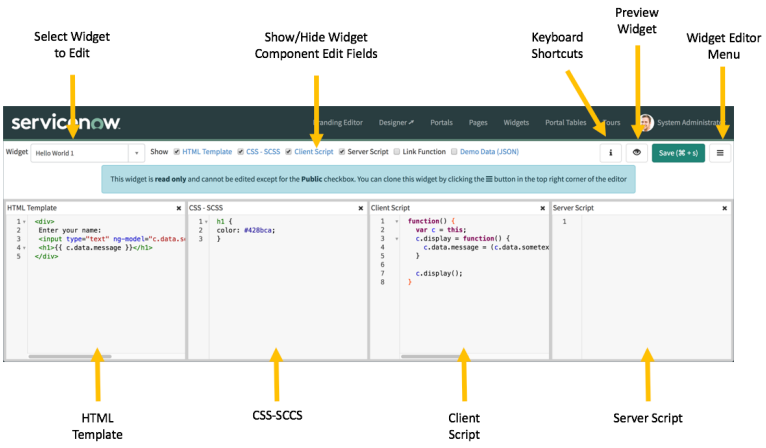
Widget Editor is the application for editing widget components.

Use the *All* menu to open **Service Portal > Service Portal Configuration**. Click the **Widget Editor** tile on the *Service Portal Configuration* page.



Use the *Show/Hide* widget component check boxes to choose which widget components to view and edit.

- Blue text, like the *Demo Data (JSON)* option in the example, indicates there is text in the widget component.
- Black text, like the *Server Script* option in the example, indicates there is no text in the widget component.

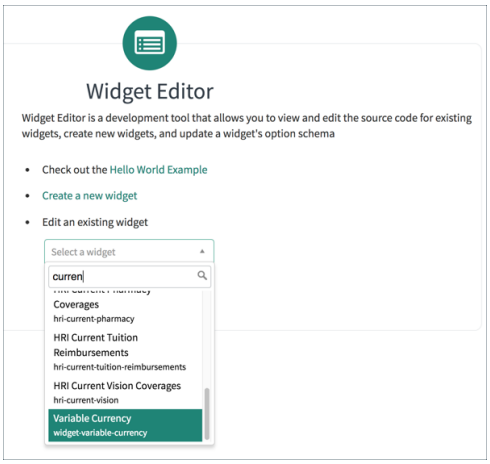


ARTICLE (6 OF 35)

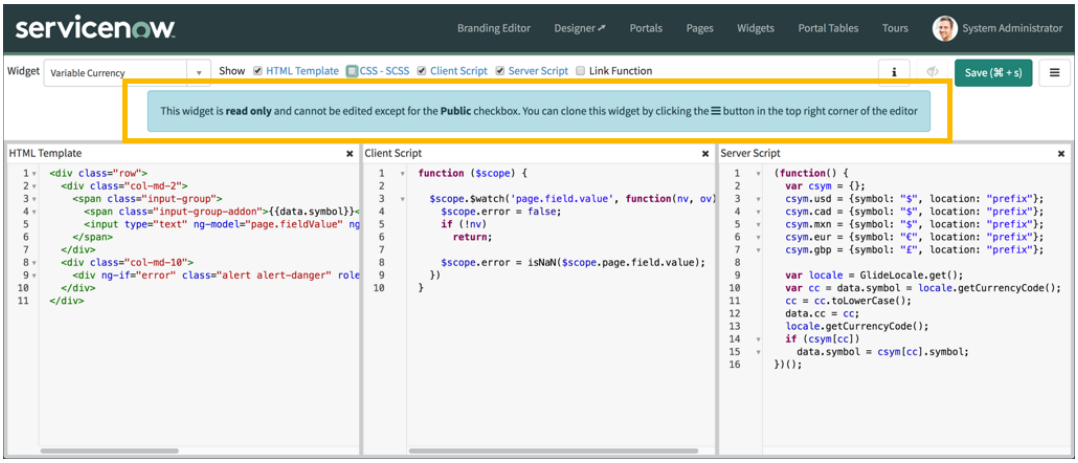
Cloning Widgets

When creating widgets, it can be time-saving to start from an existing widget instead of creating a widget from scratch. To protect existing widgets from accidental modifications, all baseline widgets are read-only. The process for creating an editable copy of a widget is known as cloning.

In Widget Editor, select a widget to be cloned from the *Edit an existing widget* menu.

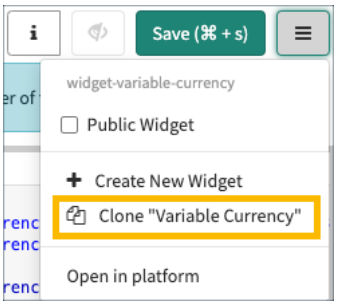


A read-only message in Widget Editor means the widget cannot be edited.



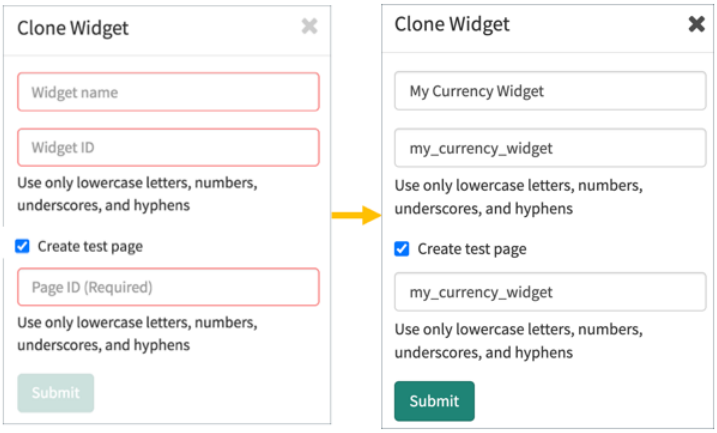
To clone a widget, click the **Widget Editor menu** button (

) in the Widget Editor header and select the **Clone "<Widget Name>"** menu item.

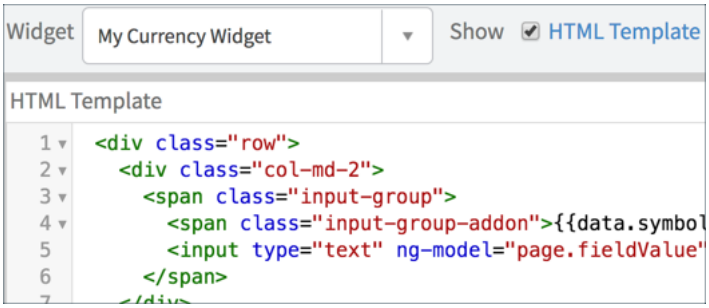


Enter a **Widget name**. The *Widget ID* field is automatically populated. The *Widget ID* field is editable. Use only lowercase letters, numbers, underscores, and hyphens.

Select the **Create test page** option to test the widget on a portal page without having to add the widget to a portal page manually. The *Page ID* field is automatically populated based on the *Widget name*. The *Page ID* is editable.



Use the *Widget* choice list to load the cloned widget into Widget Editor.



ARTICLE (7 OF 35)

Previewing Widgets

To test a widget's behavior without manually adding the widget to a Service Portal page:

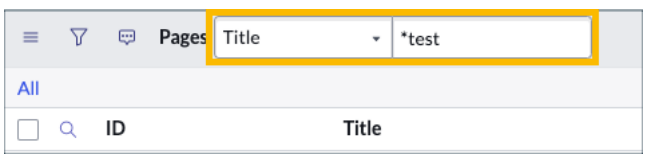
- Use the test page created when cloning the widget (creating a test page is optional)
- Use the *Preview* feature in Widget Editor

View the Test Page

To view a test page:

1. In the main ServiceNow browser window, use the *All* menu to open **Service Portal > Pages**.

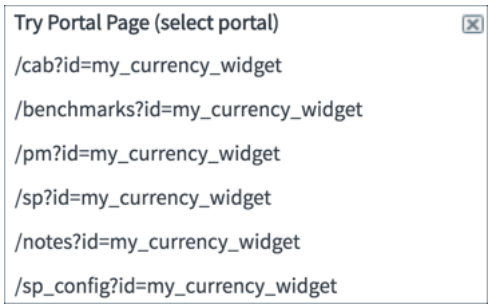
2. To see a list of all test pages, search the *Title* column for the string **test**.



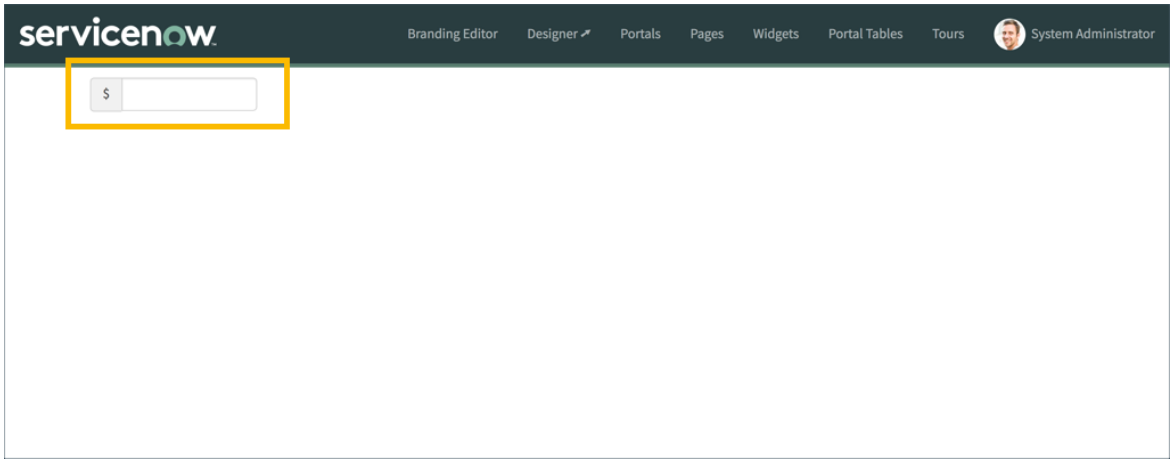
3. Open the record for the test page created when cloning the widget.

4. In the test page record, click the **Try It** button in the record header.

5. When prompted to select a portal, select the portal of your choice. The selected portal applies a header, footer, and theme to the test page.



6. Test the widget behavior.

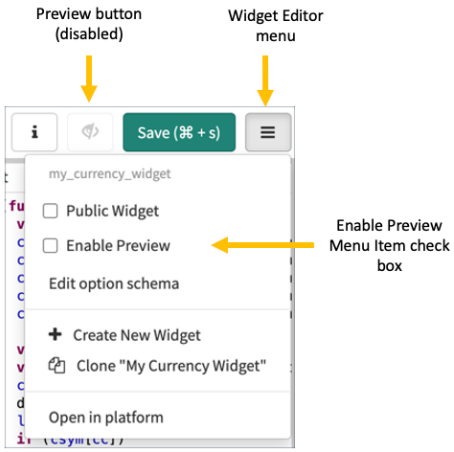


Notice the header on the test page. The header color is part of the portal theme. The theme was applied to the page when the portal was selected to use for testing.

Enabling Preview

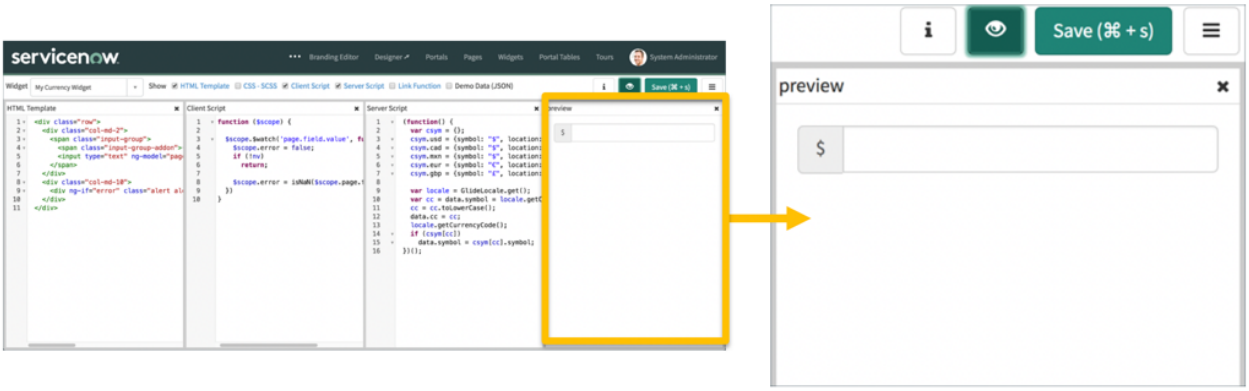
The Preview feature is not enabled by default. To enable Preview, click the **Widget Editor menu** button (

) in the Widget Editor header and select the **Enable Preview** menu item check box.



After enabling Preview, use the **Preview** button (

) to test a widget. A widget *Preview* pane opens in Widget Editor.



DEVELOPER TIP: When developing a widget, it is convenient to use the *Preview* pane to quickly test the widget's behavior. Always test the widget on a portal page before releasing a widget to production.

EXERCISE (8 OF 35)

Exercise: Widget Editor Basics

In this exercise, you will practice using Widget Editor by cloning, modifying, and previewing a baseline widget.

NOTE: If your PDI automatically opens App Engine Studio, you need to change the user role used to access the PDI. To complete the exercises, switch to the **Admin** user role (https://developer.servicenow.com/dev.do#!/guides/utah/developer-program/pdi-guide/managing-your-pdi#changing-your-instance-user-role).

Preparation

In this section of the exercise, you will prepare for cloning a widget in the *Global* scope.

1. In the main ServiceNow browser window, examine the *Application Scope* icon (

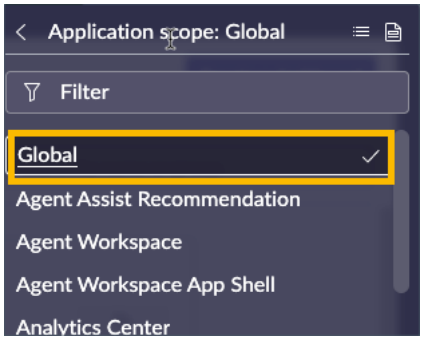
) in the banner. If there is a red circle around the icon, the scope is not *Global*.

2. Click the **Application Scope** icon.

3. If the Application scope: is *Global*, skip to the next section of this exercise.

4. IF the Application scope is not *Global*, click the **Application scope** field.

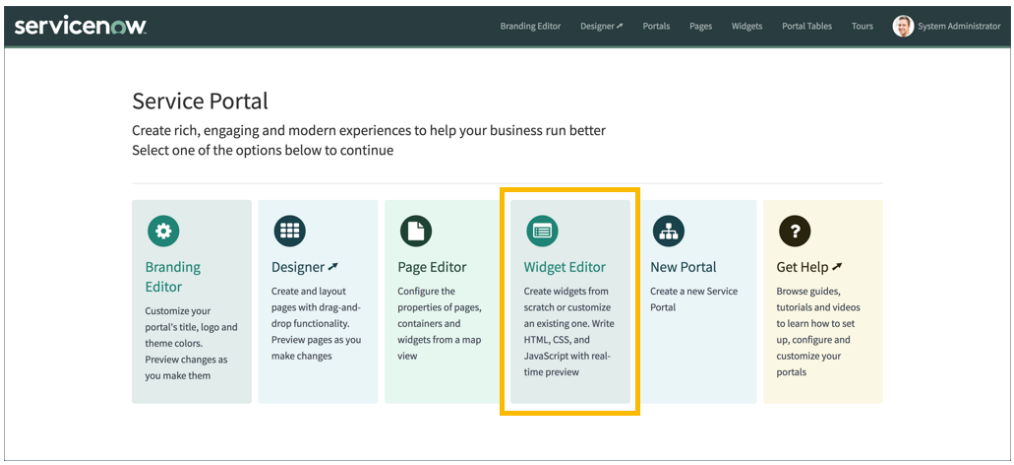
5. In the *Application scope* flyout, select **Global**.



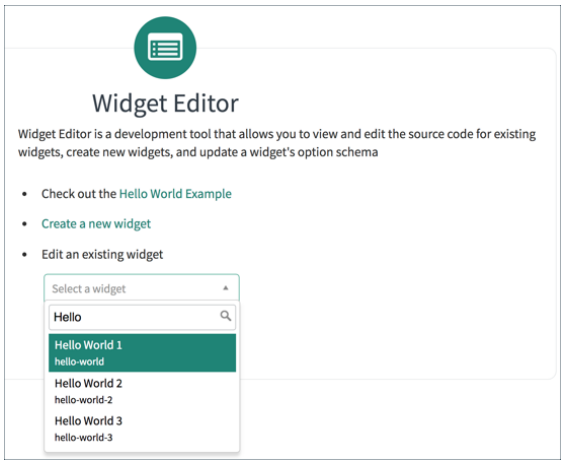
Clone the Hello World 1 Widget

1. In the main ServiceNow browser window, use the *All* menu to open **Service Portal > Service Portal Configuration**.

2. Click the **Widget Editor** tile on the *Service Portal Configuration* page.



3. Use the *Select a widget* field in the *Edit an existing widget* option to open the **Hello World 1** widget for editing.



4. Examine the Widget Editor header and notice the *Hello World 1* widget is read-only.

5. Clone the *Hello World 1* widget.

    1. Click the **Widget Editor menu** button (

) in the Widget Editor header and select the **Clone "Hello World 1"** menu item.
    2. Configure the clone:

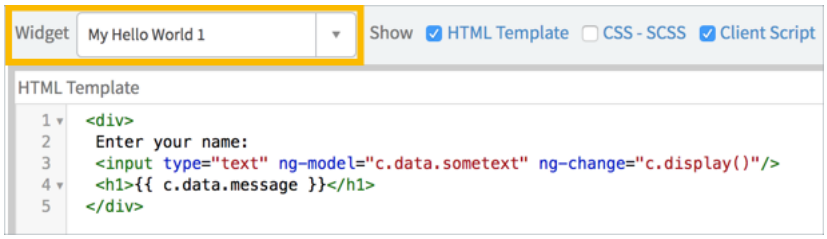
        Widget name: **My Hello World 1**

        Widget ID: **my_hello_world_1** (this value is automatically populated)

        Create test page: **Selected (checked)**

        Page ID: **my_hello_world_1** (this value is automatically populated)
    3. Click the **Submit** button.

6. Use the *Widget* field to select the **My Hello World 1** widget.



Enable Preview

1. Examine the *Preview* button in the Widget Editor header. If it looks like this, Preview is disabled:

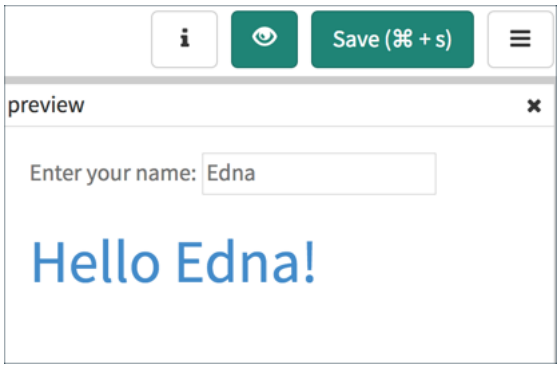
2. To enable Preview, click the **Widget Editor menu** button (

) in the Widget Editor header and select the **Enable Preview** menu item check box.

