knowledge17

May 7–11, 2017 • Orange County Convention Center • Orlando, FL

Lab Guide

Configuring and Deploying an Engaging Service Portal

Alex Steinwachs & Chris Dilley

Default Login / Password:

admin / p@ssword

Lab Sponsored by Microsoft Azure

Microsoft

servicenow

This
Page
Intentionally
Left
Blank

Lab Goal

Lab 1.0 Service Portal Overview

This lab reviews the Service Portal content from an administrative perspective. The lab explains how to perform the following tasks:

- Create new service portals and pages
- Design using no-code and low-code techniques
- Add and modify menu Items

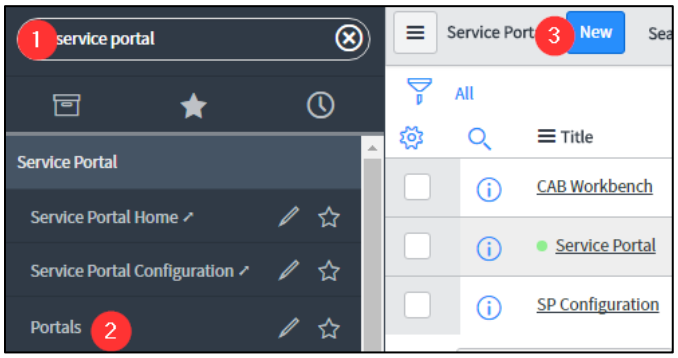
The goal of this lab is to review the Service Portal and its components from the administrative view. It is important to be on at least Helsinki Patch 6 to complete this lab.

You can download the images for this lab from http://bit.ly/2nd8atu.

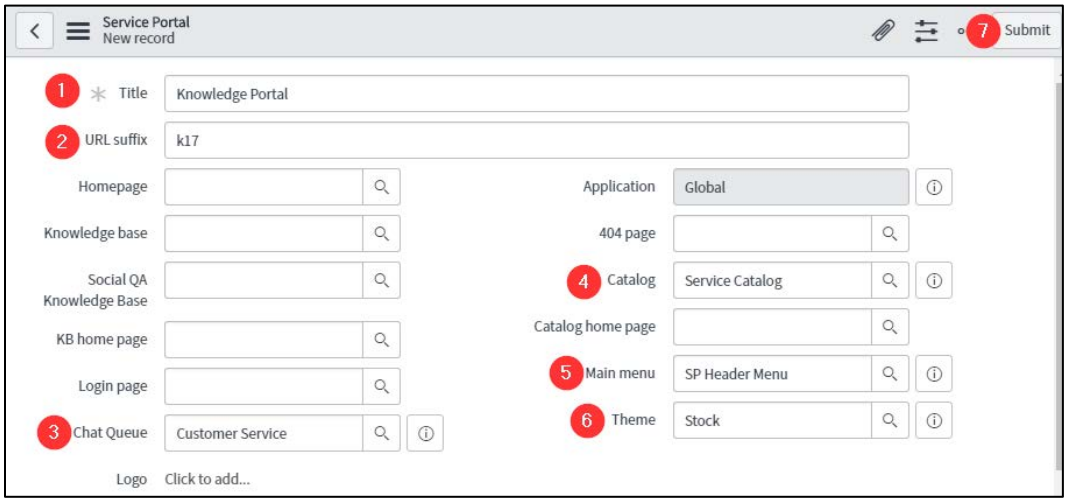
Create a New Service Portal

1. Log in to the lab instance provided.

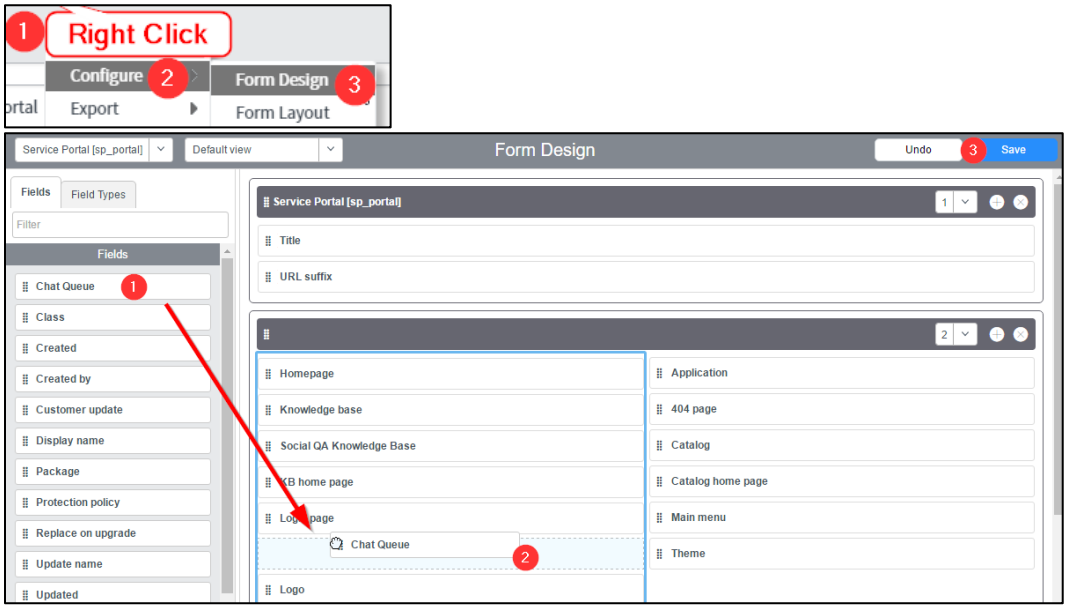
2. Using the search in the upper-right corner, navigate to **Service Portal > Portals**. Click **New**.



3. Create a new Portal using the following Values. Click **Submit**.

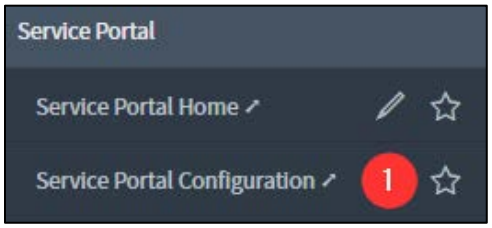


Note: To see **Chat Queue** you might need to add it to the form. If no Chat Queues are available, create one.