3. Examine the *Preview* button again. It should be *enabled*:

4. Click the **Preview** button to open the *Preview* pane.

5. Test the widget by entering **<your name>** in the input box. Notice that the widget is updated with every key you type.



Modify the Widget CSS

1. If it is not already selected, select the **CSS - SCSS** component in the Widget Editor header.

2. Examine the **CSS - SCSS** pane and note the color for *h1*.
3. Change the color to **#8bdb2e**.

4. Click the **Save** button.
5. Examine the *Preview* pane again. Notice the text color changed in the widget.
6. Change the *h1* color a second time. Specify a color you like.

ARTICLE (9 OF 35)

Widget Global Objects and Functions

Widget components include:

- Client Script

- Server Script

When a widget is instantiated, the global objects are created.

Server Script Global Objects

The Server Script global objects are:

Object Name	Description
data	Object containing a JSON object to send *to* the Client Script.
input	*Data* object received *from* the Client Script's controller.
options	The options used to invoke the widget on the server.

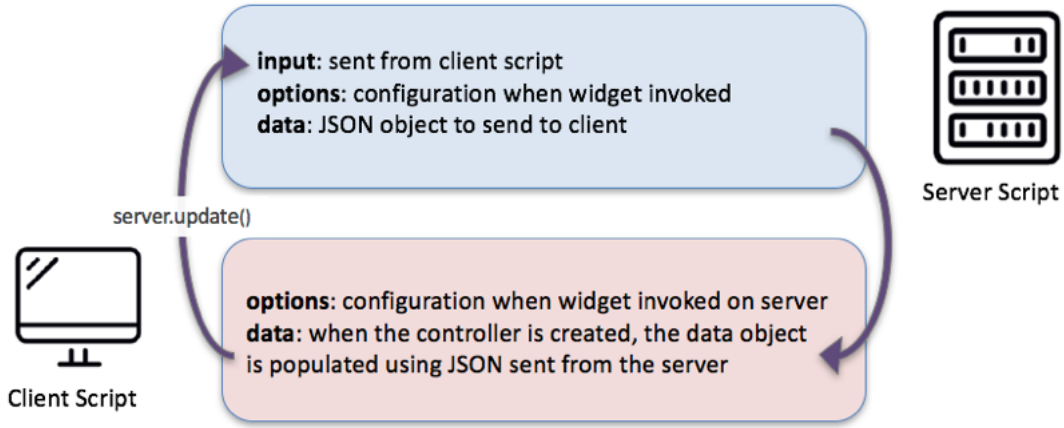
- When a widget instance is instantiated, *data* and *input* objects are initialized.
- The server script populates the *data* object.
- After the Server Script executes, the *data* object is JSON serialized and sent to the client controller.
- Use the *options* object to see what values were used to invoke the widget.

Client Script Global Objects

The Client Script global objects are:

Object Name	Description
data	The serialized *data* object from the Server Script.
options	The options used to invoke the widget on the server.

- When the controller is created, the client *data* object is populated by the serialized JSON object sent from the server (a copy of the server's *data* object)
- Use the *options* object to see what values were used to invoke the widget.
- The *options* object is read-only.



Client Script Global Functions

All Client Script global functions return a JavaScript promise. The Client Script global functions are:

Function Name	Description
this.server.get()	Calls the Server Script and passes custom input.
this.server.update()	Calls the server and posts *this.data* to the Server Script.
this.server.refresh()	Calls the server and automatically replaces the current options and data from the server response.

ServiceNow Client Scripting APIs

In addition to the Widget API, Service Portal supports some of the ServiceNow client-side APIs and methods (https://docs.servicenow.com/bundle/utah-servicenow-platform/page/build/service-portal/reference/widget-api-reference.html).

EXERCISE (10 OF 35)

Exercise: My Hello World Widget Scripts

In this exercise, you will examine and edit the Client and Server Scripts for the *My Hello World 1* widget. You will log the properties of the *data* object to understand how the HTML template, Client Script, and Server Script work when the *data* object is updated.

Preparation

1. If not still open in Widget Editor, open the *My Hello World 1* widget for editing.
    1. In the main ServiceNow browser window, use the *All* menu to open **Service Portal > Service Portal Configuration**.
    2. Click the **Widget Editor** tile on the Service Portal Configuration page.
    3. Use the *Select a widget* field in the *Edit an existing widget* option to open the **My Hello World 1** widget for editing.
2. Click the **Preview** button (
    
) to remind yourself of the widget's behavior.
    1. Notice the placeholder text in the *Enter your name* field.
    2. Type a new value in the *Enter your name* field. Do you have to press the **<enter>/<return>** key for the widget text to update as you type in the field?
    3. Delete the text in the *Enter your name* field. Does the placeholder text reappear?

My Hello World 1 Server Script

1. If not already visible in Widget Editor, open the **Server Script** pane.

2. Is there a Server Script?

My Hello World 1 Client Script

1. If not already visible in Widget Editor, open the **Client Script** pane.

2. Add the *property* variable to the Client Script. The new script statement is in bold.

```
function() {
   var c = this;
   c.display = function() {
        var property = '';
        c.data.message = (c.data.sometext) ? 'Hello ' +
     c.data.sometext + '!' : '';
   }
   c.display();
}
```

3. Copy the *for* loop script and paste it into the Client Script at the location shown.

```
for(property in c.data){
                console.log('c.data.' + property + ": " + c.data[property]);
        }
```

Client Script

```
1  function() {
2    var c = this;
3    c.display = function() {
4      var property = '';
5      c.data.message = (c.data.sometext) ? 'Hello ' + c.data.sometext + '!' : '';
6
7    }
8    c.display();
9  }
```

DEVELOPER TIP: To automatically apply indentation to code, click and drag to highlight the code, then press **<Shift> + <tab>** on the keyboard.

4. Click the **Save** button.

5. In Widget Editor, open the HTML Template pane.

6. Examine the HTML to make sure you understand how the HTML works.

   - The *c.data.sometext* value is updated when a user types in the text input field.
   - When the value in the *input* field changes, the *c.display* function in the Client Script is called.
   - The Client Script updates the *c.data.message* value which is displayed in the *<h1>* part of the HTML Template.

Test the My Hello World 1 Widget

1. Enable the Developer/JavaScript console in your browser using whatever strategy is appropriate for your browser. For example, in some versions of Chrome, open *View > Developer > JavaScript Console*.

2. Disable the *Preview* pane in Widget Editor then re-enable it to force the widget to reload.

3. Examine the console. You should see two log messages for properties on the *c.data* object.

   QUESTION: With no Server Script and without the value being set in the Client Script, how was the value of *c.data.sometext* set?
   ANSWER: The *Demo Data* contains a value for *data.sometext*. When the widget is loaded, the Demo Data is passed from the server to the Client Script.

4. Type **<your name>** in the input box in the *Preview* pane. Watch the console to see the *c.data* object property values update in real time.

5. Open the My *Hello World 1* test page.

   1. In the main ServiceNow browser window, use the *All* menu to open **Service Portal > Pages**.
   2. Open the **my_hello_world_1** page for editing.
   3. Click the **Try It** button.
   4. Select the **/sp/?id=my_hello_world_1** portal.

6. Enable the Developer/JavaScript console for the test page browser tab.

7. Reload the test page.

8. Examine the console and locate *c.data.message*. Does the message property have a value?

9. Type a value in input box. Watch the console as you type to see the values of *c.data.message* and *c.data.sometext*.

   QUESTION: Why did the *c.data.sometext* property have a value when you tested the *My Hello World 1* widget in the *Preview* pane but did not have a value when you tested using the *test* page?
   ANSWER: The Demo Data is provided as a convenience for testing widgets in the *Preview* pane. The Demo Data is not passed from the server to the client in the runtime environment.

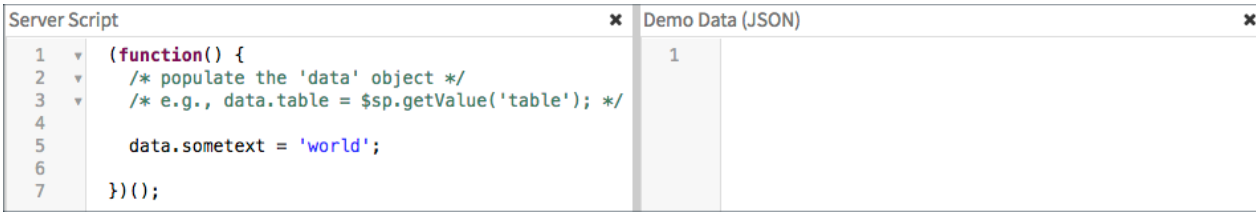
Challenge

Your Challenge is to modify the widget logic to set the initial value of *data.sometext* in the Server Script.

- Test the case in which both the Server Script AND the Demo Data set the *data.sometext* value so you can determine which takes precedence.
- Is the initial value of *demo.sometext* the same in both the *Preview* pane and the test page?

CHALLENGE SOLUTION:

Set the *data.sometext* initial value in *My Hello World 1* widget Server Script.



EXERCISE (11 OF 35)

Exercise: Fork Repository and Import Application for the Creating Custom Widgets Module

ServiceNow uses GitHub to provide application repositories to copy and use with the Developer Site learning content. The repositories contain tags, which are fixed sets of application files, to start you with a partially built application. By copying and importing a ServiceNow-provided repository into your Personal Developer Instance (PDI), you get all the files needed for the hands-on exercises in the modules.

> **NOTE**: See the GitHub Guide (/dev.do#!/guide/utah/now-platform/github-guide/github-and-the-developer-site-training-guide-introduction) for more information on how ServiceNow uses GitHub with the Developer Program learning content and to see a video on how to fork a repository and import an application.

In this exercise, you will:

1. Fork the ServiceNow repository to your GitHub account.
2. Import the application into your PDI from your fork of the repository.

> **IMPORTANT**: If you have already forked and imported the repository, you can proceed to the next exercise, where you will create a branch from a tag to load the application files to your PDI. The *CreateNotes* application files are needed to complete the module.

Fork the Repository

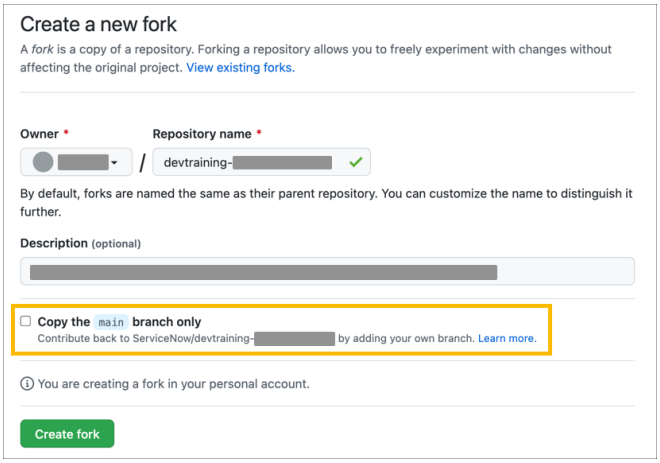
In this section of the exercise, you will create a personal fork of the application repository to use with Developer Site learning content.

1. In a web browser, open github.com (https://github.com/).
2. If you have a GitHub account, sign in. If not, sign up for a new account.

3. Once signed in, open the CreateNotes repository (https://github.com/ServiceNow/devtraining-createnotes-utah).
4. Click the **Fork** button (

   Fork

) to create a copy of the repository in your GitHub account.
5. On the *Create a new fork* page, deselect the **Copy the main branch only** option.



6. Select your personal GitHub account as the fork *Owner*, then click the **Create fork** button.

7. Verify the URL for your fork of the repository is similar to: *<YourGitHubUsername>/devtraining-application-release*.

8. Copy the forked repository's URL.

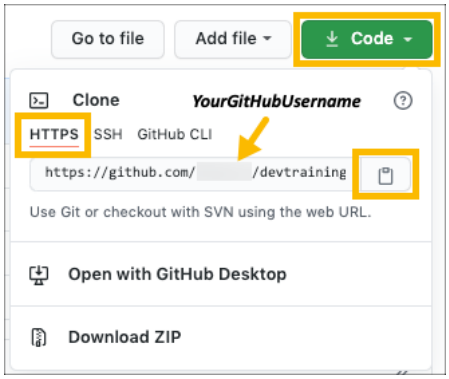
    1. Click the **Code** button.

    2. Make sure the URL contains your GitHub username, not ServiceNow.

    3. Make sure **HTTPS** is selected. If not, select the **HTTPS** tab in the *Clone* flyout.

    4. Click the **Copy to clipboard** button (

).



NOTE: You will use the copied URL to configure the connection to your forked repository in the next section.

Import the Application from the Forked Repository

In this section of the exercise, you will import the application repository into ServiceNow. As part of the process, you will first create a *Credential* record for your GitHub account, then you will use Studio to import the application repository into your PDI.

1. Log in to your PDI as the **admin** user. If you do not have a PDI, open the ServiceNow Developer Site (https://developer.servicenow.com) to obtain a **Utah** PDI.

   NOTE: See the Personal Developer Instance (PDI) Guide (/dev.do#!/guide/utah/now-platform/pdi-guide/personal-developer-instance-guide-introduction) for instructions on how to obtain a PDI.

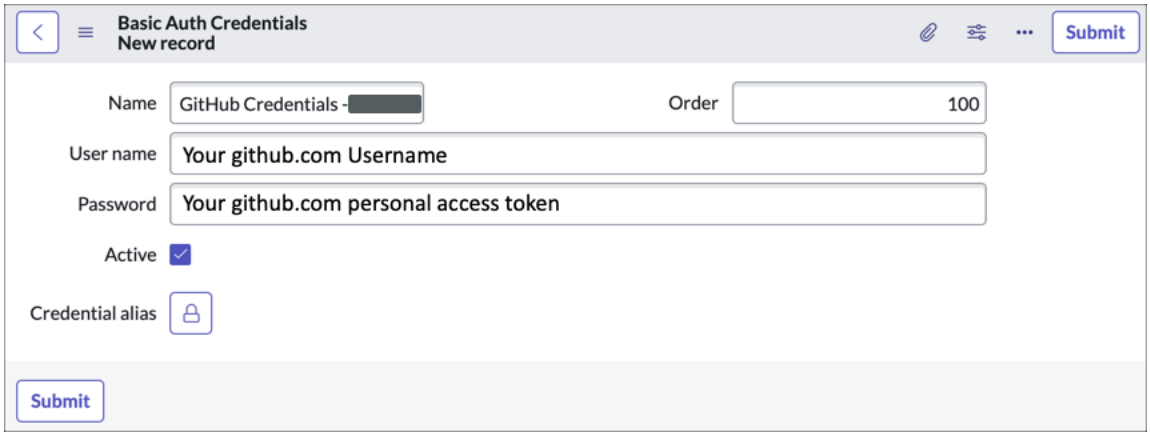
2. Create a *Credential* record for the GitHub connection.

   IMPORTANT: Credential records only need to be created once. If you have already created a credential record in another exercise, please skip this step.

   1. Use the *All* menu to open **Connections & Credentials > Credential**.
   2. Click the **New** button.
   3. In the *What type of Credentials would you like to create?* list, click the **Basic Auth Credentials** link.

4. Configure the *Credential* record.

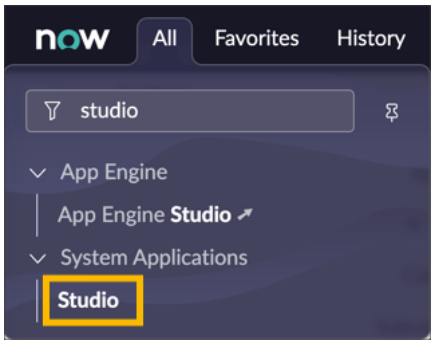
   Name: **GitHub Credentials - <Your github.com Username>**
   User name: **<Your github.com Username>**
   Password: **<Your github.com personal access token>**



   IMPORTANT: GitHub requires personal access tokens to access repositories from other platforms, like ServiceNow. **A personal access token is used in place of a password when authenticating.** See the Authenticating to GitHub (/dev.do#!/guides/utah/developer-program/github-guide/using-servicenow-provided-application-repositories#authenticating-to-github) section of the *GitHub Guide* for instructions on how to create a GitHub personal access token.

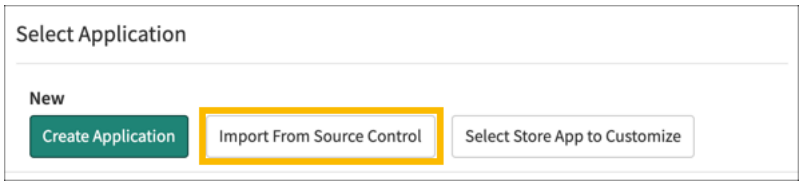
5. Click the **Submit** button.

3. Use the *All* menu to open **System Applications > Studio**.



4. Studio opens in a new browser tab.

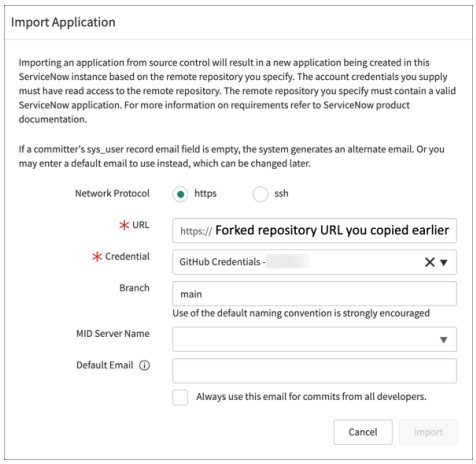
5. In the *Select Application* dialog, click the **Import From Source Control** button.



6. In the *Import Application* dialog, configure the connection to the forked repository.

   URL: **<URL you copied for your forked version of the repository>**
   Credential: **GitHub Credentials - <Your github.com Username>**
   Branch: **main**

   NOTE: When you change the *Branch* value to **main**, an information message informs you that *Use of the default naming convention is strongly encouraged*. The value in the *Branch* field must exist in the repository. The Developer Site training repositories all have a *main* branch, which should be used in place of the default value.



7. Click the **Import** button.

8. When the application import is complete, click the **Select Application** button.

   NOTE: If the connection fails, you may have entered the ServiceNow repository URL in the URL field instead of the forked repository URL, or you may have enabled two-factor authentication on your GitHub account. See Troubleshooting GitHub Issues (/dev.do#!/guide/utah/now-platform/github-guide/troubleshooting-github-issues) for

instructions on how to troubleshoot the connection.

9. In the *Select Application* dialog, click the application to open it for editing in Studio.

   IMPORTANT: You will not see any application files in Studio until you successfully create a branch from a tag in the next exercise.

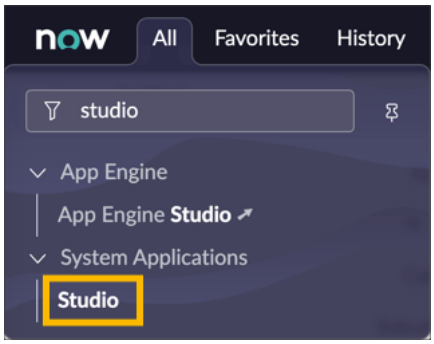
EXERCISE (12 OF 35)

Exercise: Create a Branch for Creating Custom Widgets

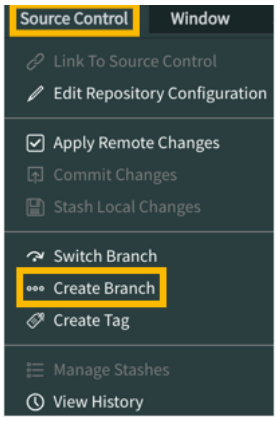
In this exercise, you will create a branch of the *CreateNotes* application for the *Creating Custom Widgets* module that includes the application files used in the module.

NOTE: Before you begin this exercise, you need to have forked and imported the *CreateNotes* repository as described in *Exercise: Fork Repository and Import Application for the Creating Custom Widgets Module*.

1. If the *CreateNotes* application is not already open from the previous exercise, open it now.
   1. In the main ServiceNow browser window, use the *All* menu to open **System Applications > Studio**.



   2. In the *Select Application* dialog, click the **CreateNotes** application.
2. In Studio, open the **Source Control** menu and select the **Create Branch** menu item.



3. Configure the branch.

   Branch Name: **CreateWidgetsModule**
   Create from Tag: **LoadForCreateWidgetsModule**

4. Click the **Create Branch** button.

5. Click the **Close** button.

6. To load the application files included in the tag, return to the main ServiceNow browser tab (not Studio) and click the browser's reload button to refresh the page.

> **NOTE**: If branch creation fails, you may have entered the ServiceNow repository URL in the URL field instead of the forked repository URL, or you may have enabled two-factor authentication on your GitHub account. See the Troubleshooting GitHub Issues (/dev.do#!/guide/utah/now-platform/github-guide/troubleshooting-github-issues) section of the *GitHub Guide* for instructions on how to troubleshoot GitHub connection issues.

ARTICLE (13 OF 35)

Studio and Service Portal

For scoped applications, use Studio to create application files for Service Portal:

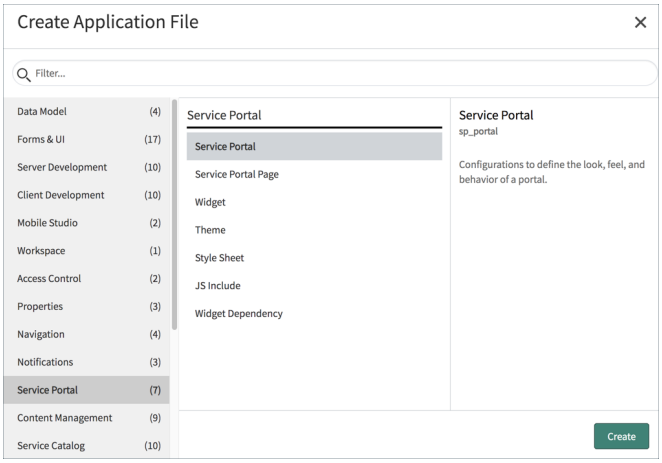
- Service Portal
- Service Portal Page
- Widget
- Theme
- Style Sheet
- JS Include
- Widget Dependency

The procedure for adding files to an application in Studio is the same regardless of application file type:

1. Click the **Create Application File** button.
2. Choose the file type, such as **Widget** OR click the **Service Portal** category, then click a file type.

3. Configure the new file.

Depending on the file type, a new tab might open in Studio or a new browser window might open.

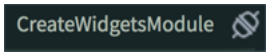


EXERCISE (14 OF 35)

Exercise: Explore the CreateNotes Application

In this exercise, you will explore the *CreateNotes* application you imported from source control. You will create four Notes records.

Preparation

1. If the *CreateNotes* application is not still open in Studio, open it now.
    1. In the main ServiceNow browser window, use the *All* menu to open **System Applications > Studio**.
    2. In the *Select Application* dialog, click **CreateNotes**.
2. If the *CreateWidgetsModule* branch is not open, use the **Source Control > Switch Branch** menu item to switch to the *CreateWidgetsModule* branch.
3. Examine the Studio status bar to make sure the *CreateWidgetsModule* branch is loaded (bottom of the screen, right-hand side).
   

Explore the CreateNotes Application Files

1. In Studio, use the Application Explorer to open **Data Model > Tables > Note**.
2. Examine the *Note* table's columns and notice the field data types. In particular, look for:
    - *Number*
    - *User*
    - *Title*
    - *Note*
3. Use the Application Explorer to open **Forms & UI > Forms > Note [Default view]** to see the form layout.
4. Use the Application Explorer to open **Forms & UI > List Layouts > Note [Default view]** to see the columns in the list layout.

5. In the *Navigation* category in the Application Explorer, locate the *CreateNotes* application menu: **Navigation > Application Menus > CreateNotes**. Also locate the *Notes* module: **Navigation > Modules > Notes**.

Create CreateNotes Records

1. Switch to the main ServiceNow browser window, not Studio. Reload the page, then use the *All* menu to open **CreateNotes > Notes**.
2. Click the **New** button.
3. Configure the record fields:

   Number: **(this value is automatically set)**

   User: **System Administrator**

   Title: **Note 1**

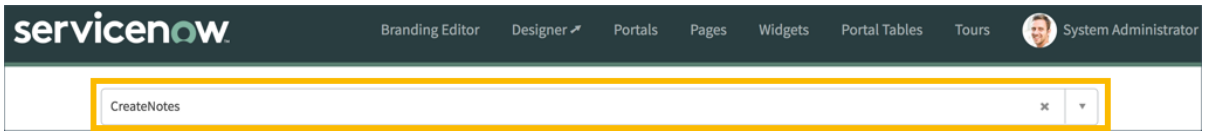
   Note: **note 1**
4. Click the **Submit** button.
5. Create two more *Notes* records. Set the *User* field value to **System Administrator** for both records. Use the values of your choice for the remaining fields.
6. Create one more *Notes* record. Set the *User* field value to **Beth Anglin**. Use the values of your choice for the remaining fields.

Explore the CreateNotes Service Portal

1. In Studio, examine the *Service Portal* category in the Application Explorer. Notice a Service Portal and a Service Portal Page have already been created.

2. In the Application Explorer, open **Service Portal > Service Portals > CreateNotes**. Page Editor opens in a new browser window.

3. If not already selected, select **CreateNotes** from the Portal list.



4. If not already selected, click the **CreateNotes** node in the tree.

5. Notice the values in these fields:

   - *Homepage*
   - *Logo*
   - *URL suffix*

QUESTION: What is the purpose of the *URL suffix* field?
ANSWER: Append the *URL suffix* field value to the end of an instance URL to open the portal. For example, if the *URL suffix* field value is *my_portal*:

```
https://<your_instance>.service-now.com/my_portal
```

QUESTION: What is the purpose of the *Homepage* field?
ANSWER: The *Homepage* field specifies the page a user sees when the portal is loaded.

6. Close the Page Editor window and return to Studio.

Explore the CreateNotes Home Service Portal Page

1. In Studio, use the Application Explorer to open **Service Portal > Service Portal Pages > CreateNotes Home**. Page Editor opens in a new browser window.

2. Click the **notes_home** box in the tree.

3. Examine the value in the *ID* field. Was this value referenced in the portal definition?

4. Close the Page Editor window and return to Studio.

EXERCISE (15 OF 35)

Exercise: Create Two Widgets

In this exercise, you will create two widgets then add the widgets to the *CreateNotes* application's *Home* portal page.

Create the Notes List Widget

1. If the *CreateNotes* application is not open in Studio from the last exercise, open it now.
    1. In the main ServiceNow browser window use the *All* menu to open **System Applications > Studio**.
    2. In the *Select Application* dialog, click the **CreateNotes** application.

2. Create a widget.

1. In Studio, click the **Create Application File** button.
2. In the *Filter...* field enter the text **widget** OR select **Service Portal** from the categories in the left hand pane.
3. Select **Widget** in the middle pane as the file type, then click the **Create** button. Widget Editor opens in a new window.

3. Click the **Create a new widget** link.

4. Configure the widget:

   Widget Name: **Notes List**
   Widget ID: **notes_list** (this value is automatically populated)
   Create test page: **Not Selected (unchecked)**

5. Click the **Submit** button.

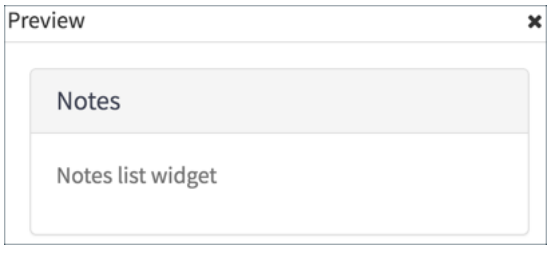
6. If the *HTML Template* pane is not visible, select **HTML Template** in the Widget Editor header.

7. Replace the contents of the *HTML Template* pane with this HTML:

```
<div class="panel panel-default">
          <div class="panel-heading clearfix">
            <h3 class="panel-title pull-left">
              ${Notes}
            </h3>
          </div>
          <div class="panel-body">
            <p>
              Notes list widget
            </p>
          </div>
        </div>
```

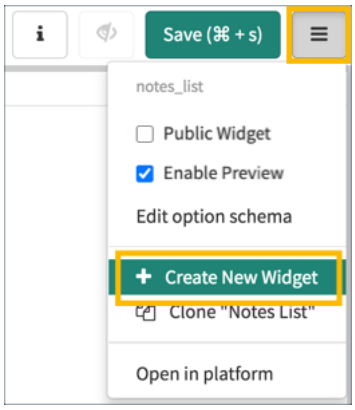
8. Click the **Save** button.

9. Preview the *Notes List* widget using the *Preview* pane in Widget Editor. If Preview is not enabled, enable it using the same procedure as in earlier exercises.



Create the Notes Body Widget

1. Create a widget in Widget Editor by clicking the **Widget Editor** menu and selecting the **Create New Widget** menu item.



2. Configure the widget:

   Widget Name: **Notes Body**
   Widget ID: **notes_body** (this value is automatically populated)
   Create test page: **Not Selected (unchecked)**

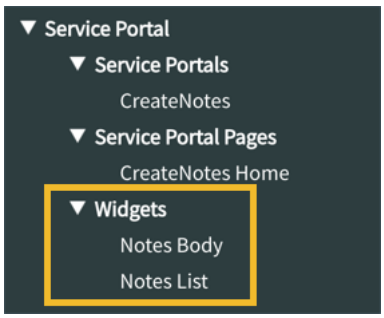
3. Click the **Submit** button.

4. Do not add any logic to the *Notes Body* widget.

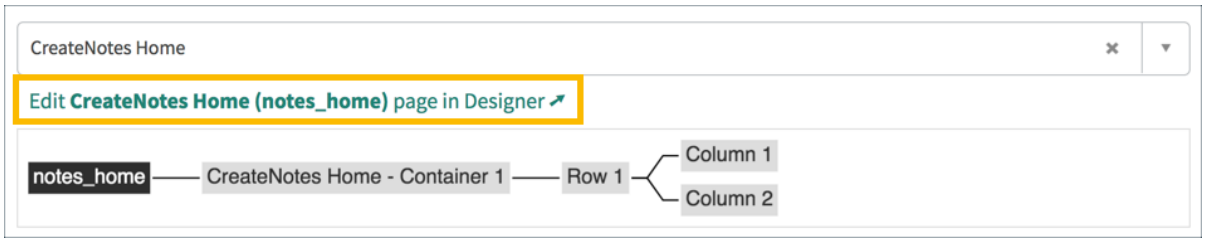
5. Click the **Save** button, then close Widget Editor.

Add the New Widgets to the CreateNotes Home Service Portal Page

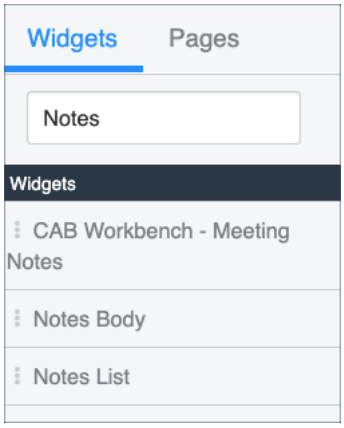
1. Return to Studio and examine the **Service Portal > Widgets** category in the Application Explorer. Look for the two new widgets. You may need to reload Studio to see the Widgets in the Application Explorer.



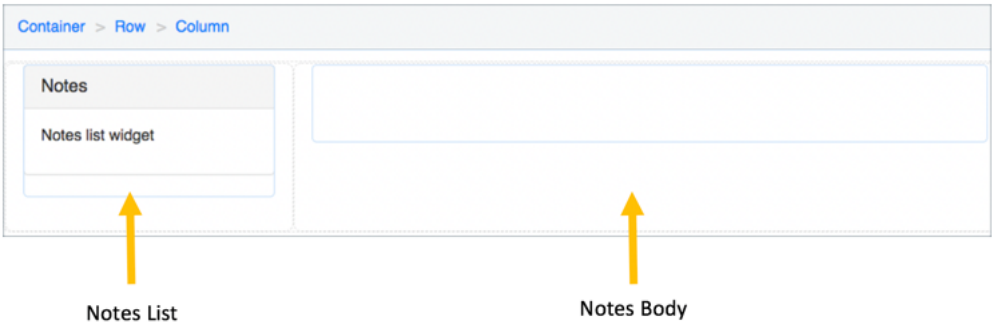
2. In the Application Explorer, open **Service Portal > Service Portal Pages > CreateNotes Home**.
3. Click the **Edit CreateNotes Home (notes_home) page in Designer** link.



4. In Designer, use the *Filter Widget* field to locate the *Notes List* and *Notes Body* widgets.

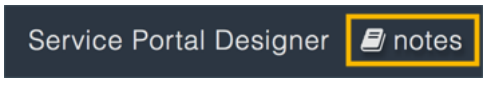


5. Add the Notes widgets to the container.
6. Drag the *Notes List* widget to the container column on the left.
7. Drag the *Notes Body* widget to the container column on the right.



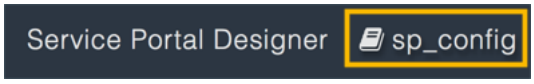
Test the CreateNotes Home Service Portal Page

1. Examine the Designer header to determine the active portal. If the *CreateNotes* portal is the active portal, you will see **notes** in the header:

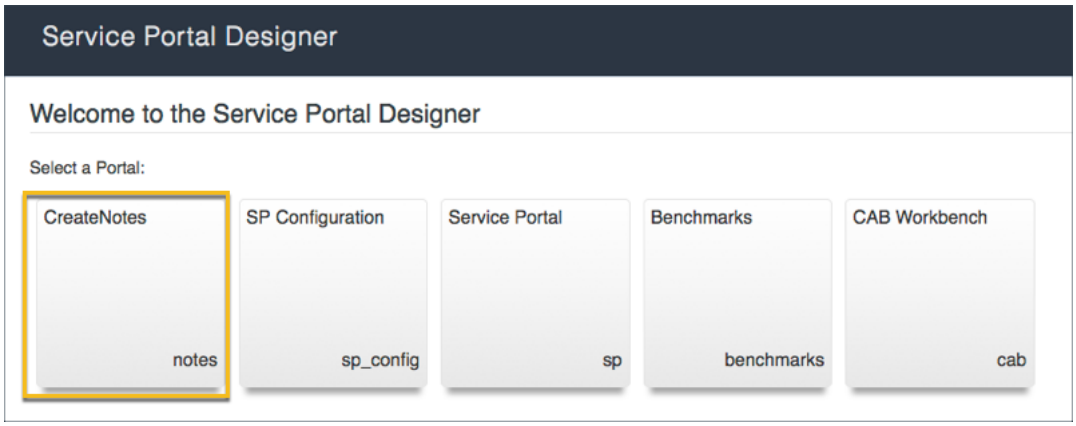


2. If the *CreateNotes* portal is not the active portal, make it the active portal.

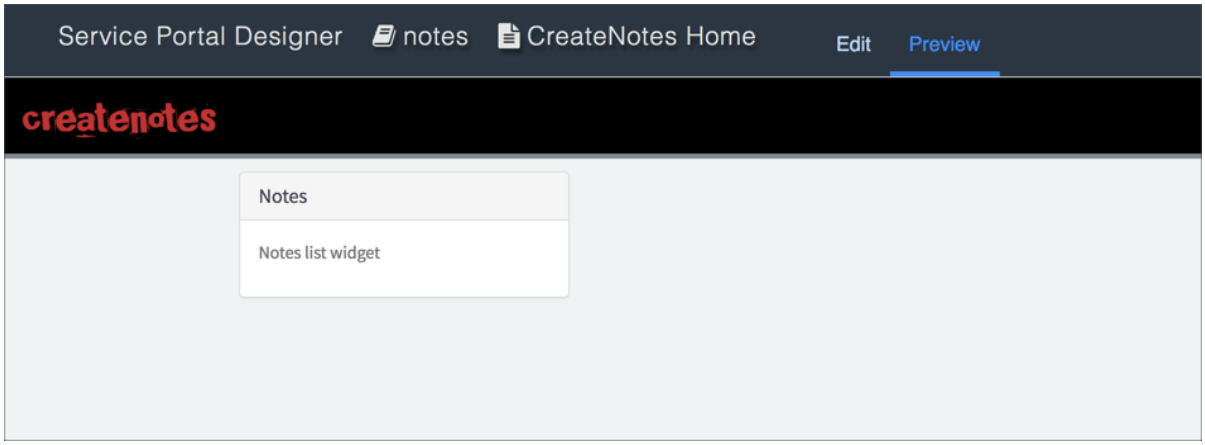
    1. Click the active portal name in Designer.



    2. When prompted to select a portal, select the **CreateNotes** tile.



3. Preview the *CreateNotes Home* portal page by switching to the **Preview** tab in Designer.



4. Notice the logo and header.

   QUESTION: Why do you not see the *Notes Body* widget in the *Preview* pane?
   ANSWER: The *Notes Body* widget has no HTML to render yet. There is no default HTML Template.

5. Click the **Edit** tab to return to editing the portal widgets.

ARTICLE (16 OF 35)

Widget API

Service Portal has an API known as the *Widget API*. The *Widget API* contains classes for both client-side and server-side scripting.