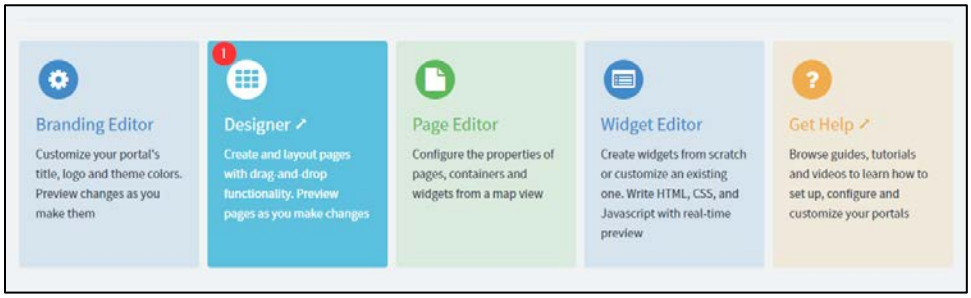


Create a New Portal Page

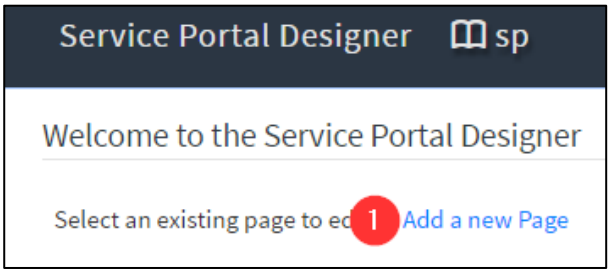
1. Navigate to **Service Portal > Service Portal Configuration** ↗. A new tab opens.



2. Click **Designer** ↗. A new tab opens.



3. Click **Add a new Page**. You can also click the **+** tile at the bottom of the page.

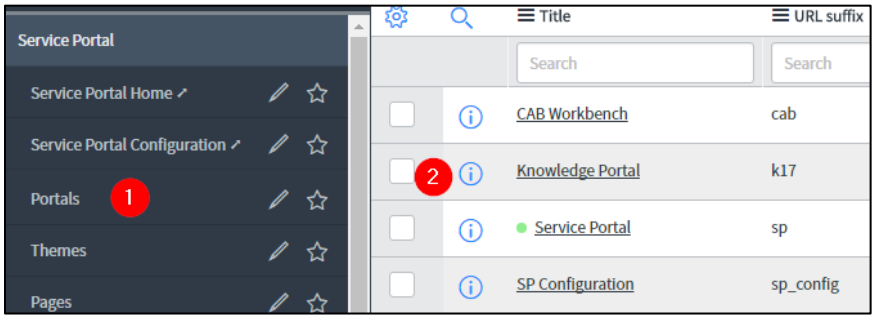




4. Give it a Page Title of **Knowledge17 Index**. **Page ID** auto populates. Click **Submit**.



5. Navigate to the **main ServiceNow window> Service Portal > Portals > Knowledge Portal**.



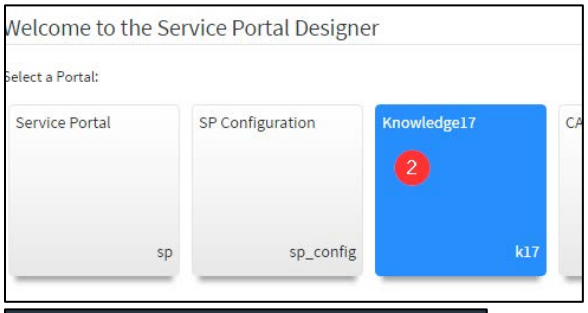
6. Assign the page just created to the Homepage field. Click **Update**.



Design the Portal Page

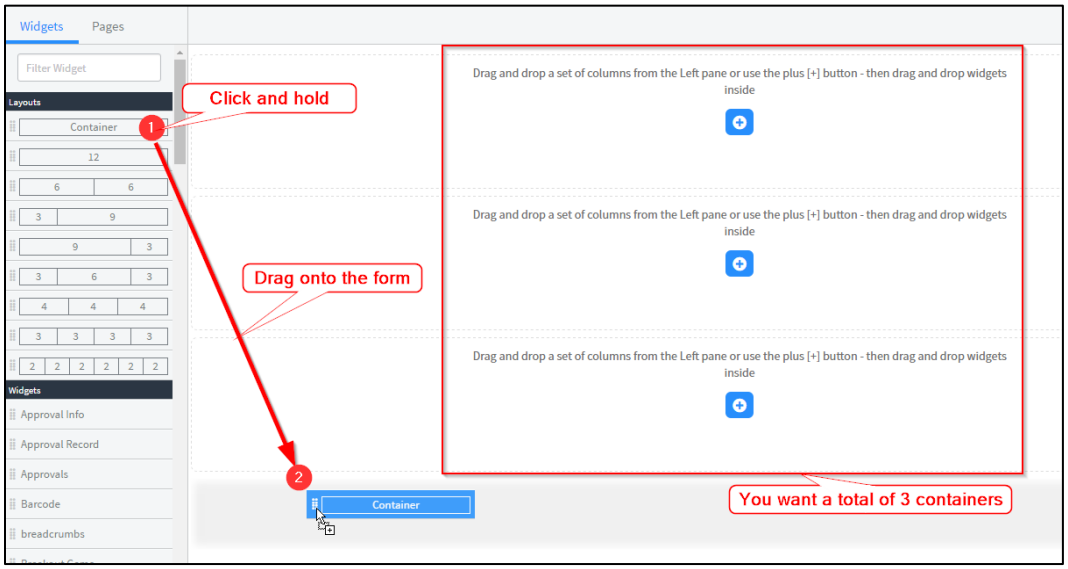
1. Navigate back to the **Page Designer Tab**. Change the **Portal** from **sp** to **k17** and choose the **Knowledge17** page.





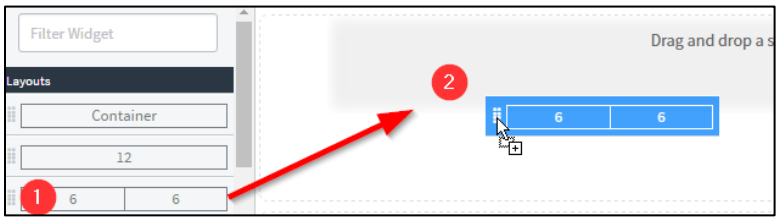


2. Create three containers by dragging the **Container** object under **Layouts** onto the form.

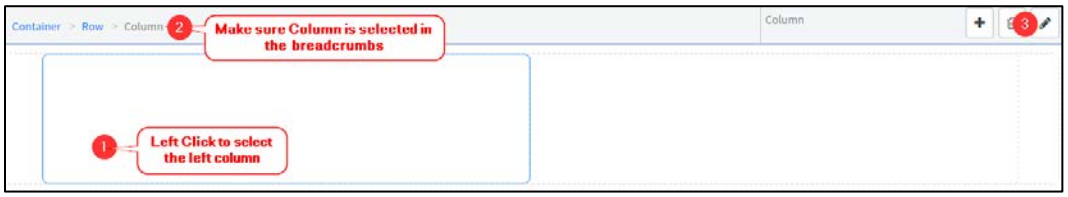


For the Top Container:

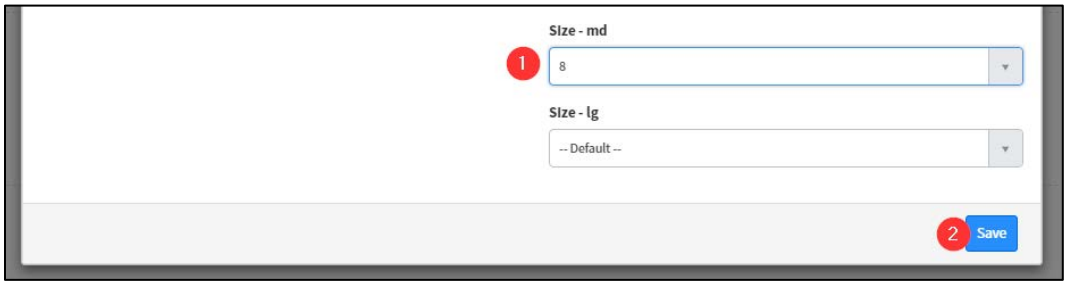
1. Drag a **6|6** layout onto the container.