Client-side API

The client-side *Widget API* classes are:

- ***spUtil*** (https://developer.servicenow.com/app.do#!/api_doc?v=utah&id=spUtilAPI): Contains utility methods to perform common functions in a Service Portal widget client script. Access the methods from this class using *spUtil*. For example, *spUtil.addErrorMessage()*.
  - ***addErrorMessage()***: display an error message
  - ***addInfoMessage()***: display an informational message

    - ***addTrivialMessage()***: display a message which automatically disappears after a short period of time
    - ***createUid()***: create a unique ID
    - ***format()***: used to build strings from variables (alternative to concatenation)
    - ***get()***: gets a widget model by *ID* or *sys_id*
    - ***getHost()***: gets complete host domain
    - ***getHeaders()***: retrieves all headers to be used for API calls
    - ***getPreference()***: executes callback with user preference response
    - ***getURL()***: gets current service portal URL
    - ***isMobile()***: returns true if current client is a mobile device
    - ***parseAttributes()***: returns the attributes for a field in csv format
    - ***recordWatch()***: watches for updates to a table or filter and returns the value from a callback function
    - ***scrollTo()***: scrolls to element with specified selector, over specified period of time.
    - ***setBreadCrumb()***: updates the header breadcrumbs
    - ***setPreference()***: sets a user preference
    - ***setSearchPage()***: sets the search page
    - ***refresh()***: calls the server and replaces the current *options* and *data* objects with the server response
    - ***update()***: updates the *data* object on the server within a given scope
- ***spModal*** (https://developer.servicenow.com/app.do#!/api_doc?v=utah&id=SPModal-API): Methods provide an alternative way to show alerts, prompts, and confirmation dialogs. Access the methods from this class using *spModal*. For example, *spModal.alert()*.
    - ***alert()***: displays an alert
    - ***confirm()***: displays a confirmation message
    - ***open()***: opens a modal using the specified options
    - ***prompt()***: displays a prompt for user input
- ***spAriaUtil*** (https://developer.servicenow.com/app.do#!/api_doc?v=utah&id=spAriaUtil-API): Uses an AngularJS service to show messages on a screen reader. Access the method from this class using *spAriaUtil*. For example, *spAriaUtil.sendLiveMessage()*.
    - ***sendLiveMessage()***: announce a message to a screen reader
- ***spContextManager*** (https://developer.servicenow.com/dev.do#!/reference/api/utah/client/spContextManagerAPI): Makes data from a Service Portal widget available to other applications and services in a Service Portal page. For example, pass widget data to Agent Chat when it opens in a Service Portal page. Access the methods from this class using *spContextManager*. For example, *spContextManager.getContext()*.
    - ***addContext()***: initializes a key and adds widget data as the value
    - ***getContext()***: returns each key and associated data object defined by any widget on a page
    - ***getContextForKey()***: returns the widget data associated with a key
    - ***updateContextForKey()***: sends data to an existing key

For the complete API documentation, including method arguments and return values, follow the links to the API classes.

Server-side API

The server-side *Widget API* classes includes:

- ***GlideSPScriptable*** (https://developer.servicenow.com/dev.do#!/reference/api/utah/server/no-namespace/c_GlideSPScriptableScopedAPI): Methods for use in Service Portal widget Server Scripts. Access the GlideSPScriptable methods using the global *$sp* object. For example, *$sp.canRead()*.
    - ***canReadRecord()***: returns true if the user can read the specified *GlideRecord*
    - ***canSeePage()***: returns true if the user can view the specified page
    - ***getCatalogItem()***: returns a model and view model for a *sc_cat_item* or *sc_cat_item_guide*
    - ***getDisplayValue()***: returns the display value of the specified field from either the widget's *sp_instance* or *sp_portal* record
    - ***getField()***: returns information about the specified field in a *GlideRecord*
    - ***getFields()***: checks the specified list of field names, and returns an array of valid field names

- ***getFieldsObject()***: checks the specified list of field names and returns an object of valid field names
- ***getForm()***: returns the form
- ***getKBCategoryArticles()***: returns Knowledge Base articles in the specified category and its subcategories
- ***getKBCount()***: returns the number of articles in the specified Knowledge Base
- ***getListColumns()***: returns a list of the specified table's columns in the specified view
- ***getMenuHREF()***: returns the *?id=* portion of the URL based on the *sp_menu* type
- ***getMenuItems()***: returns an array of menu items for the specified instance
- ***getParameter()***: returns the value of the specified parameter
- ***getPortalRecord()***: returns the portal's *GlideRecord*
- ***getRecord()***: returns the current portal context
- ***getRecordDisplayValues()***: copies display values for the specified fields into the data parameter
- ***getRecordElements()***: for the specified fields, copies the element's name, display value, and value into the data parameter
- ***getRecordValues()***: copies values for the specified field names from the *GlideRecord* into the data parameter
- ***getRecordVariables()***: returns Service Catalog variables associated with a record
- ***getRecordVariablesArray()***: returns an array of Service Catalog variables associated with a record
- ***getStream()***: gets the activity stream for the specified record. This method works on tables which extend the *Task* table
- ***getUserInitials()***: returns the user's initials
- ***getValue()***: returns the named value of the JSON request, instance, or portal
- ***getValues()***: copies values from the request or instance to the data parameter
- ***getWidget()***: gets a widget by id or *sys_id*, executes that widget's server script using the provided options, then returns the widget model
- ***mapUrlToSPUrl()***: transforms a URL requesting a list or form in the platform UI into the URL of the corresponding *id=list* or *id=form* Service Portal page

For the complete API documentation, including additional classes, method arguments and return values, follow the links to the API class or go to the Widget API page on the ServiceNow docs site (https://docs.servicenow.com/bundle/utah-servicenow-platform/page/build/service-portal/reference/widget-api-reference.html).

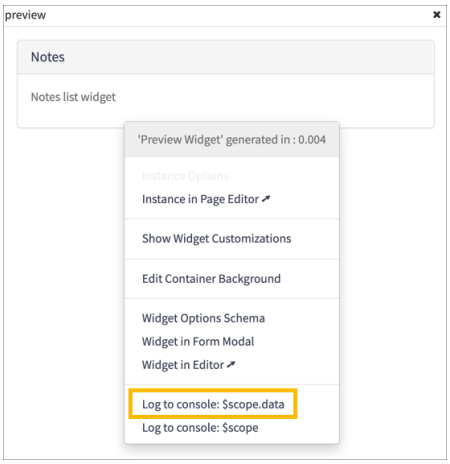
ARTICLE (17 OF 35)

Debugging Widgets

In this module you have already used the *console.log()* method to see debugging information for a widget.

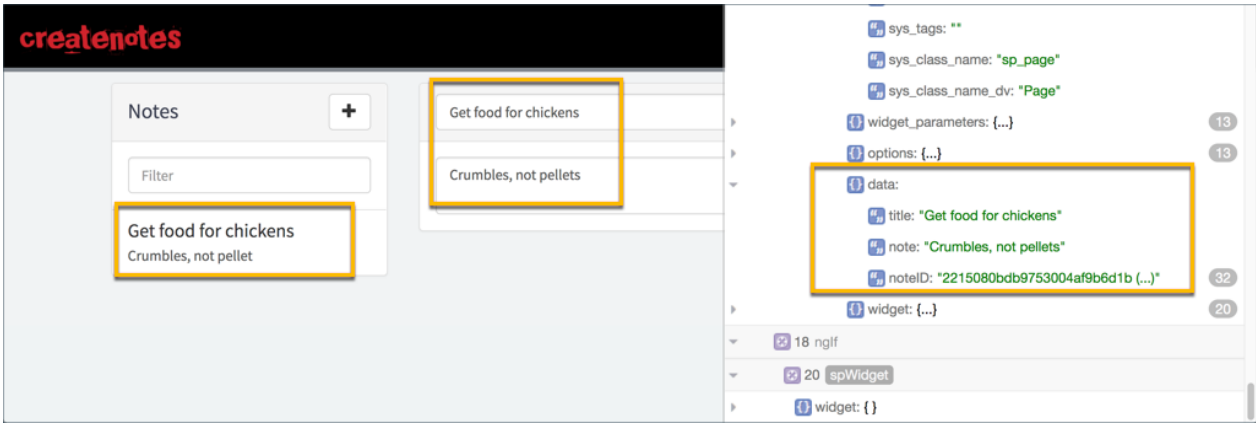
```
for(var property in c.data){
                console.log('c.data.' + property + ": " + c.data[property]);
        }
```

Instead of adding *console.log()* or *alert()* messages into scripts, use the *Log to console: $scope.data* or *Log to console: $scope* Widget Context menu items to write object properties and values to the console. To access the ^ menu, **<ctrl> + <right-click>** the widget in the Preview pane or portal page.



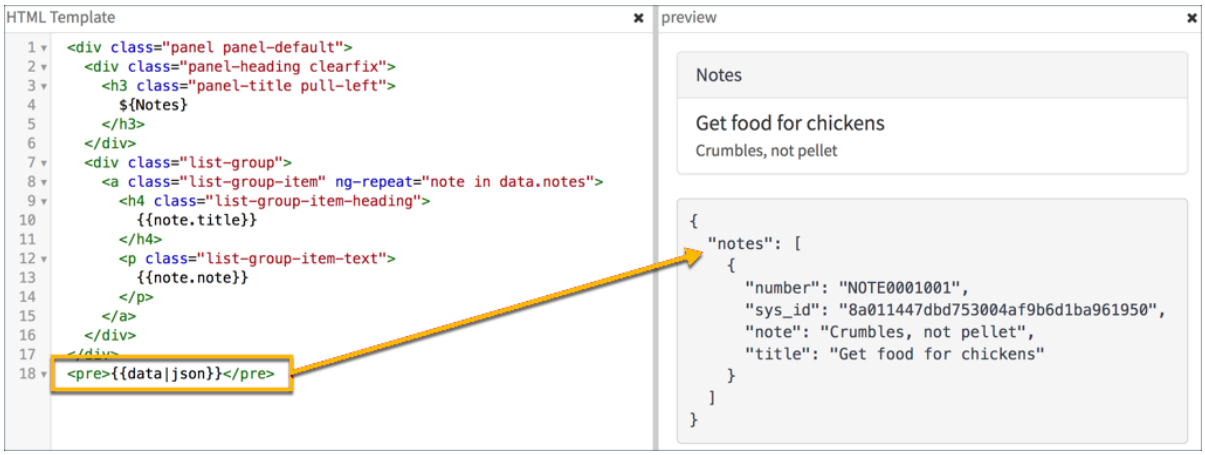
Third-Party Debugging Tools

Many developers use third-party debugging tools when debugging browser-based applications. For example, ng-inspector Chrome extension (https://chrome.google.com/webstore/detail/ng-inspector-for-angularj/aadgmnobpdmgmigaicncghmmoeflnamj?hl=en)

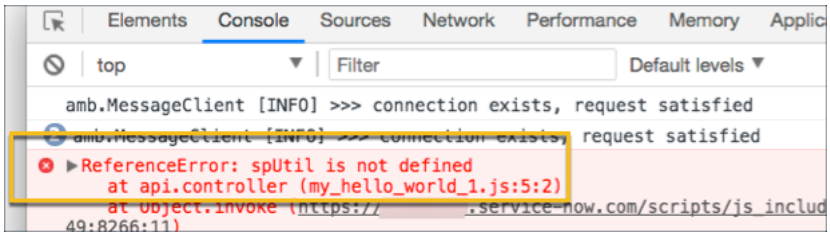


Client-side Debugging

- The client-side Widget API includes methods which can be used for logging/debugging:
  - *spUtil.addErrorMessage()*
  - *spUtil.addInfoMessage()*
  - *spUtil.addTrivialMessage()*
  - *spModal.alert()*
- Output an object to the portal page by modifying the HTML Template:



- Browser tools for debugging:



Server-side Debugging

The server-side *GlideSystem* (https://developer.servicenow.com/app.do#!/api_doc?v=utah&id=c_GlideSystemScopedAPI) class includes methods that can be used for logging/debugging:

- Global API
  - *gs.log()*
  - *gs.logError()*
  - *gs.logWarning()*
- Scoped API
  - *gs.warn()*
  - *gs.info()*
  - *gs.debug()*
  - *gs.error()*
- Global API and Scoped API
  - *gs.addInfoMessage()*
  - *gs.addErrorMessage()*

EXERCISE (18 OF 35)

Exercise: Populate the Notes List Widget

In this exercise, you will populate the *Notes List* widget with *Note* records belonging to the currently logged in user. You will display the Note *title* and the first twenty characters of the *description*.

Preparation

1. Open the *Notes List* widget in *Widget Editor* .
   1. If not still open, open the *CreateNotes* application in Studio for editing.
   2. Use the Application Explorer to open **Service Portal > Widgets > Notes List**.

2. Use the developer site API documentation to learn about the *getRecordDisplayValues()* method in the scoped ***GlideSPScriptable*** (https://developer.servicenow.com/app.do#!/api_doc?v=utah&id=c_GlideSPScriptableScopedAPI) API.

Write the Server Script

1. If the *Server Script* pane is not open in Widget Editor, enable it by selecting **Server Script** in the Widget Editor header.

2. Examine the pseudo-code for the script you will write:

   Create the notes property on the data object which will contain an array of objects
   Query the database to find all Note table records for the currently logged in user
   Sort the records by descending order based on the sys_created_on date field value
   For each of the Note records returned
   Create an empty object, noteObj
   Get the display values for the number, title, and sys_id fields and put those values into the noteObj object
   Get the first 20 characters of the description field and add that value to the noteObj object
   Push the noteObj into the notes array

3. Replace the contents of the *Server Script* pane with this script:

```
(function() {
//create an array to populate with notes
data.notes = [];
var noteGR = new GlideRecord('x_snc_createnotes_note');
noteGR.addQuery('user', gs.getUser().getID());
noteGR.orderByDesc('sys_created_on');
noteGR.query();
while (noteGR.next()) {
var noteObj = {};
//use service portal helper method to get some display values
$sp.getRecordDisplayValues(noteObj, noteGR, 'number,title,sys_id');
//get the first 20 characters of the description
noteObj.note = noteGR.getValue('note').slice(0,20);
//push the populated obj into the array
data.notes.push(noteObj);
}
})();
```

4. Apply formatting to the script.

   1. Highlight the contents of the *Server Script* pane.
   2. Press **<Shift> + <tab>** on your keyboard.

5. Read through the script and compare it against the psudeo-code to make sure you understand what the script does.

6. Click the **Save** button.

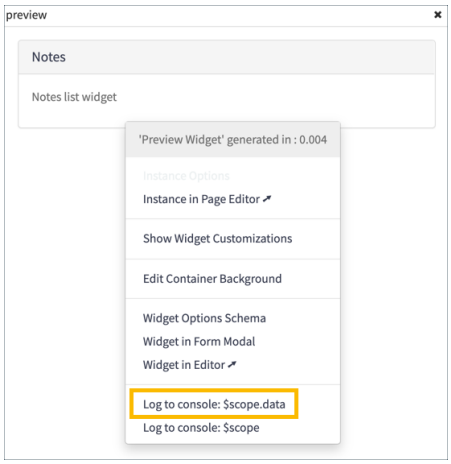
Preview the Notes List Widget

1. Click the **Preview** button in Widget Editor.

2. Notice the *Notes widget* appearance is unchanged.

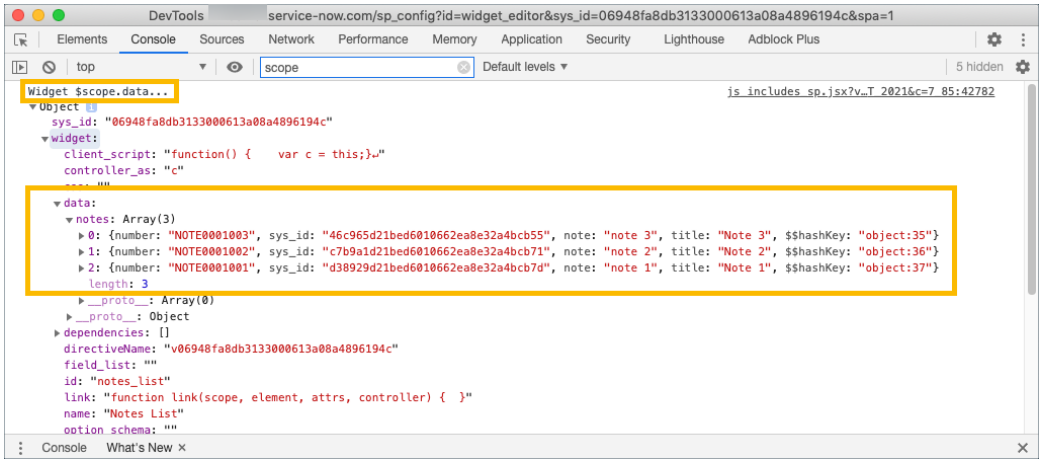
   QUESTION: Why do the *Notes* records not appear in the *Notes List* widget?
   ANSWER: Although you wrote a Server Script to query the database, you have not updated the HTML to respond to changes to the *$scope.data* object.

3. In the *Preview* pane, **<ctrl> + click** on the *Notes List* widget (on some operating systems **<ctrl> + right-click**) and select the **Log to console: $scope.data** menu item.



4. Using the appropriate strategy for your browser, open the JavaScript console.

5. Look for the notes array of objects in *$scope.data*. The example was captured in Chrome. The appearance of your console window may be different depending on which browser you are using.



QUESTION: The *Notes* record you created for Beth Anglin does not appear in the *notes* array of objects. Why not?
ANSWER: The *GlideRecord* query looks for records belonging to the currently logged in User. The currently logged in user is *System Administrator* so Beth Anglin#singleQuotes Notes records will not be included in the results of the query.

6. Close the *Preview* pane.

Update the HTML Template to Display the List of Note Records

1. If the *HTML Template* pane is not open in Widget Editor, enable it by clicking **HTML Template** in the Widget Editor header.

2. Replace the contents of the *HTML Template* pane with this logic:

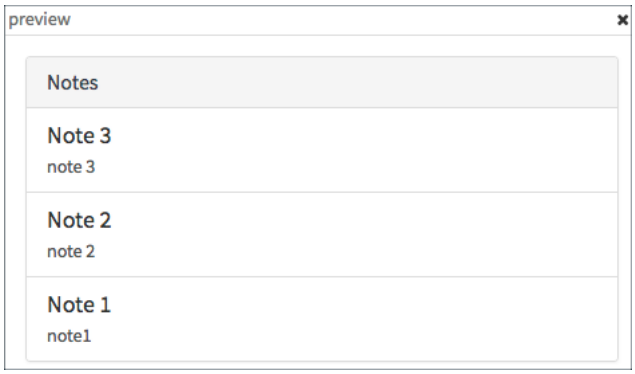
```
<div class="panel panel-default">
<div class="panel-heading clearfix">
<h3 class="panel-title pull-left">
${Notes}
</h3>
</div>
<div class="list-group">
<a class="list-group-item" ng-repeat="note in data.notes">
<h4 class="list-group-item-heading">
{{note.title}}
</h4>
<p class="list-group-item-text">
{{note.note}}
</p>
</a>
</div>
</div>
```

3. Apply formatting to the HTML Template.
    1. Highlight the contents of the *HTML Template* pane.
    2. Press **<Shift> + <tab>** on your keyboard.
4. In the HTML, notice the use of ng-repeat to iterate through the objects in the *data.notes* array.
5. Click the **Save** button.

Preview the Notes List Widget Again

1. Click the **Preview** button in Widget Editor.

2. The *Notes* records returned by Server Script's *GlideRecord* query should be displayed in the *Notes List* widget. Only the title and the first 20 characters of the description are displayed. Your records may be different than the example.



3. Close the *Preview* pane.

ARTICLE (19 OF 35)

Using AngularJS Events with Widgets

AngularJS uses a publish and subscribe strategy for handling events. Events are useful for notifying widgets about important things happening in other widgets. For example:

- Record selected
- Changed value
- Data deleted
- Data added
- And more....

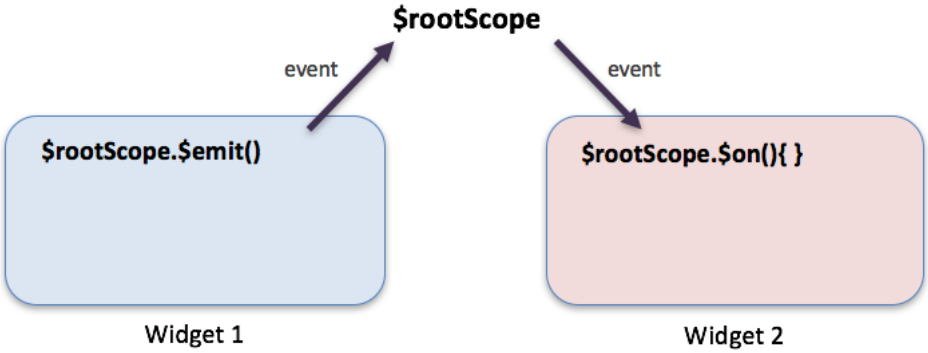
Work with events in AngularJS using these functions:

- *$emit()* (https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$emit): Send an event up the scope hierarchy
- *$on()* (https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on): Listen for events of a given type

> **DEVELOPER TIP**: Avoid the use of *$rootScope.$broadcast()* (https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$broadcast) because it can cause performance issues.

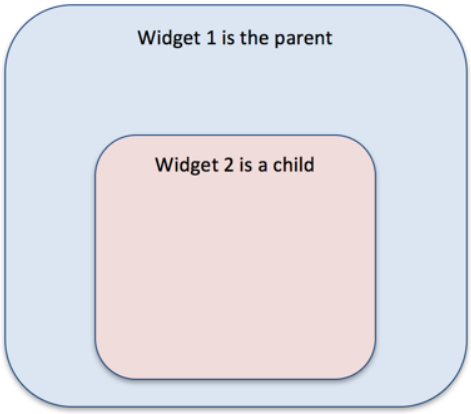
Widget Client Scripts create the widget's controller, *c*.

When working with multiple widgets on a page, widget controllers are siblings; they do not have a parent-child relationship. To emit and listen for events, use the parent of all scopes, *$rootScope*.



Embedded Widgets

Although not covered in this module, it is possible to embed a widget (https://docs.servicenow.com/bundle/utah-servicenow-platform/page/build/service-portal/concept/c_NestedWidgets.html) in another widget. When widgets are embedded, a parent-child relationship does exist.



EXERCISE (20 OF 35)

Exercise: Emit and Respond to Events

In this exercise, you will emit an event from the *Notes List* widget to the *Notes Body* widget when a user selects a record. The *Notes Body* widget will respond to the event by displaying record information.

Edit the Notes List HTML Template to Respond to a Click

1. In the HTML Template, locate this line:

```
<a class="list-group-item" ng-repeat="note in data.notes">
```

2. Edit the line by adding an *ng-click* for when a user clicks a *Note* record in the list:

```
<a class="list-group-item" ng-click="c.selectItem($index)" ng-repeat="note in data.notes">
```

3. Click the **Save** button.

Edit the Notes List Client Script to Emit an Event

1. If the *Client Script* pane is not open in Widget Editor, enable it by clicking **Client Script** in the Widget Editor header.

2. Replace the Client Script with this script:

```
function($rootScope,$scope) {
        /* widget controller */
        var c = this;
        c.selectItem = function(idx) {
                var id = c.data.notes[idx].sys_id;
                console.log('Note ID: ' + id);
                $rootScope.noteID = id;
                $rootScope.$emit('selectNote', id);
        }
}
```

3. If needed, apply indentation using **<Shift> + <tab>**.

4. Examine the script to see what it does.

5. Click the **Save** button.

Test the Notes List Widget

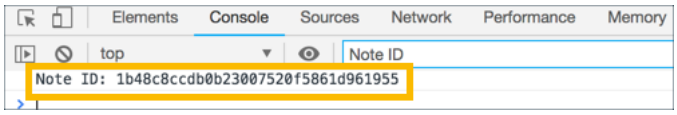
1. You will soon be working with two widgets so instead of testing with the *Preview* pane, open a new browser tab or window and navigate to:

```
https://<your_instance>.service-now.com/notes?id=notes_home
```

2. Using the appropriate strategy for your browser, open the JavaScript console.

3. Click a record in the *Notes List* widget.

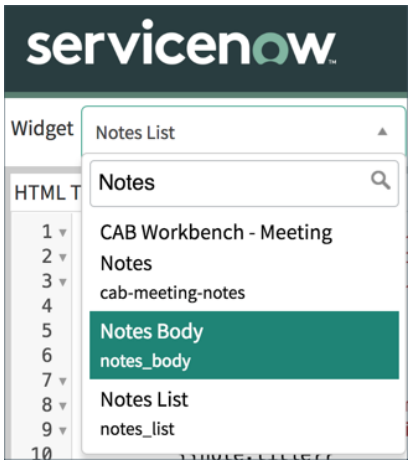
4. Examine the JavaScript console for a log message. The message is written when the event is emitted.



5. Leave the testing tab/window open and return to Widget Editor.

Edit the Notes Body Widget Client Script

1. In Widget Editor, use the *Widget* list to switch to editing the *Notes Body* Widget.



2. Replace the Client Script with this script:

```
function($scope,$rootScope) {
                /* widget controller */
                var c = this;

                $rootScope.$on('selectNote', function(event,data) {
                        console.log('Listener caught NoteID: ' + $rootScope.noteID);
                });
        }
```

3. Examine the Client Script to make sure you understand what it does. Notice, in particular, that the script responds when the *selectNote* event is emitted.

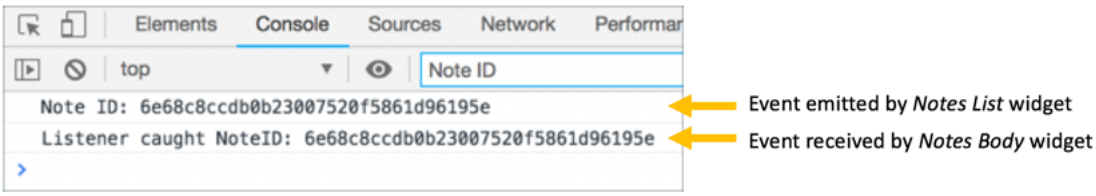
4. Click the **Save** button.

Test Receiving the Event

1. Switch back to the testing tab/window you have open and reload the page.

2. If you closed the JavaScript console, open it now.

3. Click a *Note* record in the *Notes List* widget. You should see log messages for when the event is emitted and when it is received.



4. Leave the testing tab/window open and return to Widget Editor.

Edit the Notes Body Client Script Again

Instead of logging a *sys_id* to the JavaScript console when an event is received, the Client Script should get the selected record's field values from the server to display in the *Notes Body* Widget.

1. Replace the Client Script with this script:

```
function($scope,$rootScope) {
                /* widget controller */
                var c = this;

                $rootScope.$on('selectNote', function(event,data) {
                        c.server.get({
                                action: 'getNote',
                                noteID: $rootScope.noteID
                        }).then(function(r) {
                                c.data.title = r.data.note.title;
                                c.data.note = r.data.note.note;
                                c.data.noteID = r.data.note.sys_id;
                        });
                });
        }
```

2. Examine the script to make sure you understand what it does.
    - *this.server.get()* calls the Server script and passes custom input
    - *this.server.get()* returns a promise. When the response is received from the server, the *.then()* function logic executes.

3. Click the **Save** button.

Edit the Notes Body Widget Server Script

In the *Notes Body* widget Server Script, write the logic to respond to the *getNote* action called from the Client Script.

1. In the *Script Editor* pane, replace the existing Server Script with this script:

```
(function() {
                /* populate the 'data' object */

                if (input && input.noteID) {
                        var note = new GlideRecord('x_snc_createnotes_note');
                        if (note.get(input.noteID)) {
                                if (input.action == 'getNote') {
                                        data.note = {};
                                        $sp.getRecordValues(data.note, note, "title, note, sys_id");
                                }
                        }
                }
        })();
```

2. Examine the script to make sure you understand what it does:
    - Recall that the *input* object is the *data* object received *from* the Client Script's controller.
    - What does the *GlideSPScriptable* *getRecordValues()* (https://developer.servicenow.com/app.do#!/api_doc?v=utah&id=r_GSPS-getRecordValues_O_GR_S) method do?
3. Click the **Save** button.

Edit the Notes Body Widget HTML Template

1. In the *HTML Template* pane, replace the existing HTML with this HTML:

```
<div class="panel panel-default" ng-show="c.data.noteID">
          <div class="panel-heading clearfix">
            <div class="row">
              <div class="col-md-12">
                <input class="form-control" id="note-title" ng-model="c.data.title" />
              </div>
            </div>
          </div>
          <div class="panel-body">
            <textarea class="form-control" id="note-body" ng-model="c.data.note" ></textarea>
          </div>
        </div>
```

2. Examine the HTML to make sure you understand what it does.
3. Click the **Save** button.

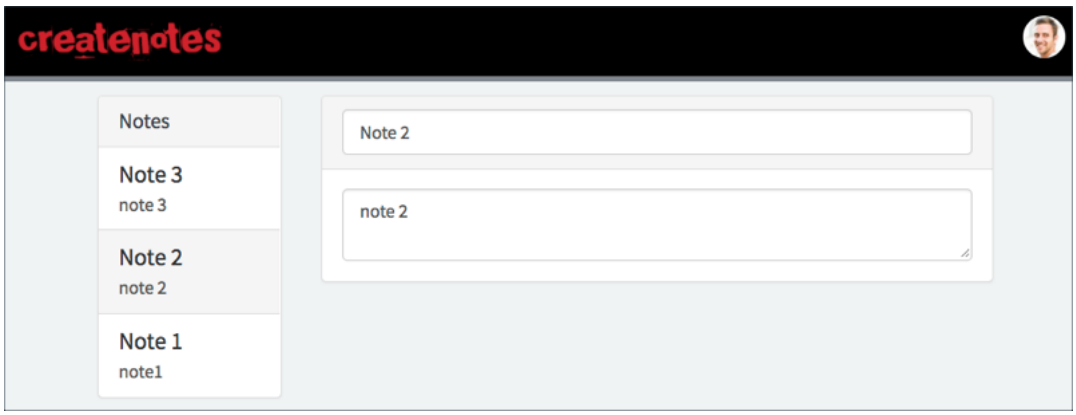
Test the Notes Body Widget Logic

1. Switch back to the testing tab/window you have open and reload the page.

    QUESTION: The *Notes Body* widget has an HTML Template but is not rendered on the portal page until you select a record in the *Notes List* widget. Why not?
    ANSWER: In the HTML Template, the AngularJS directive *ng-show* is used to show/hide the widget

based on whether *c.data.noteID* has a value. Until a record is selected in the *Notes List* widget, the *Notes Body c.data.noteID* property has no value.

2. Click a **Note** record in the *Notes List* widget. You should see the record's *Title* and *Description* in the *Notes Body* widget.



EXERCISE (21 OF 35)

Exercise: Update Notes

In this exercise, you will update the *Notes List* and *Notes Body* widgets to allow users to update *Note* records from the *Notes Body* widget.

Preparation

1. Return to the tab/window you have been using to test the *Notes List* and *Notes Body* widgets and reload the page. If you closed the tab/window, open a new one:

```
https://<your_instance>.service-now.com/notes?id=notes_home
```

2. Click a **Note** record in the *Notes List* widget.

3. In the *Notes Body* widget, edit the *Note* record **Title** and **Description**.

4. Examine the *Notes List* widget. Do you see the record updates you made in the *Notes Body* widget in the *Notes List* widget?

5. In the *Notes List* widget, click a different *Note* record, then click the *Note* record you edited in the last step.

Attempting to update a *Note* record through the widgets has a couple of issues:

- Changes are not saved

- The *Notes List* widget is not notified of changes in the *Notes Body* widget

Update the Notes Body Widget HTML Template

Add the *ng-change* directive to both the *note-title* and *note-body* HTML elements so the widget can respond to changes.

1. Open the **Notes Body** widget for editing in Widget Editor.

2. Replace the contents of the HTML template with this HTML. The *ng-change* and *ng-model-options* directives were added to the *note-title* input and the *note-body textarea* (lines 5 and 10).

```
<div class="panel panel-default" ng-show="c.data.noteID">
  <div class="panel-heading clearfix">
    <div class="row">
      <div class="col-md-12">
        <input class="form-control" id="note-title" ng-model="c.data.title" ng-change="c.updateNote('tit
      </div>
    </div>
  </div>
  <div class="panel-body">
    <textarea class="form-control" id="note-body" ng-model="c.data.note" ng-change="c.updateNote('body')"
  </div>
</div>
```

3. Examine the use of the *ng-change* directive. What function is it calling on the Client Script?

4. Click the **Save** button.

Update the Notes Body Widget Client Script

The *ng-change* directives in the HTML Template call *updateNote()*. You must add a function to the widget's Client Script to handle the updates. The function must specify the action name, pass necessary properties and values to the server so the updates can be written to the database, and supply a callback function for the *server.get()*.

1. Add this new function to the *Notes Body* widget Client Script. Do not replace the entire Client Script. It is up to you to determine where to place the new function in the Client Script.

```
c.updateNote = function(updateType) {
                c.server.get({
                        action: 'updateNote',
                        noteID: c.data.noteID,
                        noteBody: c.data.note,
                        noteTitle: c.data.title
                }).then(function(r) {
                });
        }
```

   QUESTION: Where should you place the *c.updateNote()* function in the Client Script?
   ANSWER: Place the *c.updateNote()* function on line 4 OR line 15:

Client Script

```
1  function($scope,$rootScope) {
2    /* widget controller */
3    var c = this;
4
5    $rootScope.$on('selectNote', function(event,data)
6      c.server.get({
7        action: 'getNote',
8        noteID: $rootScope.noteID
9      }).then(function(r) {
10       c.data.title = r.data.note.title;
11       c.data.note = r.data.note.note;
12       c.data.noteID = r.data.note.sys_id;
13     });
14   });
15
16 }
```

2. Examine the Client Script to make sure you understand what it does. The script is currently not doing anything with the *updateType* value passed in from *ng-change*.

3. Click the **Save** button.

Update the Notes Body Widget Server Script

The Server Script must update the *Notes* record in the database using the new values received from the Client Script.

1. Add this *else if* to the Server Script. Do not replace the entire Server Script. It is up to you to determine where to place the else if in the Server Script.

```
else if (input.action == 'updateNote') {
                note.title = input.noteTitle;
                note.note = input.noteBody;
                note.update();
        }
```

QUESTION: Where should I place the *else if* logic in the Server Script?
ANSWER: Place the *else if* logic on line 11:

Server Script

```
1  (function() {
2    /* populate the 'data' object */
3
4    if (input && input.noteID) {
5      var note = new GlideRecord('x_snc_createnotes_note');
6      if (note.get(input.noteID)) {
7        if (input.action == 'getNote') {
8              data.note = {};
9              $sp.getRecordValues(data.note, note, "title, note, sys_id");
10       }
11
12     }
13   }
14 })();
```

2. Examine the logic to make sure you understand what it does. Recall that:
    - The *input* object is received *from* the Client Script
    - The *GlideRecord update()* method writes new values for an existing record to the database
3. Click the **Save** button.

Test the Record Update Feature

1. Return to the tab/window you have been using to test and do a hard reload of the page to make sure the widgets are not running using cached logic.
2. Click a **Note** record in the *Notes List* widget.
3. In the *Notes Body* widget, edit the **Title** and **Description**. You may see a message about cross-scope privilege which you can ignore.
4. Click a different **Note** in the *Notes List* widget.
5. In the *Notes List* widget, examine the record you edited. Do you see the changes in the *Note* record?
6. In the main ServiceNow browser window (not Studio), use the *All* menu to open **CreateNotes > Notes**. Do you see the changes to the record you edited?
7. Return to the test tab/window and reload the page. Do you see the changes now?

QUESTION: Why did you have to reload the page to see the updates to the record in the *Notes List* widget?
ANSWER: The Server Script was able to successfully update the *Note* record in the database but the *Notes List* widget had no way of knowing about the change. That is to say, the change was not emitted.

Challenge

As you have seen, the *Notes List* widget does not know about changes made to *Note* records in the *Notes Body* widget. Your Challenge is to emit the update for the *Note* record *Title* and *Description* from the *Notes Body* widget. The *Notes List* widget will listen for the event and will update the appropriate record in the list in response to the event.

CHALLENGE SOLUTION:
Any set of requirements can be solved in many ways. There is a working solution in the *devtraining-createnotes-utah* repository. To see the solution:

1. Save your work to the GitHub repository. Saving your work will not save any *Notes* records you created because the *Notes* records are not application files.
    1. In Studio, open the **Source Control** menu and select the **Commit Changes** menu item.
    2. Add a **Commit Message** of your choice.
    3. Click the **Commit Changes** button.
2. In Studio, open the **Source Control** menu and select the **Create Branch** menu item.
3. For the *Branch Name*, use the name of your choice such as **UpdateNotesChallengeSolution**.
4. In the *Create from Tag* field, choose **Solution_UpdateNotes_WithChallenge** then click the **Create Branch** button.

5. To load the changes, return to the main ServiceNow browser tab (not Studio) and click the browser#singleQuotes reload button to refresh the page.