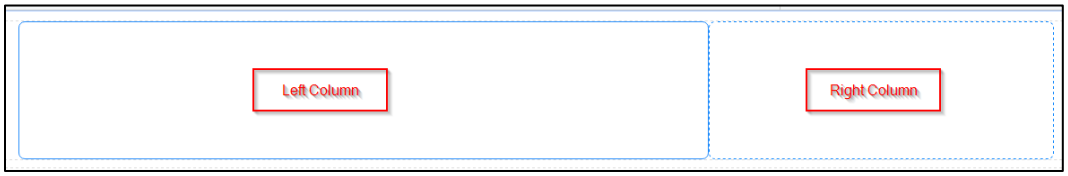
2. Select **the left column** then click the **pencil** in the upper-right corner to **edit** the properties.



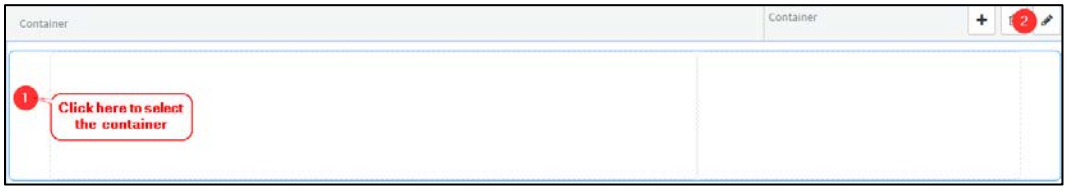
3. Change **Size - md** from 6 to 8. Click **Save**.

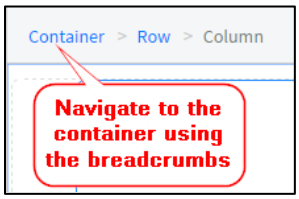


4. Change the right column **Size - md** to **4.** Your columns should look like the example:

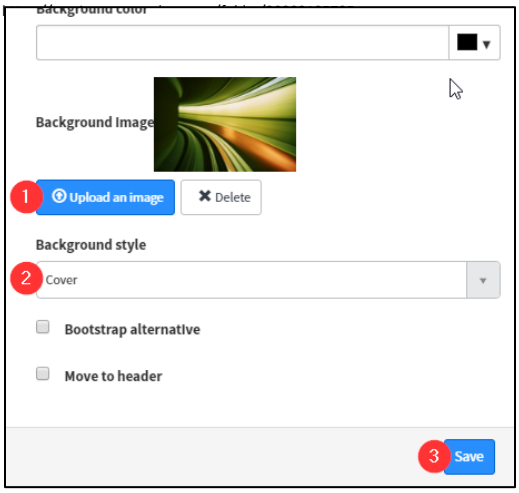


5. Select the container (not the columns or rows) and edit the properties. If you get stuck, use the breadcrumbs to navigate to the container from the column.





6. Download the **background.jpg** from the shared folder and then upload it as a **Background Image**. Set the Background Style to **Cover**. Click **Save**.



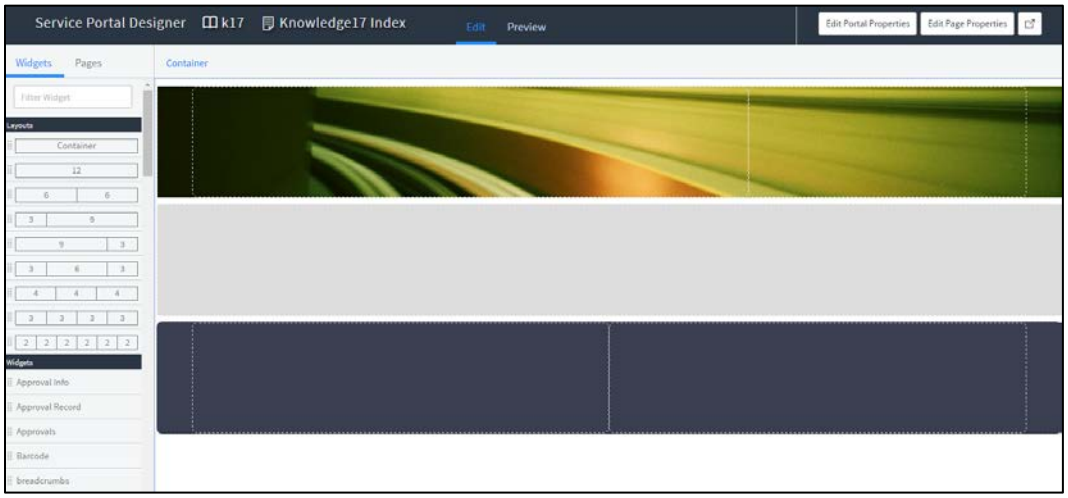
For the Middle Container:

1. Set to 4 columns, each 3 wide (**3|3|3|3**)
2. In the Container properties (pencil), set the Background color to **#DDDDDD**.

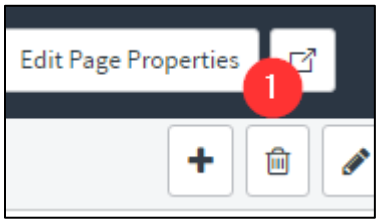
For the Bottom Container:

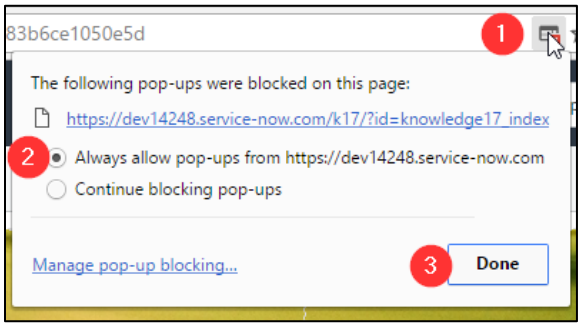
1. Set to 2 columns, each 6 wide (**6|6**).
2. In the Container properties (pencil), set the Background color to **#343d47**.

3. In the Container properties (**pencil**), set the Parent class to **navbar-inverse**. Your portal should look like this in the Designer:

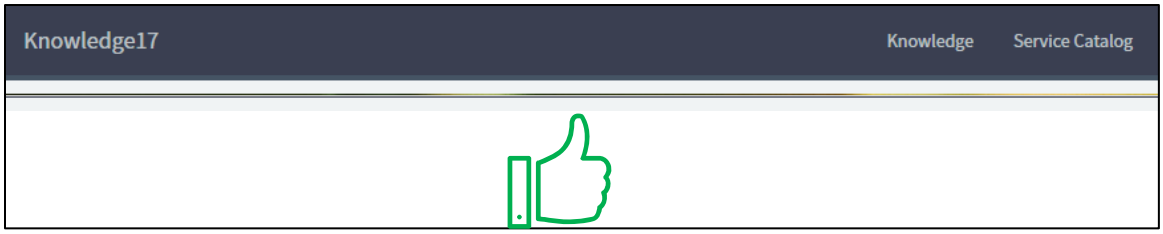


Note: You can preview the portal be selecting the button in the top-right corner. If nothing opens, allow pop-ups for this site.



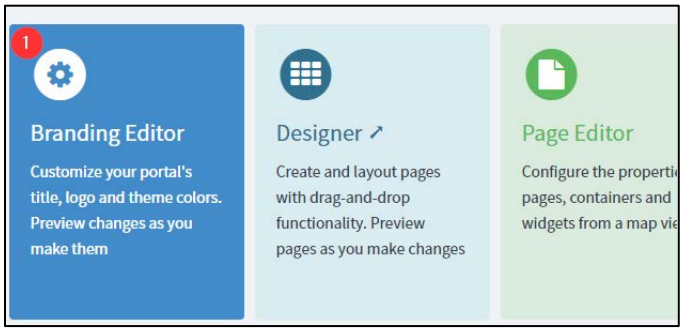


Note: If you preview the page at this point, you will not see anything. Just the containers are in place; there is no content to expose the backgrounds and image you placed. Once you add the widgets the page comes to life.

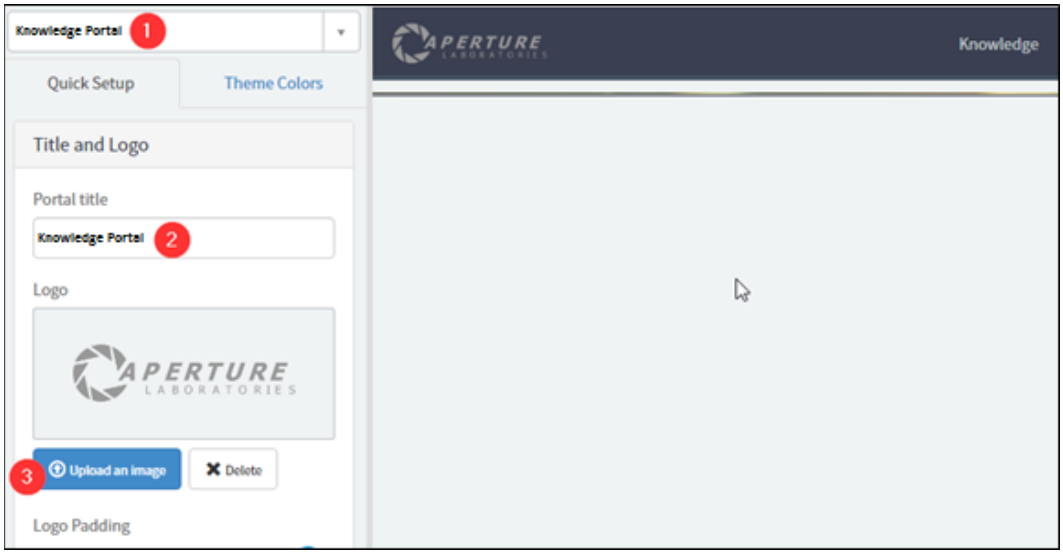


Brand a Portal

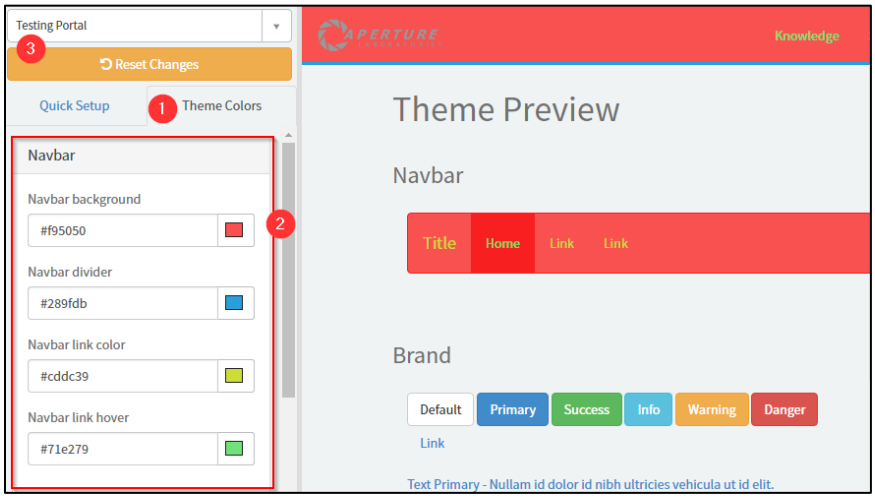
1. Navigate to the Service Portal Configuration ↗ **tab > Branding Editor**.



2. Select **Knowledge Portal** in the top-left if it is not already selected. Change the Portal Title and upload a Logo with the **logo.png** from the shared folder or your own company logo.



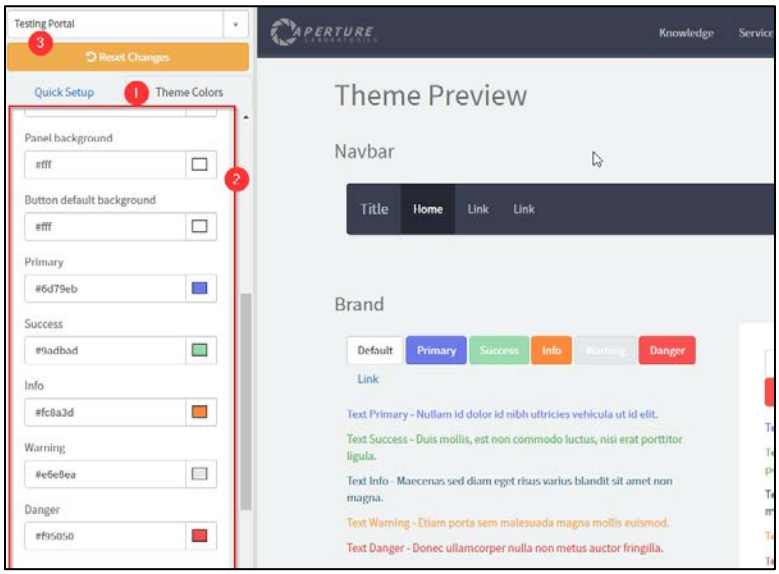
3. Change the Navbar colors. Watch the preview refresh after each change. Click **Reset Changes** to undo the changes.