The changes to the code for both widgets is commented. You will see code changes in:

- **Notes Body Widget**: Client Script, Server Script
- **Notes List Widget**: Client Script

ARTICLE (22 OF 35)

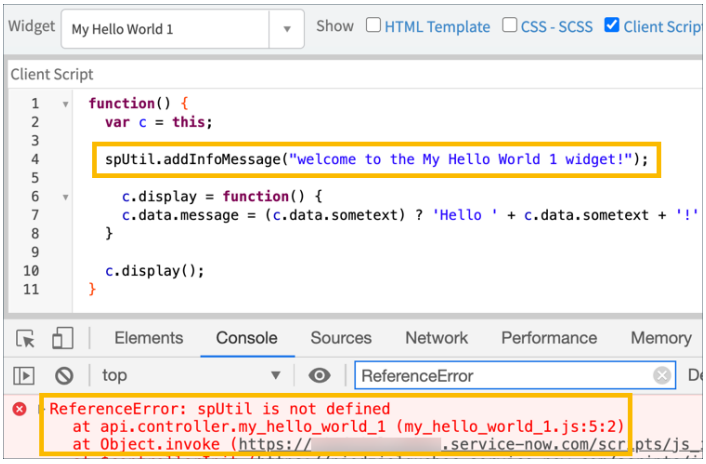
Using the Client-side Widget API

In an earlier part of this module, you learned there are four client-side Widget APIs:

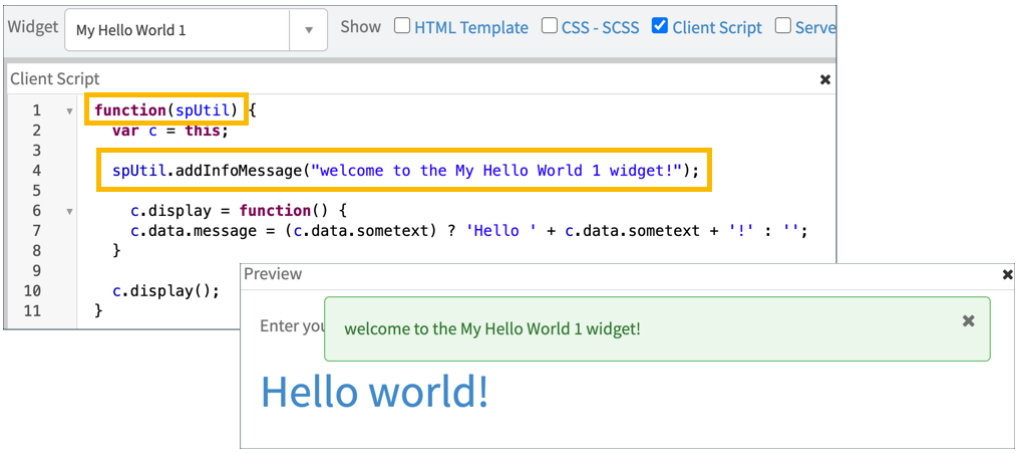
- ***spUtil*** (https://developer.servicenow.com/dev.do#!/reference/api/utah/client/spUtilAPI): Contains utility methods to perform common functions in a Service Portal widget client script.
- ***spModal*** (https://developer.servicenow.com/dev.do#!/reference/api/utah/client/SPModal-API): Methods provide an alternative way to show alerts, prompts, and confirmation dialogs.
- ***spAriaUtil*** (https://developer.servicenow.com/dev.do#!/reference/api/utah/client/spAriaUtil-API): Uses an AngularJS service to show messages on a screen reader.
- ***spContextManager*** (https://developer.servicenow.com/dev.do#!/reference/api/utah/client/spContextManagerAPI): Makes data from a Service Portal widget available to other applications and services in a Service Portal page.

In order to use classes from the client-side Widget API, the *global* object for the API must be passed as a dependency to the Client Script. The Client Script creates the AngularJS controller using the passed-in dependencies.

If a Client Script attempts to use a client-side Widget API without passing the dependency, a *ReferenceError* occurs at runtime.



Pass dependencies in the Client Script function.



EXERCISE (23 OF 35)

Exercise: Confirm Delete Modal

In this exercise, you will load a new version of the *CreateNotes* app from your GitHub repository. The new version has logic for filtering, adding new *Note* records, and deleting *Note* records. You will add a *confirmation* dialog to the logic for deleting *Note* records.

Preparation

1. Save your work to the GitHub repository.
    1. In Studio, open the **Source Control** menu and select the **Commit Changes** menu item.
    2. Make sure all changes are selected, then click the **Continue** button.
    3. Add a **Commit comment** of your choice.
    4. Click the **Commit Files (<number of changes>)** button.
    5. When the commit has completed successfully, click the **Close** button in the *Commit Changes* dialog.
2. Load a new version of the *CreateNotes* application. Loading the new version may take some time. Be patient while the version loads.
    1. In Studio, open the **Source Control** menu and click the **Create Branch** option.
    2. Configure the branch.

        Branch Name: **ConfirmationModal**

        Create from Tag: **LoadForConfirmDeleteModal**
    3. Click the **Create Branch** button.
    4. In the *Create Branch* dialog, click the **Close** button.

Explore the CreateNotes Application

1. Return to the main ServiceNow browser window, not Studio, and reload the browser page.

2. Examine the list of *Notes* records.

    1. Use the *All* menu to open **CreateNotes > Notes**.
    2. Examine the list of Notes records to see which records have your user in the *User* field.

3. Return to the tab/window you have been using to test the *Notes List* and *Notes Body* widgets and do a hard reload of the page. If you closed the tab/window, open a new tab or window:

```
https://<your_instance>.service-now.com/notes?id=notes_home
```

4. Create a *Note* record.

    1. Click the **Add Note** button (

) in the *Notes List* widget header.

    2. Add the **Title** and **Description** of your choice.

    QUESTION: Recall that only *Notes* records for the currently logged in user are displayed in the *Notes List* widget. How is the *User* value set on the *Notes* records created in the *CreateNotes* portal?
    ANSWER: There are different strategies one could use to set the *User* value for new *Note* records including:

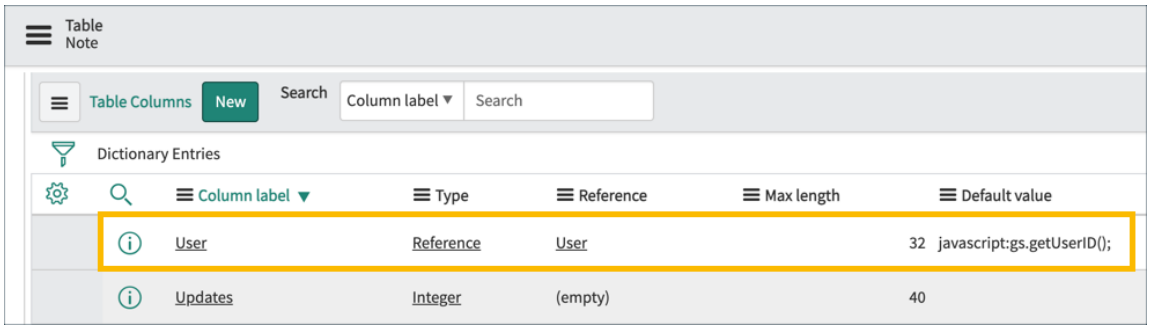
    - Set the value in the widget#singleQuotes Server-side script
    - Set the value using a before Business Rule for record inserts
    - Set a default value in the *User* field#singleQuotes Dictionary Entry

    The drawback to using the Server-side widget script is that if users use the **CreateNotes > Notes** module to create a record, the *User* field value must be set manually. The benefit to using this strategy is that the logic exists in the widget itself and is easy to find.

    The drawback to using a before Business Rule is that you have another application file to maintain and you, the developer, have to remember that not all of the logic for your application is in the widget scripts.

    The drawback to setting a default value in the *User* field#singleQuotes Dictionary Entry is that you have to remember you did that. You can see the field#singleQuotes default value in the *Table* record but you have to remember to look there.

    The Dictionary Entry *Default* value is how the *User* field is set in the *CreateNotes* application you loaded from the GitHub repository for this exercise.

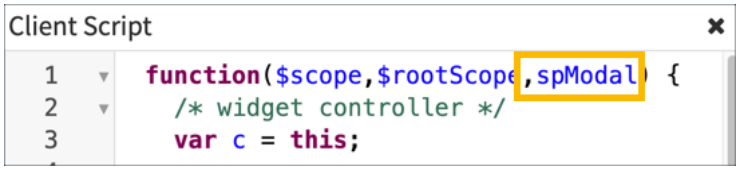


5. Delete one of the *Notes* records.
    1. Select a *Notes* record in the *Notes List* widget.
    2. In the *Notes Body* widget, click the **Delete** button (
    
).
    3. Examine the *Notes List* widget. Is the record deleted? To be sure, switch to the main ServiceNow browser window and use the *All* menu to open **CreateNotes > Notes**.

Add Logic to Confirm Before Deleting

You may have noticed that *Notes* records are deleted immediately when the *Delete* button is clicked. What happens if a user accidentally clicks the *Delete* button? Whoops. Records are not recoverable (no undo option) and must be created again if mistakenly deleted. To prevent accidental loss of data you will add a confirmation modal to the *CreateNotes* application logic.

1. Edit the *Notes Body* widget HTML Template.
    1. Open the **Notes Body** widget for editing.
    2. If not already open, open the HTML Template pane in Widget Editor.
    3. Locate the HTML which renders the *Delete* button and change the *ng-click* value to **c.confirmDelete()**.
    4. Click the **Save** button.
2. If not already open, open the **Client Script** pane in Widget Editor.
    1. Add the **spModal** global object as a dependency.



    2. Add the **c.confirmDelete** logic to the Client Script.

```
c.confirmDelete = function(){
     spModal.confirm("Are you sure you want to delete this Note record?").then(function(confirmed){
       if(confirmed){
       c.deleteNote();
       }
      });
   }
```

3. Click the **Save** button.

Test the Confirmation Modal

1. Return to the tab/window you have been using to test the *Notes List* and *Notes Body* widgets and do a hard reload of the page.
2. Select a **Note** record in the *Notes List* widget.
3. In the *Notes Body* widget, click the **Delete** button.
4. When asked to confirm deletion, click the **Cancel** button. The *Note* record should not have been deleted.
5. Click the **Delete** button again.
6. When asked to confirm deletion, click the **OK** button. The *Note* record should be deleted.

Challenge

After a record is deleted, use the *addTrivialMessage()* method from the *spUtil* API to inform the user the record has been deleted. The message should be something like:

The <note title here> record has been deleted.

The message should appear only after the record is gone from the database.

CHALLENGE SOLUTION:
As always, there are many ways to solve a coding requirement. A possible solution to the Challenge is:

Client Script

```
function($scope,$rootScope,spModal,spUtil) {
  /* widget controller */
  var c = this;
```

Do not forget the dependency

```
c.deleteNote = function() {
  c.server.get({
    action: 'deleteNote',
    noteID: c.data.noteID
  }).then(function(r) {
    spUtil.addTrivialMessage("The " + c.data.title + " record has been deleted.");
    $rootScope.$emit('deleteNote', c.data.noteID);
    c.data.title = '';
    c.data.note = '';
    c.data.noteID = '';
  });
}
```

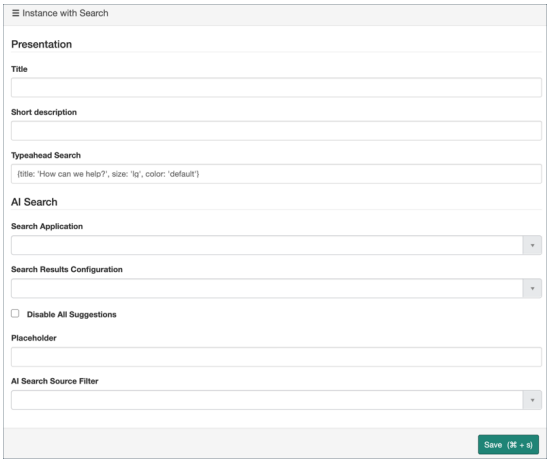
Display the message

What are Widget Options?

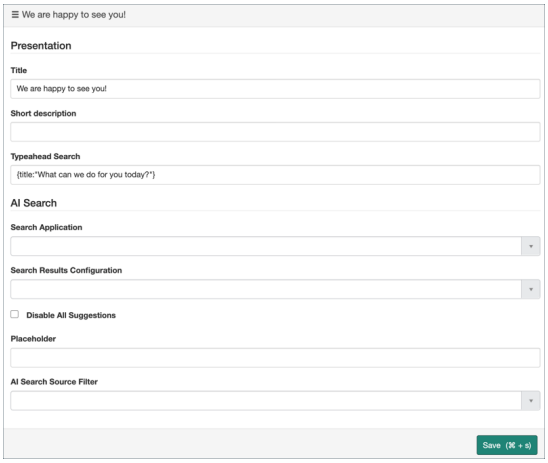
Widgets options are developer-settable parameters which allow each widget instance to be uniquely configured. The example shows the baseline *Homepage Search* widget and two uniquely configured instances of the *Homepage Search* widget.

When a widget is added to a page, a unique instance of the widget is created. When setting widget options, only that unique instance's options are changed. For example, setting the widget options for a *Homepage Search* widget instance sets the options for that unique instance of the *Homepage Search* widget and NOT all instances of the *Homepage Search* widget.

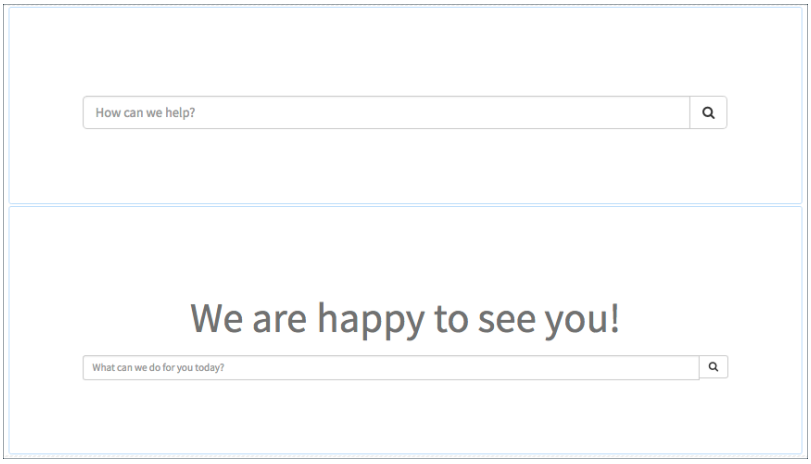
The default *Homepage Search* widget configuration:



Customized *Homepage Search* widget configuration:



The default and customized *Homepage Search* widgets have different appearances but the same behavior. Changing one widget instance's options does not affect any other instances of the same widget.

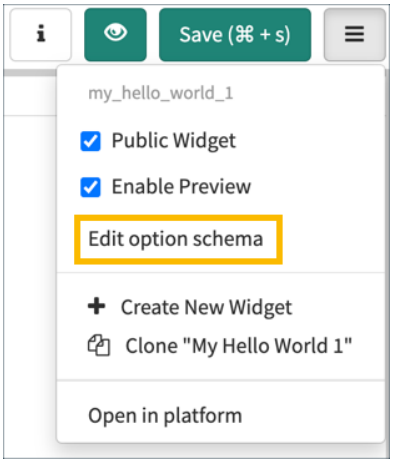


DEVELOPER TIP: Different widgets have different options.

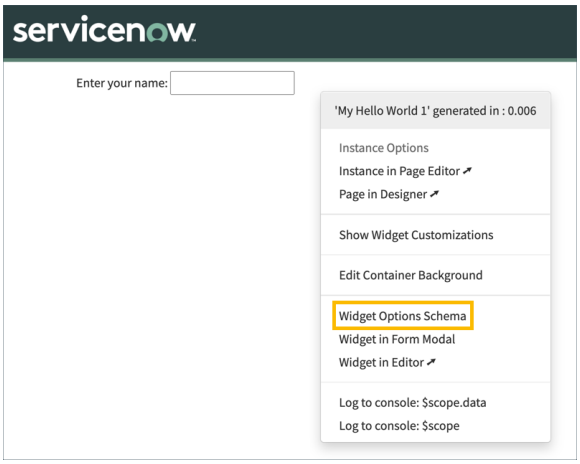
ARTICLE (25 OF 35)

Widget Option Schema

Using widgets options makes widgets more easily reusable. The widget option schema defines the user-configurable fields. To add, edit, or delete option fields, select the **Edit Options Schema** menu item in the *Widget Editor* menu.



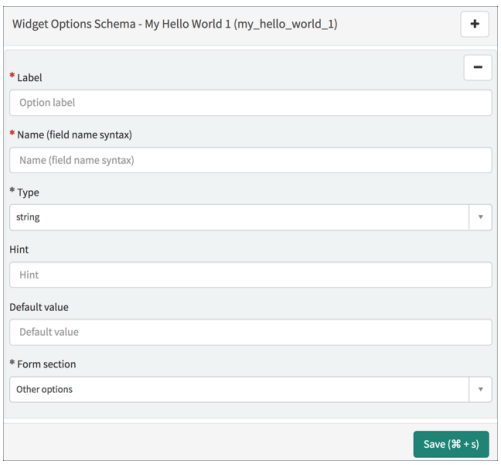
On a portal page, **<ctrl> + click** on a widget (on some operating systems **<ctrl> + right-click**) and select the **Widget Options Schema** menu item.



Add Widget Options

In the widget option schema, click the **Add** button (

) then configure the widget options.



An asterisk indicates mandatory fields.

- ***Label***: Human readable form of the field name
- ***Name***: Name of the field used in scripts
- ***Type***: Data type of the field
- ***Hint***: Provide a brief explanation of the widget for the users
- ***Default Value***: Value to use if the user does not provide a value

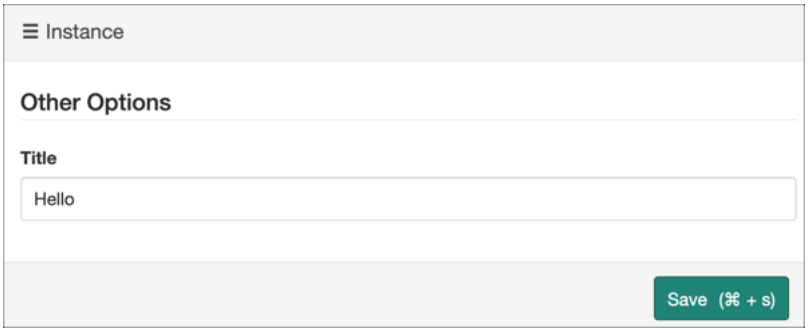
Widget options types are:

- *String*
- *Boolean*
- *Integer*
- *Reference*
- *Choice*
- *Field_list*
- *Field_name*
- *Glide_list*

For field types not supported in the option schema, create an extension table to store a custom widget option schema (https://docs.servicenow.com/bundle/utah-servicenow-platform/page/build/service-portal/concept/c_WidgetInstanceOptions.html).

Setting Widget Options

Service Portal uses the widget option schema to create a modal for setting widget options.



Widget options and their values are part of the *Widget Instance* record (**Service Portal > Widget Instances**). The options are stored in JSON format.

Using Widget Options

The widget options are accessible in both Client and Server Scripts using the *options* global object.

```
// Snippet from a Server Script
if(!options.title){
        options.title = "Hello World";
}


//Snippet from a Client Script
if(!c.options.title){
        c.options.title = "Hello World";
}
```

Widget options are also accessible in the HTML Template using the *options* global object.

```
<h1>{{::c.options.title}}</h1>
```

EXERCISE (26 OF 35)

Exercise: Create Widget Option Schema

In this exercise, you will create and test a widget option schema for the *Notes List* widget.

Preparation

1. Return to the tab/window you have been using to test the *Notes List* and *Notes Body* widgets. If you closed the tab/window, open a new one:

```
https://<your_instance>.service-now.com/notes?id=notes_home
```

2. Examine the *Notes List* widget header.
    1. What is the title of the widget?
    2. How many records are in the *Notes List* widget? If you do not have at least three *Notes* records, create more *Notes* records.

Create a Widget Option Schema - Notes List Widget

1. Open the *Notes List* widget for editing in Widget Editor.
2. Open the **Widget Editor menu** (

) and select the **Edit option schema** menu item.
3. Add an option to the schema.
    1. Click the **Add** button (

).
    2. Configure the new option.

        Label: **Title**

        Name: **title**

        Type: **string**

        Hint: **Specify a title for the Notes List widget**

        Default value: **Notes**
4. Add a second option to the schema.

    Label: **Maximum records to display**

    Name: **maximum_records_to_display**

    Type: **choice**

Choices:

Choices

2
3
5
7
11
13
17

Hint: **Select the maximum number of records to display**

Default value: **5**

5. Click the **Save** button. The *Widget Options Schema* modal closes.

Set the Notes List Panel Title

1. Make this change to the *Notes List* widget's HTML Template:

HTML Template

```
<div class="panel panel-default">
  <div class="panel-heading clearfix">
    <h3 class="panel-title pull-left">
      {{::c.options.title}}
    </h3>
    <button class="btn btn-default pull-right" ng-cli
```

2. Click the **Save** button.

Retrieve a Fixed Number of Notes Records

1. If you are unfamiliar with the *GlideRecord setLimit()* method, read about it in the API docs (https://developer.servicenow.com/app.do#!/api_doc?v=utah&id=r_ScopedGlideRecordSetLimit_Number).
2. Make this change to the *Notes List* widget's Server Script:

Server Script

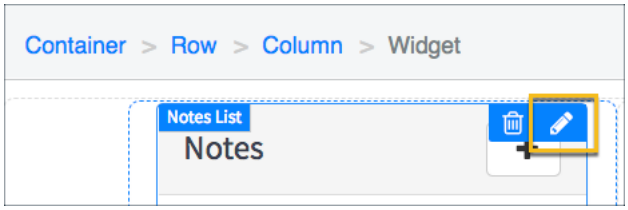
```
(function() {

  data.notes = [];
  var noteGR = new GlideRecord('x_snc_createnotes_note');
  noteGR.addQuery('user', gs.getUser().getID());
  noteGR.orderByDesc('sys_created_on');
  noteGR.setLimit(options.maximum_records_to_display);
  noteGR.query();
```

3. Click the **Save** button.

Configure the Notes List Widget Options

1. In the main ServiceNow browser window, use the *All* menu to open **Service Portal > Service Portal Configuration**.
2. Click the **Designer** tile.
3. Locate the **CreateNotes Home (notes_home)** page and open it for editing.
4. Open the options for the *Notes List* Widget.

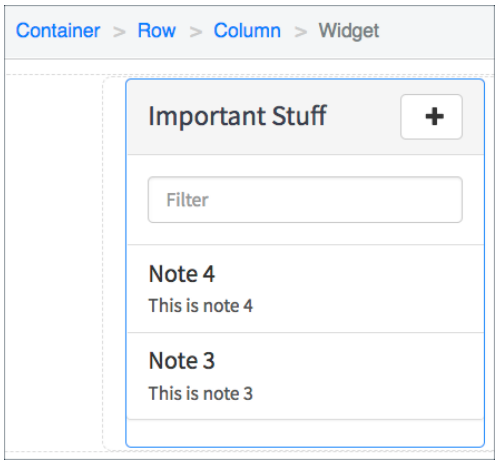


5. Configure the options:

   Title: **Important Stuff**

   Maximum records to display: **2**

6. Click the **Save** button.
7. The *Notes List* widget in the Designer should be updated to show the new options settings.



8. Modify the options again.
   1. Return to the tab/window you have been using to test the *Notes List* and *Notes Body* widgets.
   2. **<ctrl> + click** the *Notes List* widget (on some operating systems **<ctrl> + right-click**) and select the **Instance Options** menu item.
   3. Set the *Title* and *Maximum records to display* values to the values of your choice.
   4. Click the **Save** button.

ARTICLE (27 OF 35)

Directives

Directives in AngularJS are extended HTML attributes. AngularJS has a number of built-in directives, all starting with *ng*, such as:

- *ngApp*
- *ngRepeat*
- *ngShow*
- *ngModel*
- *ngClick*

AngularJS allows user-defined directives to extend the functionality of HTML in applications. The directives can attach a specified behavior to:

- *Element (<macro-name></macro-name>)*
- *Attribute (<div macro-name>)*
- *CSS Class ( )*
- *Comments (<!-- macro-name -->)*

Create an Angular Provider

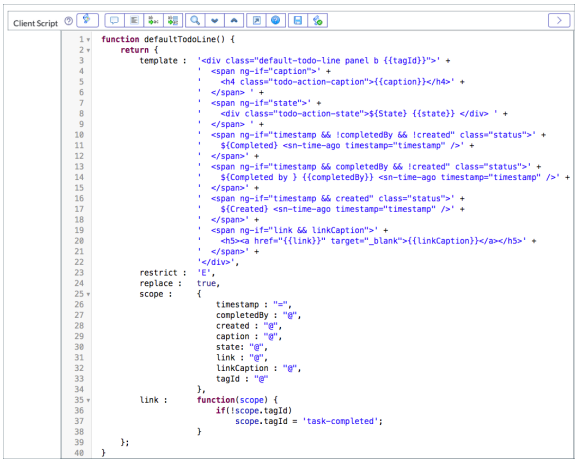
Directives are defined as Angular Providers.

In the main ServiceNow browser window, use the *All* menu to open **Service Portal > Angular Providers**.

Directive names use camelCase which must be referenced using kebab-case in HTML. For example, a directive named *myDirective* is *<my-directive>* in HTML. In kebab-case:

- All letters are lowercase
- Spaces are replaced by hyphens
- Hyphens are inserted before letters which were uppercase

The *Client Script* field defines what the directive does.



In the example, the directive has several properties:

- ***template***: The HTML to render.
- ***restrict***: Specifies which methods can invoke the directive: *Element*, *Attribute*, *Class*, or *Comment*.
- ***replace***: Tells AngularJS to replace the element the directive is declared on.
- ***scope***: An object that contains a property for each isolate scope binding. This is typically used when making components reusable.
- ***link***: The link function is executed while attaching the template to the DOM.

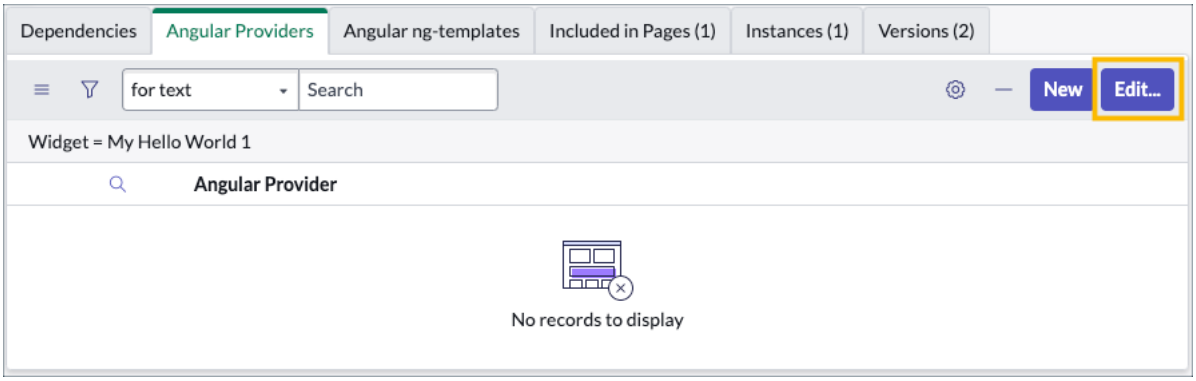
Any valid directive syntax (https://docs.angularjs.org/guide/directive) can be used in an Angular Provider.

ARTICLE (28 OF 35)

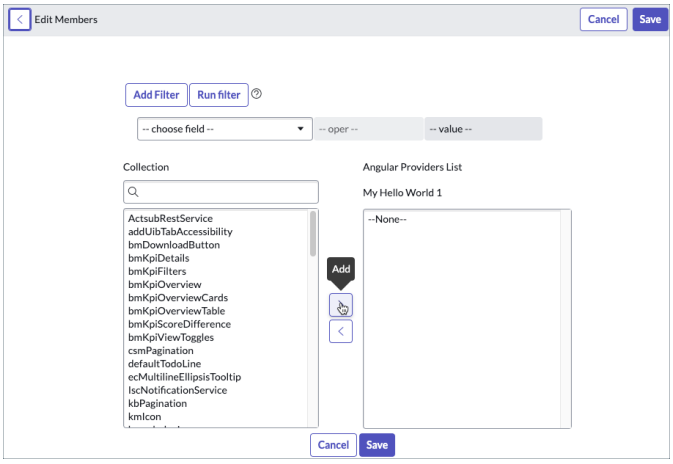
Angular Provider Relationship

For performance reasons, widgets load only the angular providers they use.

To add angular provider relationships, open **Service Portal > Widgets** in the main ServiceNow browser window. Open a widget record for editing. Scroll to the bottom of the form and switch to the Angular Providers related list. Click the **Edit...** button.



Locate the Angular Provider to add in the *Collection* slushbucket and add it to the *Angular Providers List* slushbucket.



The Angular Provider appears on the Widget Editor header in the *Dependencies* list.

EXERCISE (29 OF 35)

Exercise: Create a Directive

In this exercise, you will create a directive and add it as a dependency to the *Notes Body* widget. The directive will render the *Delete* button in the widget.

Create a Directive

1. In the main ServiceNow browser window, use the *All* menu to open **Service Portal > Angular Providers**.

2. Click the **New** button.

3. Configure the new Angular Provider.

   Type: **Directive**
   Name: **deleteButtonConfirm**

4. Add the Client Script to the directive.

```
function(){
                return{
                        template:  '<div class="col-md-1"><button class="btn btn-danger pull-right" ng-cl
                        restrict: 'E'

                };
        }
```

5. If you see an error about the function having an unexpected token, ignore the error. Click the **Submit** button.

Add the Angular Provider to the Notes Body Widget

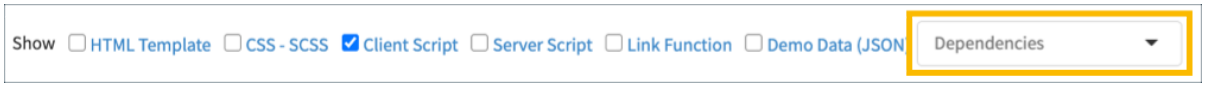
1. In the main ServiceNow browser window, use the *All* menu to open **Service Portal > Widgets**.
2. Open the **Notes Body** widget for editing.
3. Scroll down and open the **Angular Providers** related list.
4. Click the **Edit...** button.
5. Locate the **deleteButtonConfirm** directive in the *Collection* slushbucket.
6. Click the **Add** button

   to move the directive to the *Angular Providers List* slushbucket.
7. Click the **Save** button.

Edit the Notes Body Widget HTML Template

1. Open the **Notes Body** widget for editing in Widget Editor.

2. Click the **Dependencies** list in the Widget Editor header to see the list of dependencies.



3. Edit the HTML. Replace the HTML for the *Delete* button with the directive.

DEVELOPER TIP: The directive could also be written as *<delete-button-confirm/>* since it is an empty element.

4. Click the **Save** button.

Test the Widget

1. Depending on the *Notes Body* widget options set in the previous exercise, you may want to set the *Maximum records to return* value to a high number such as **11** or **17**.

2. Return to the tab/window you have been using to test the *Notes List* and *Notes Body* widgets and reload the page. If you closed the tab/window, open a new one:

```
https://<your_instance>.service-now.com/notes?id=notes_home
```

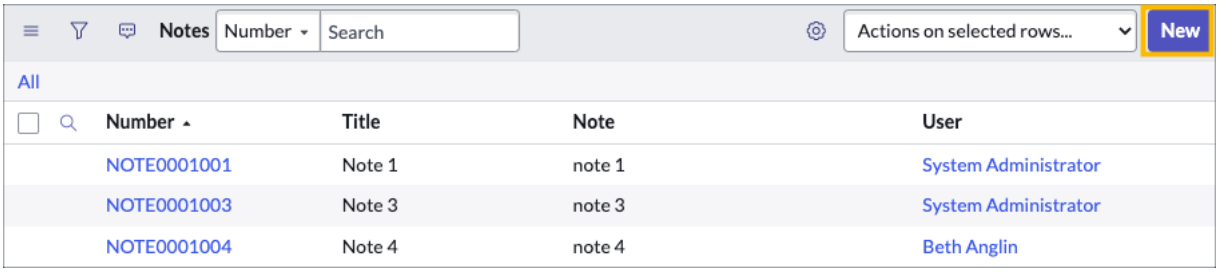
3. Click a **Note** record in the *Notes List* widget.

4. In the *Notes Body* widget, click the **Delete** button. When prompted to confirm, click the **OK** button. The record should be deleted.

ARTICLE (30 OF 35)

recordWatch()

In the *CreateNotes* application's widgets, you are able create, update, and delete *Notes* records. What happens in the widgets if a user creates a record using the standard ServiceNow UI?



Although the *Notes List* and *Notes Body* widgets have been developed to communicate with each other, they are not notified of changes made to the *Notes* table records when the interaction happens through the standard ServiceNow UI.

Use the Client API method *spUtil.recordWatch()* to register a listener in a widget. The listener is notified when table records are inserted, deleted, or updated.

```
spUtil.recordWatch($scope, "table_name", "filter", function(name) {
                console.log(name); //Returns information about the event that has occurred
                console.log(name.data); //Returns the data inserted or updated on the table
          });
        }
```

To use a Record Watch, pass *spUtil* as a dependency to the Client Script.

The *recordWatch* method is passed *$scope*, the name of the table to watch, and any filter condition.

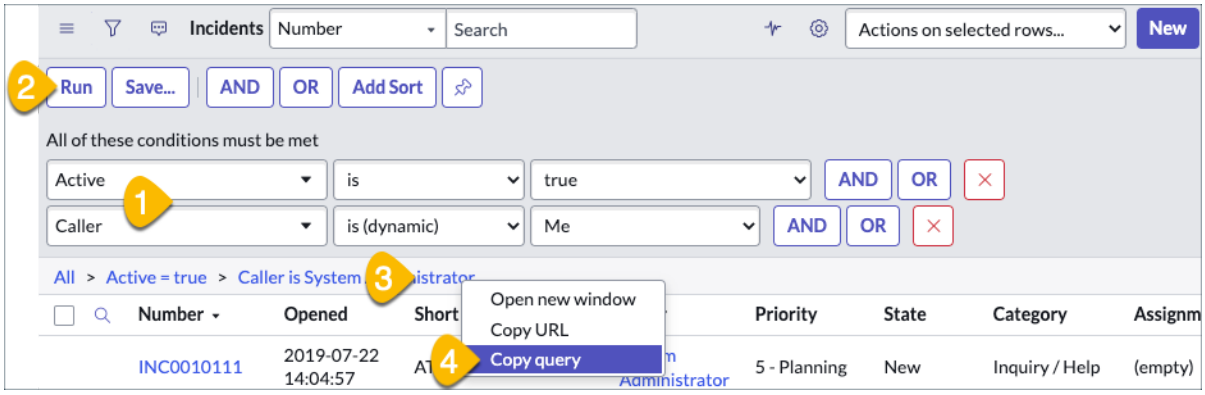
```
spUtil.recordWatch($scope,"incident","active=true",function(name){

        // listen for changes to incident table records where the record is active i

                console.log(name); //Returns information about the event that has occurred
                console.log(name.data); //Returns the data inserted or updated on the table
          });
        }
```

If you are not sure of the syntax for the filter, use a ServiceNow list to build the filter for you.

1. Construct the query using the Filter Builder.
2. Click the **Run** button to execute the query.
3. Right-click the rightmost part of the breadcrumbs.
4. Select the **Copy query** menu item.
5. Paste the query into the *recordWatch* parameters.




```
spUtil.recordWatch($scope,"incident","caller_idDYNAMIC90d1921e5f510100a9ad2572f2b477fe^active=true",function(

                console.log(name.data.operation);
                console.log(name);
        });
```

When changes meeting the filter condition are made to the table records, the callback function executes.

Viewing the Record Changes

Use the browser's JavaScript console to view information from the recordWatch objects. The *data* object, which is a property of the *name* object, contains information about the record which changed.

Responding to Record Changes

Client Script logic responds to the Record Watch event including the Widget API, Widget Script global objects, and Client Script global functions (https://developer.servicenow.com/dev.do#!/learn/courses/utah/app_store_learnv2_serviceportal_utah_service_portal/a

EXERCISE (31 OF 35)

Exercise: Use Record Watch to Detect Notes Record Changes

In this exercise, you will use Record Watch to detect changes to *Notes* table records and respond to:

- Delete
- Update
- Create

Preparation

1. Return to the tab/window you have been using to test the *Notes List* and *Notes Body* widgets and reload the page. If you closed the tab/window, open a new one:

```
https://<your_instance>.service-now.com/notes?id=notes_home
```

2. Return to the main ServiceNow browser window and use the *All* menu to open **CreateNotes > Notes**.

3. Make modifications to the *Notes* table records:

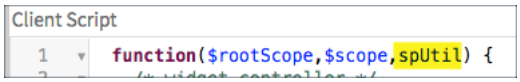
    1. Edit one record's **Title** or **Description** field.
    2. Create a new record.
    3. Delete a record.

4. Return to the tab/window you have been using to test the *Notes List* and *Notes Body* widgets. Do not reload the page. Are the records you modified (update, insert, delete) and their field values in the *Notes List* widget? Are the values correct?

Add the Record Watcher to the Notes List Widget Client Script

1. Open the **Notes List** widget for editing in Widget Editor.

2. Add **spUtil** to the Client Script dependencies.



3. Add the Record Watch logic to the Client Script.

```
  c.newNote = function() {
    c.server.get({
      action: 'newNote'
    }).then(function(r) {
      c.data.notes.unshift(r.data.newNote);
      c.data.noteID = r.data.noteID;
      $rootScope.noteID = c.data.noteID;
      $rootScope.$emit('selectNote', c.data.noteID);
    });
  }

  $rootScope.$on('deleteNote', function(event,data) {
    c.data.notes.splice($scope.notePos, 1);
  });
```


```
}
```

```
spUtil.recordWatch($scope, "table_name", "filter", function(name) {
                console.log(name.data.operation);
                console.log(name);
          });
```

4. Configure the Record Watch parameters.

   table_name: **Name of the Create Notes table**
   filter: **Records with any value in the Number field** (Hint: Use the Filter Builder in the **CreateNotes > Notes** list to determine the query syntax.)