Note: You can also modify the CSS values from the main ServiceNow interface (**Service Portal > Portals > Knowledge Portal > CSS** variables). Here are the default values:

```
$brand-primary: #428bca;
$body-bg: #fff;
$state-success-text: #5cb85c;
$sp-homepage-bg: #fff;
$sp-navbar-divider-color: #455464;
$text-color: #717171;
$sp-logo-margin-y: .5rem;
$navbar-inverse-link-hover-color: #ffffff;
$panel-bg: #fff;
$brand-success: #5cb85c;
$navbar-inverse-bg: #3a3f51;
$brand-warning: #f0ad4e;
$brand-danger: #d9534f;
$sp-tagline-color: #717171;
$navbar-inverse-link-color: #bdc0c4;
$brand-info: #5bc0de;
$link-color: #428bca;
$sp-logo-margin-x: 1rem;
$btn-default-bg: #fff;
$text-muted: #979797;
```

4. You could also modify the Default, Primary, Success, Info, Warning, and Danger colors. These colors propagate to all the bootstrap color values (we use these colors in the next lab). Click **Reset Changes** to undo the changes.



Add a Menu Item

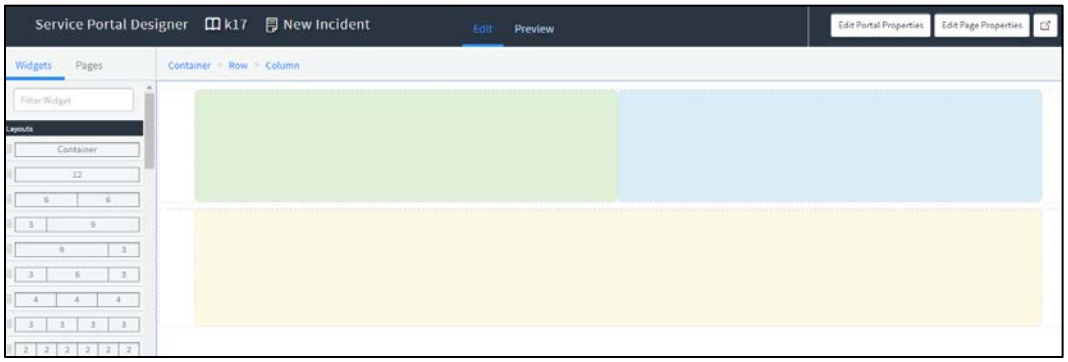
1. **Create** a **New Incident** Portal Page. (You already learned how to create a new page in the **Create a New Portal Page** section of this).



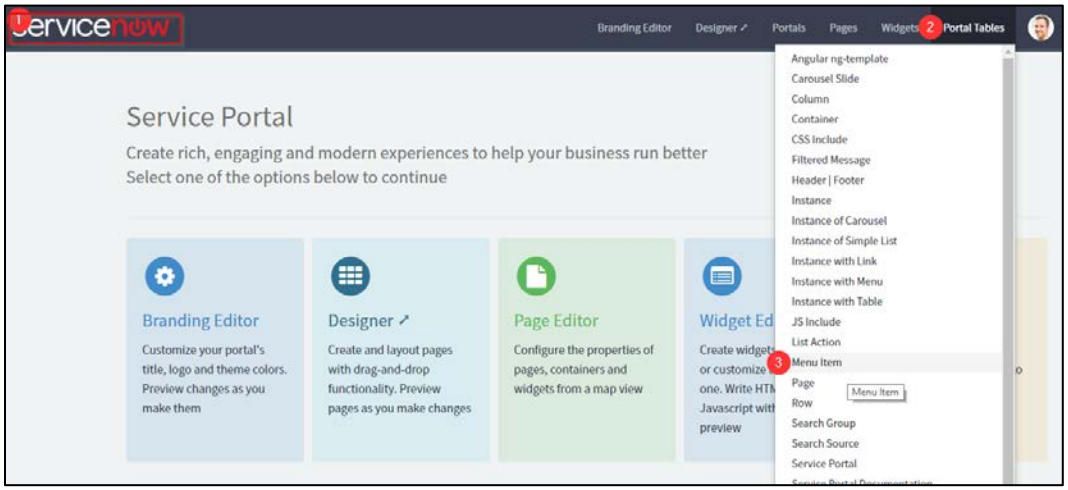
2. In the **Page Designer** open the new page. Add 1 additional container (for a total of 2).

3. In the Top Container apply a **6|6** layout. Give the left column a CSS class of **bg-success** and the right column a CSS class of **bg-info**.

4. In the bottom Container, apply a **12** layout. Give the only column a CSS class of **mark**.



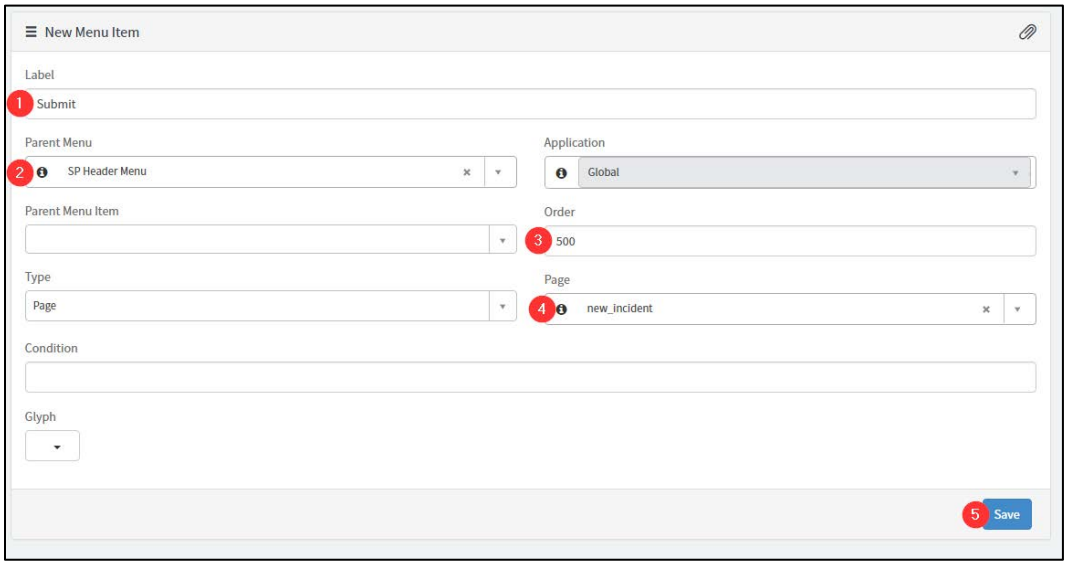
5. Navigate to the **Service Portal Configuration tab > Portal Tables > Menu Item**.