   SOLUTION:

```
spUtil.recordWatch($scope, "x_snc_createnotes_note", "numberANYTHING", function(name) {
```

5. Click the **Save** button.

Examine the Table Record Change Events

1. Return to the tab/window you have been using to test the widgets and do a hard reload of the page to make sure the widgets are not running using cached logic.
2. Enable the browser's JavaScript console using the strategy appropriate to your browser.
3. Return to the main ServiceNow browser window and create a *Notes* record. Use the *Title* and *Description* values of your choice.
4. Save the new *Notes* record.
5. Return to the tab/window you have been using to test the widgets and examine the JavaScript console. Look for the insert.
6. Open **name.data** and **name.data.record** to see the structure of the data sent to the Record Watch. Notice the value in *name.data.operation*.

Respond to the Record Insert

1. Return to editing the *Notes List* widget in Widget Editor.

2. Add logic to the *spUtil.RecordWatch* callback function to respond to the insert event received by the Record Watch. Add the logic after the two *console.log* statements.

```
// Fast and easy... replace the client data object with the server data object
        if(name.data.operation == "insert"){
                c.server.refresh();
        }
```

3. Click the **Save** button.

Test the Record Insert Logic

1. Return to the tab/window you have been using to test the widgets and do a hard reload of the page to make sure the widgets are not running using cached logic.
2. If you closed the JavaScript console, open it again.
3. Switch to the main ServiceNow browser window and open **CreateNotes > Notes**.
4. Add a new *Note* record.
    1. Click the **New** button.
    2. Enter the **Title** and **Description** values of your choice.
    3. Click the **Submit** button.
5. Return to the test tab/window.
6. The new record should appear in the *Notes List* widget.
7. Examine the JavaScript console to see the structure of the *name.data* object and the values in the *name.data.record* object.

Respond to Record Update and Delete - Client Script

In this section you will add logic to the *spUtil.recordWatch* callback function to respond when *Notes* records are updated or deleted. Although in all cases the logic could be to call *this.server.refresh()*, two different strategies will be used for the purpose of demonstration.

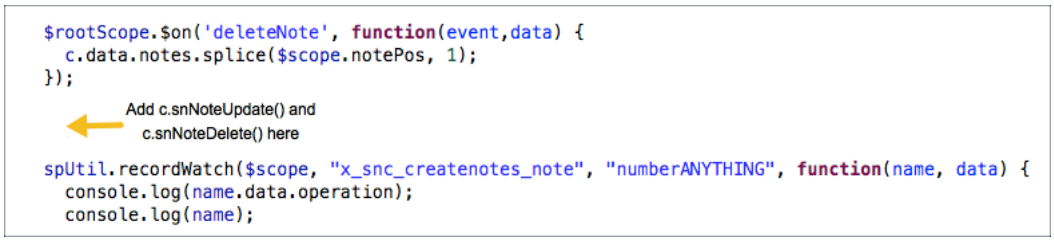
1. Return to editing the **Notes List** widget in Widget Editor.

2. Add logic to the *spUtil.RecordWatch* callback function to respond to the update and delete event received by the Record Watch. Add the logic after the two *console.log* statements and before the record insert logic.

```
// Calls a Client Script function to do the update and pass the updated data object
        // to the server.
        if(name.data.operation == "update"){
                c.snNoteUpdate(name, name.data.sys_id);
        }
        // Calls a Client Script function which deletes a record on the server and passes
        // the updated data object back to the client.
        if(name.data.operation == "delete"){
                c.snNoteDelete(name.data.sys_id);
        }
```

3. Examine the update and delete callback function logic. Notice the calls to *c.snNoteUpdate()* and *c.snNoteDelete()*.

4. Add the **c.snNoteUpdate()** and **c.snNoteDelete()** functions to the Client Script.



```
// Record removed from data object on the client and passed to server
        c.snNoteUpdate = function(name,sysID){

                for (var i=0;i<c.data.notes.length;i++){
                        if(c.data.notes[i].sys_id == name.data.sys_id){
                                if(name.data.record.note){
                                        c.data.notes[i].note = name.data.record.note.display_value;
                                }
                                if(name.data.record.title){
                                        c.data.notes[i].title = name.data.record.title.display_value;
                                }
                                c.server.update();

                        }
                }
        }

        // Record removed from data object on the server and passed back to client
        c.snNoteDelete = function(sysID){

                c.server.get({
                        action: 'snDeleteNote',
                        noteID: sysID

                }).then(function(r){
                        c.data.notes = r.data.notes;
                });

        }
```

5. Click the **Save** button.

Respond to Record Delete - Server Script

The Client Script logic for the *c.snNoteDelete()* function does a *c.server.get*. In this section of the exercise, you will add the Server Script logic for the record deletion event.

1. If not already open in Widget Editor, open the Server Script pane.

2. Add the *snDeleteNote* action logic.

```
  if (input) {
    if (input.action == 'newNote') {
      var newNote = new GlideRecord('x_snc_createnotes_note');
      newNote.newRecord();
      var id = newNote.insert();
      data.noteID = id;
      data.newNote = {};
      $sp.getRecordValues(data.newNote,newNote,"title,note,sys_id");

    }

        ← add the snDeleteNote logic here

  }
}
)();
```

```
if (input.action == 'snDeleteNote') {
                        var delNote = new GlideRecord('x_snc_createnotes_note');
                        // The notes record should already be gone.  Just
                        // making sure it no longer exists.
                        if(delNote.get(input.noteID)){
                                delNote.deleteRecord();
                        }

                }
```

3. Click the **Save** button.

Test the Insert, Delete, and Update Logic

1. Return to the tab/window you have been using to test the widgets and do a hard reload of the page to make sure the widgets are not running using cached logic.
2. If you closed the JavaScript console, open it again.
3. Switch to the main ServiceNow browser window and open **CreateNotes > Notes**.
4. Test the insert record logic to make sure it still works. Add a new Note record.
    1. Click the **New** button.
    2. Enter the **Title** and **Description** values of your choice.
    3. Click the **Submit** button.
5. Return to the test tab/window.
6. The new record should appear in the *Notes List* widget.
7. Examine the JavaScript console to see the structure of the *name.data* object and the values in the *name.data.record* object.
8. Test the record update feature.
    1. Return to the main ServiceNow browser window and open an existing *Note* record for editing.
    2. Change the **Description**, **Title**, or both values on an existing record.
    3. Click the **Update** button.
    4. Return to the test tab/window. The record should also have been updated in the *Notes List* widget.
    5. Examine the JavaScript console to see the structure of the *event* object.
9. Test the record delete feature from a form.
    1. Return to the main ServiceNow browser window and open an existing *Note* record for editing.
    2. Click the **Delete** button.
    3. When prompted to confirm the delete operation, click the **Delete** button.
    4. Return to the test tab/window. The record should not appear in the *Notes List* widget.
    5. Examine the JavaScript console to see the structure of the *event* object.
10. Test the record delete feature from a list.
    1. In the main ServiceNow browser window, use the *All* menu to open **CreateNotes > Notes**.
    2. Click the check box next to a record to select it.
    3. Click the **Actions on selected rows...** button and select the **Delete** item.
    4. When prompted to confirm the delete operation, click the **Delete** button.
    5. Return to the test tab/window. Is the record deleted from the *Notes List* Widget?

6. Examine the JavaScript console. Did the Record Watch receive a *delete* event from the List?

SOLUTION: There is a lot of code to add in this exercise. If your widgets are not responding correctly to the Record Watch events, check out the solution in the *devtraining-createnotes-utah* repository.
Any set of requirements can be solved in many ways. To see the solution:

1. In Studio, open the **Source Control** menu and commit your work.
2. Open **Source Control** menu again and select the **Create Branch** menu item.
3. For the Branch Name, use the name of your choice such as **RecordWatchSolution**.
4. In the *Create from Tag* field, choose **Solution_RecordWatch**, then click the **Create Branch** button.

EXERCISE (32 OF 35)

Exercise: Save Your Creating Custom Widgets Work (Optional)

Source control applications, like GitHub, allow developers to commit changes (save completed work) outside of the Personal Developer Instance (PDI). Commit changes made to the application to save your work in source control.

In this exercise, you will save the work completed in this module to your GitHub repository.

> **NOTE**: See the *GitHub Guide* (/dev.do#!/guide/utah/now-platform/github-guide/github-and-the-developer-site-training-guide-introduction) for more information on how ServiceNow uses GitHub with the Developer Program learning content and to see a video on how to save your work.

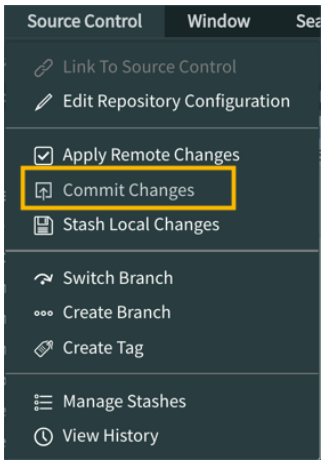
Commit Changes

1. If the *CreateNotes* application is not open in Studio, open it now.

   a. In the main ServiceNow browser window, use the *All* menu to open **System Applications > Studio**.

   b. In the *Select Application* dialog, click the *CreateNotes* application.

2. Open the **Source Control** menu and select the **Commit Changes** menu item.

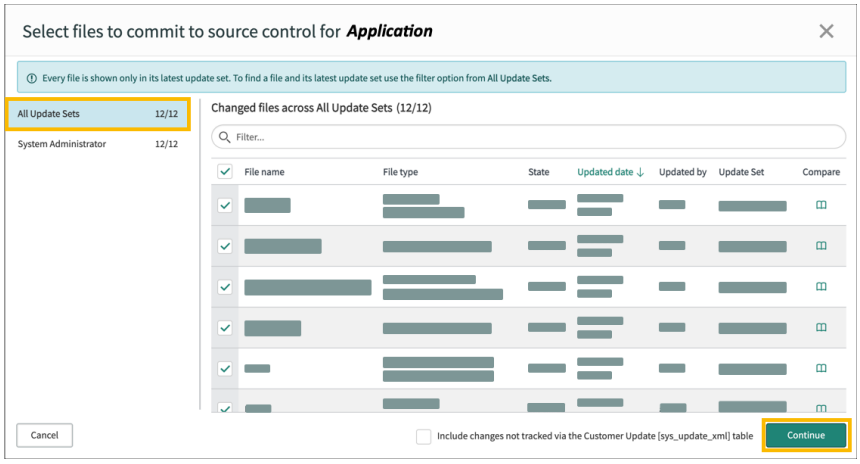


3. Select the updates to commit.

    1. In the *Select files to commit to source control for <Application>* dialog, select **All Update Sets**.

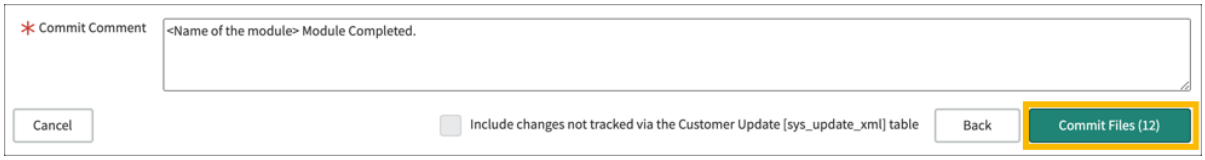
    2. Review the application files to be committed. Your files will differ from those pictured.

    3. Click the **Continue** button.



4. In the *Confirm files to commit to source control for <Application>* dialog, enter a *Commit comment*, such as **Creating Custom Widgets Module Completed**.

5. Click the **Commit Files** button.



6. When the *Commit Changes* dialog reports success, click the **Close** button.

> **NOTE**: If the commit changes fail, you may have entered the ServiceNow repository URL in the URL field instead of the forked repository URL. See the Troubleshooting GitHub Issues (/dev.do#!/guide/utah/now-platform/github-guide/troubleshooting-github-issues) section of the *GitHub Guide* for instructions on how to troubleshoot GitHub connection issues.

ARTICLE (33 OF 35)

Test Your Custom Widgets Knowledge

Want to verify your understanding of custom widgets? These questions will help you assess your progress. For each question, determine your response then click anywhere in the question to see the answer.

QUESTION: Which one of the following is NOT a true statement about widget Client Scripts?

1. Maps server data from *JavaScript* and *JSON* objects to client objects
2. Gathers data from user inputs like input text, radio buttons, and check boxes
3. Processes data for rendering
4. Passes data to the HTML template
5. Passes user input and data to the server for processing

ANSWER: **Response 2 is not a true statement.** The HTML Template gathers data from user inputs like input text, radio buttons, and check boxes.

QUESTION: Which one of the following is NOT a true statement about widget Server Scripts?

1. Sets the initial widget state
2. Sends data to the widget#singleQuotes Client Script using the *data* object
3. Works with record data, web services, and any other data available in ServiceNow server-side scripts
4. Specifies widget parameters
5. Runs server-side queries

ANSWER: **Response 4 is not a true statement.** The Options Schema specifies widget parameters.

QUESTION: Which of the following are widget Server Script Global Objects? More than one response may be correct.

1. *current*
2. *widget*
3. *data*
4. *input*
5. *options*

ANSWER: **The correct responses are 3**, **4**, and **5**. Widgets do not have a *current* or *widget* object.

QUESTION: Which of the following are true statements about widget clones? More than one response may be correct.

1. Clones are created every time a widget is added to a portal
2. Clones are copies of widget records
3. Baseline widgets cannot be edited and must be cloned to make changes
4. Clones are read-only
5. Any widget can be cloned

ANSWER: **The correct responses are 2, 3** and **5**.. When widgets are added to a portal, a widget instance, and not a clone, is created. Clones are editable and are not read-only.

QUESTION: Which one of the following best describes the Client Script global function *this.server.update()*?

1. Calls the server and posts *this.data* to the Server Script
2. Calls the server and automatically replaces the current options and data from the server response
3. Retrieves the options used to invoke the widget on the server
4. Calls the Server Script and passes custom input
5. Retrieves the serialized *data* object from the Server Script

ANSWER: **The correct response is 1**. The *this.server.refresh()* Client Script global function calls the server and automatically replaces the current options and data from the server response. The *this.server.get()* Client Script global function calls the Server Script and passes custom input. The Client Script global object *data* is the serialized *data* object from the Server Script. The Client Script global object *options* contains the options and values used to invoke the widget on the server.

QUESTION: Which of the following are classes in the Widget API? More than one response may be correct.

1. *recordWatch*
2. *spUtil*
3. *spModal*
4. *spAriaUtil*
5. *GlideSPScriptable*

ANSWER: **Responses 2**, **3**, **4**, and **5 are correct**. recordWatch is a method in the spUtil class.

QUESTION: Which of the following are widget debugging strategies? More than one response may be correct.

1. Log to console: $scope.data
2. *console.log()*
3. *spModal.alert()*
4. *spUtil.addTrivialMessage()*
5. Third-party browser-based debugging tools

ANSWER: **All of the responses are correct**.

QUESTION: True or False? When multiple widgets are on a page, widget controllers are siblings.

ANSWER: **It depends**. This is a trick question because there is not enough information to know for sure. In most cases, the statement is true. In the case where one widget is embedded in another widget, the statement is false. Why does this matter? When emitting and listening for events, you need to know the relationship between the widgets.

QUESTION: Which of the following are true about widget options? More than one response may be correct.

1. Setting widgets option values affects all instances of a widget
2. Widget options are identical for all widget types
3. Portal users can change the widget option values
4. Developers can add widget options to the widget option schema
5. Reference is a valid data type for a widget option

ANSWER: **The correct responses are 4** and **5**. Setting widget options affects only one instance of a widget. Every widget type has its own widget options schema. Only developers can change the widget option values.

QUESTION: Which one the following best describes an Angular Providers?

1. Directive
2. Controller
3. Widget API Class
4. Event
5. Function

ANSWER: **The correct response is 1**. Angular Providers are extended HTML attributes known as directives.

QUESTION: Which the following are true statements about Record Watch? More than one response may be correct.

1. Notifies tables when records are changed by widgets
2. The *recordWatch()* method is part of the *spUtil* Widget API class
3. Is automatically part of a widget#singleQuotes logic
4. Registers a listener in a widget
5. Notified when table records are inserted, updated, or deleted

ANSWER: **The correct responses are 2**, **4**, and **5**. Record Watches notifies widgets when table records are changed. Developers must add Record Watch to a widget#singleQuotes logic.

ARTICLE (34 OF 35)

Creating Custom Widgets Module Recap

Core concepts:

- Baseline widgets are read-only; to edit a baseline widget create a copy, known as a clone
- The Widget Editor panes can be shown/hidden
    - HTML Template
    - CSS - SCCS
    - Client Script
    - Server Script

    - Link Function
    - Demo Data (JSON)

- The Server global objects are:

    - ***data***: object containing property/value pairs to send to the Client Script
    - ***input***: Data object received *from* the Client Script's controller
    - ***options***: The options used to invoke the widget on the server

- The Client global objects are:

    - ***data***: The serialized data object from the Server Script
    - ***options***: The options used to invoke the widget on the server

- The Client Script Global Functions are:

    - ***this.server.get()***: Calls the Server Script and passes custom input
    - ***this.server.update()***: Calls the server and posts this.data to the Server Script
    - ***this.server.refresh()***: Calls the server and automatically replaces the current options and data from the server response

- The Widget API includes:

    - *spUtil* (client)
    - *spModal* (client)
    - *spAriaUtil* (client)
    - *GlideSPScriptable* (server)

- Test Widgets

    - Preview
    - Test Page
    - JavaScript console

- Directives are reusable, extended HTML attributes

    - Defined as Angular Providers
    - Must be attached to a widget as a dependency

- The Widget Option Schema defines user-settable options for a widget

    - Options are set on a widget instance basis
    - Options are loaded into the options object

- Record Watch monitors a table for record changes which occur outside of Service Portal

ARTICLE (35 OF 35)

After Completing Creating Custom Widgets, You Might Also Be Interested In...

Congratulations on completing the Creating Custom Widgets module. Based on your interest in creating widgets, you might also enjoy:

- **Developing Virtual Agent Topics** (http://developer.servicenow.com/to.do?u=CCW-U-MOD-VIA): In this Developer Site learning module, you will learn how to use Virtual Agent to automate user interactions. Virtual Agents can be included in service portals..
- **Service Portal Documentation** (http://developer.servicenow.com/to.do?u=CCW-U-MOD-DOC-SPLanding): On the ServiceNow docs site, you will find the complete set of reference material for Service Portal.