6. Click **New**.

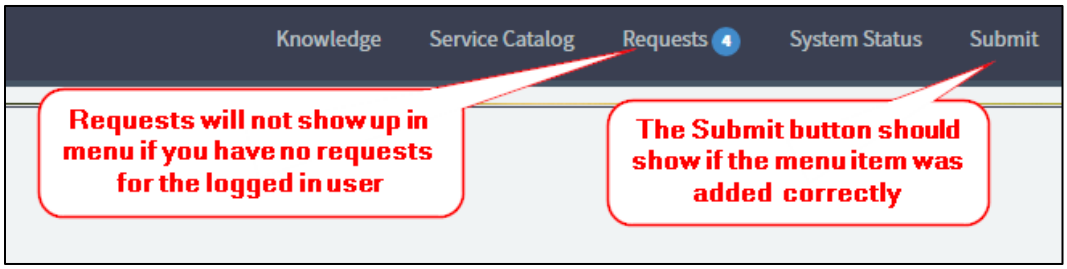


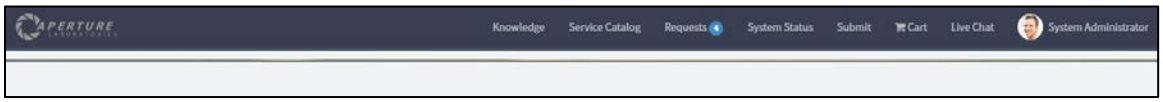
7. Create a new **Menu Item** with the following values and click **Save**.



Lab Verification

1. Navigate to https://instance.service-now.com/k17 to view the Portal. The header looks like the example.





Note: The page looks blank because you have not added any widgets yet. If a row has no visible widgets, it will automatically collapse.

Lab Goal

Lab 2.0 Service Portal Widget Overview

This lab reviews the Service Portal Widget content from an administrative perspective. Subject areas include the following:

- Adding widgets to a page
- Configuring widget instance options
- Dynamically show or hide widgets based on device

The goal of this lab is to add and configure widgets on an existing page.

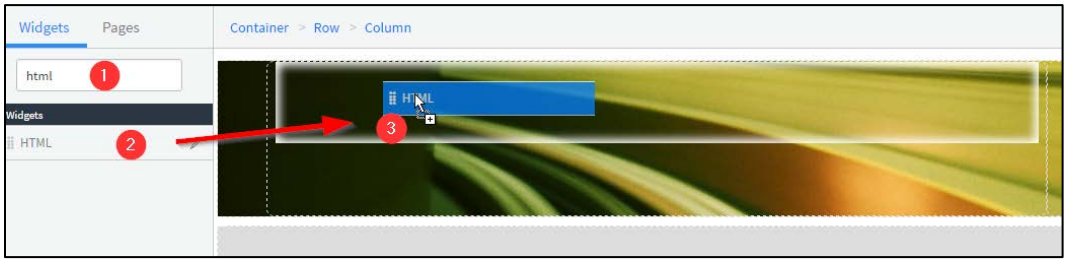
Add Widgets to a Page

1. Navigate to **Service Portal > Service Portal Configuration ➚ > Designer ➚**.

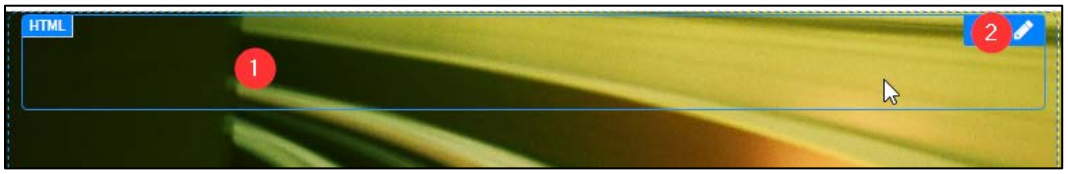
2. Select the **Knowledge17 Index** page and the **k17** portal.

For the Top Container:

1. Search for the **HTML** widget in the top-left **Filter Widget** field; then drag the **HTML Widget** into the left column.



2. Select the widget and modify the properties.



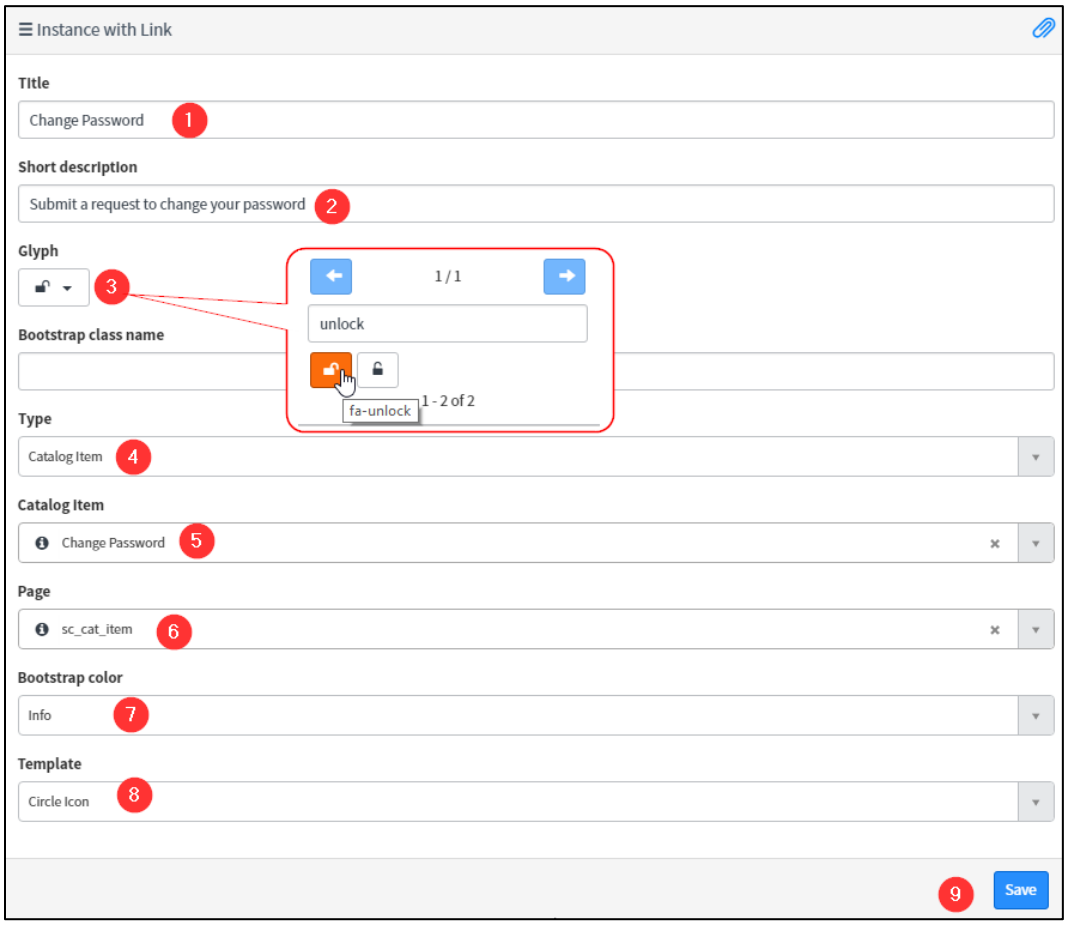
3. Give the HTML field a value of **Welcome!** using Heading 2 as the font. Click **Save**.

4. Drag the **Simple List** widget into the right column; then modify the properties. Select Knowledge **[kb_knowledge]** for Table and choose **Short Description** for the Display field. Click **Save**.

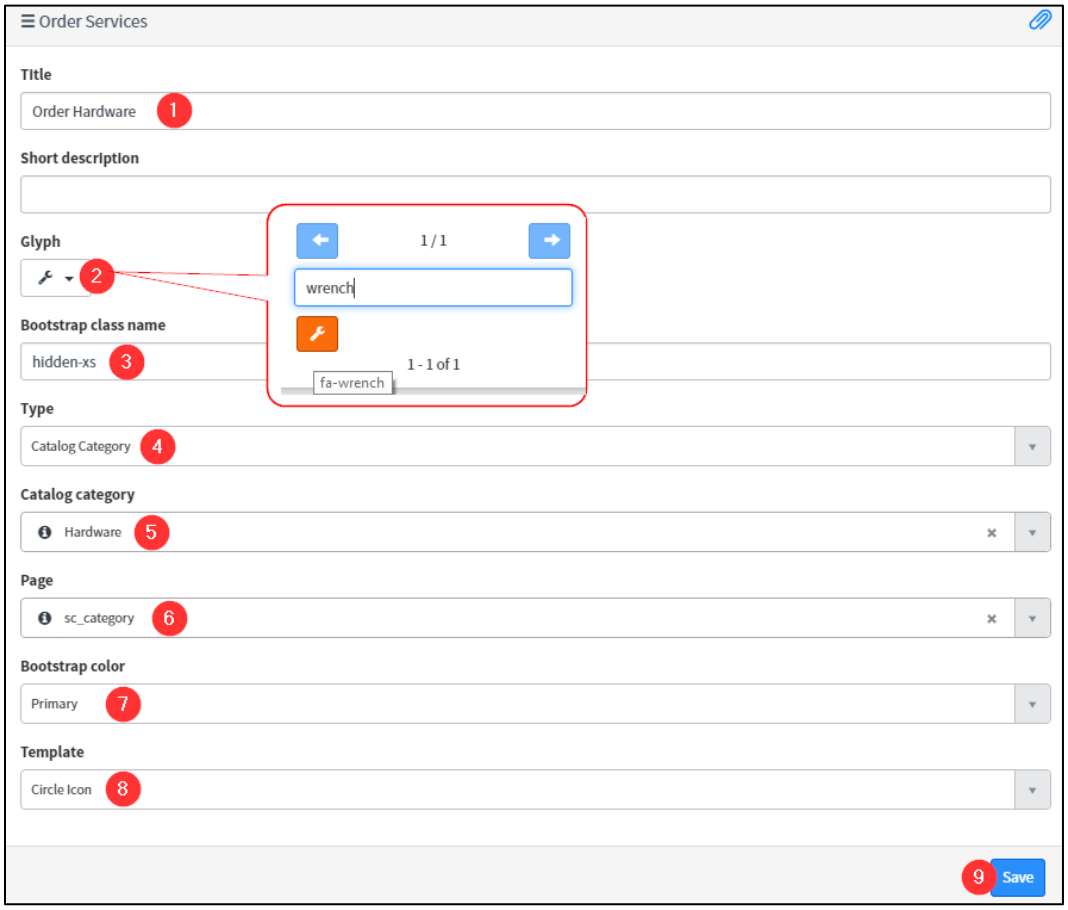
For the Middle Container:

1. Drag the **Icon Link** widget into each of the 4 columns.

2. First Icon (left-most) properties:

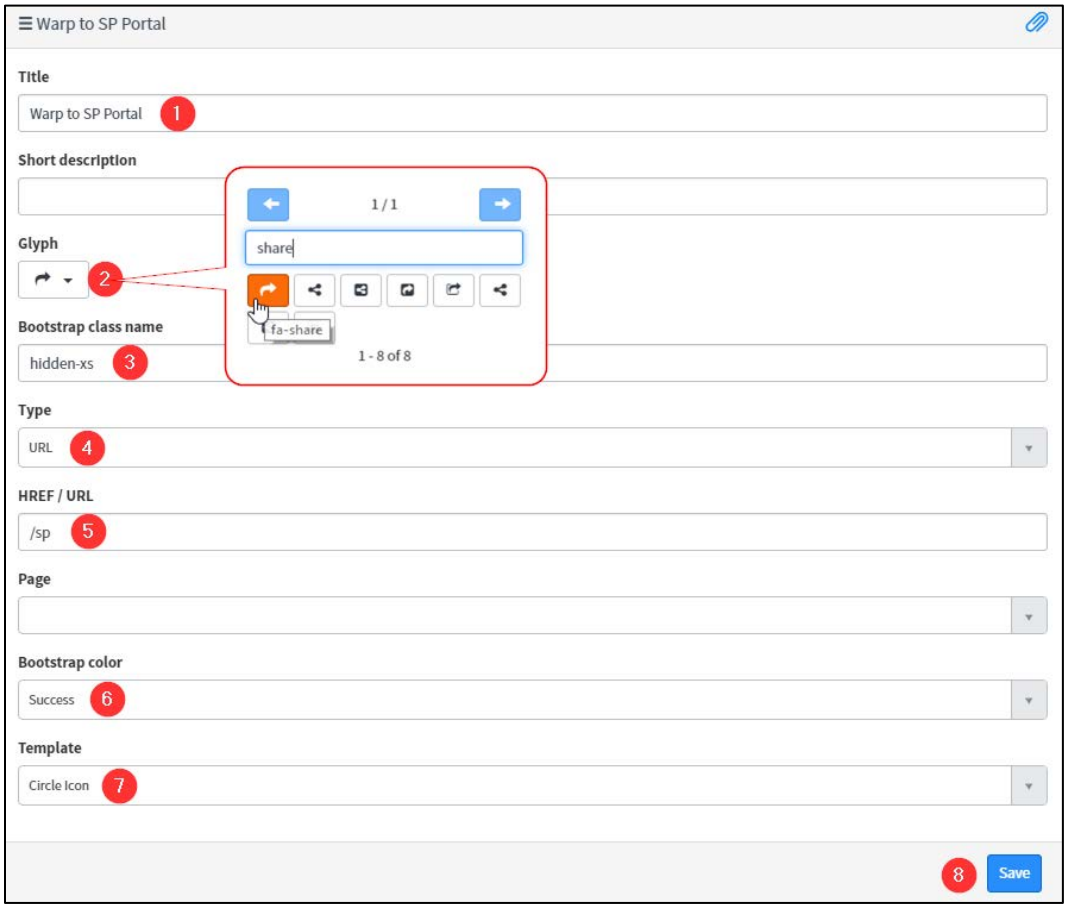


3. Second Icon properties:



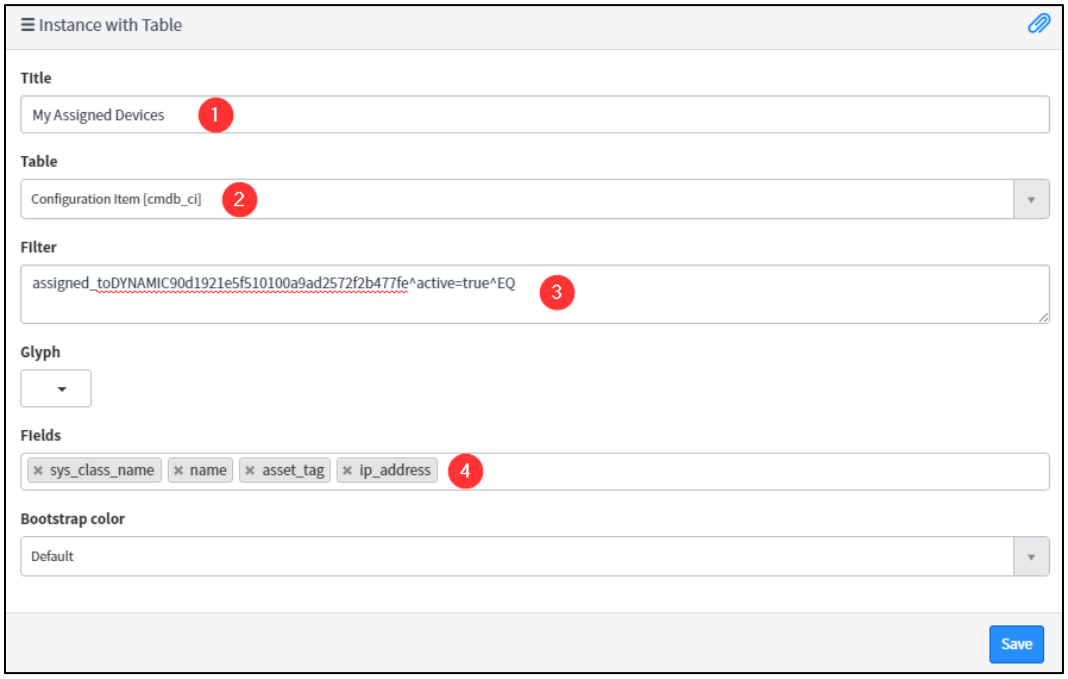
4. For the Third Icon, make a link to the **Requests** page with **Top Icon** for the Template, **rebel** for the Glyph and **Danger** for the Bootstrap color.

5. Fourth Icon properties:

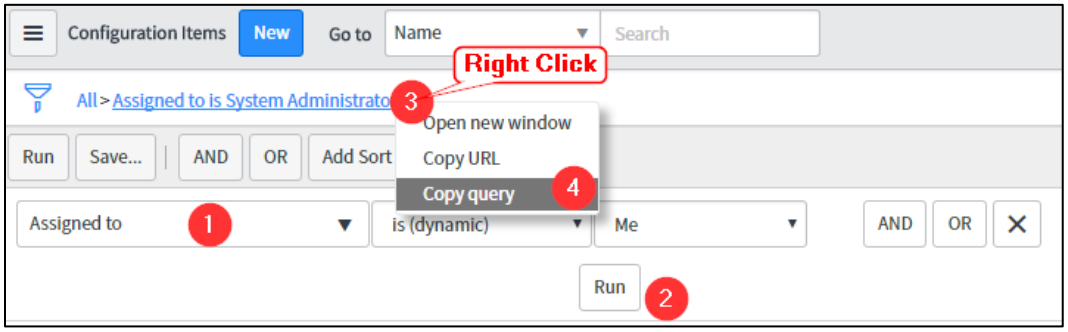


For the Bottom Container:

1. Drag the **Data Table** from Instance Definition widget into the left column. Select the **Configuration Item [cmdb_ci]** for the Table, **Class (sys_class_name), Name**, **Asset Tag**, **IP Address** for the Fields, and **assigned_toDYNAMIC90d1921e5f510100a9ad2572f2b477fe** for the Filter.

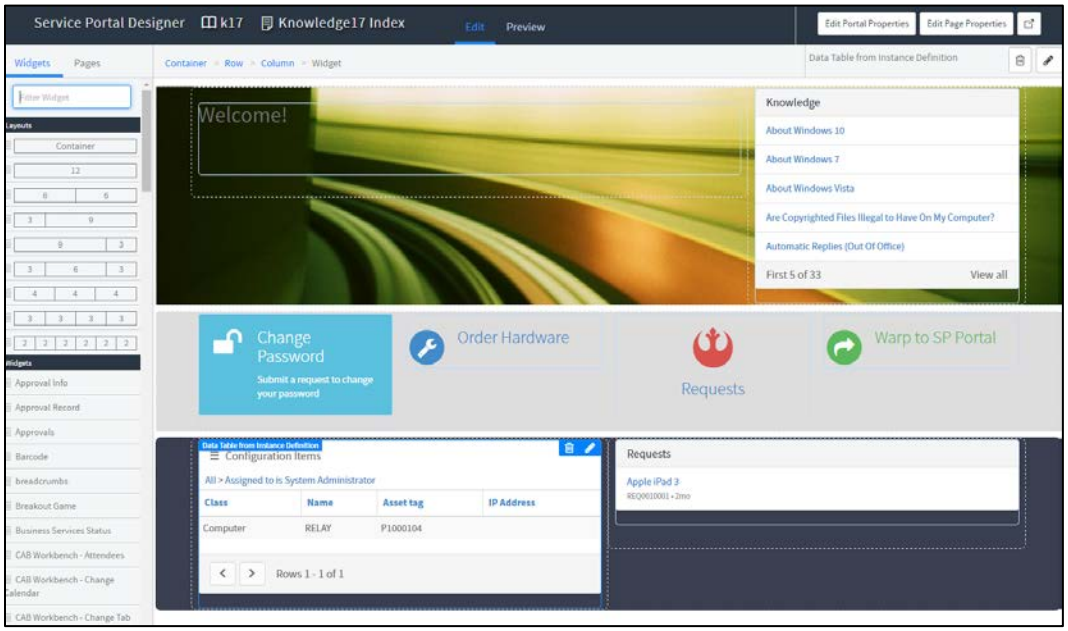


Note: You can get the filter by navigating to the table in the main ServiceNow interface, creating a query, right clicking on the breadcrumb, and selecting copy query to clipboard. Paste that into the **Filter** field.

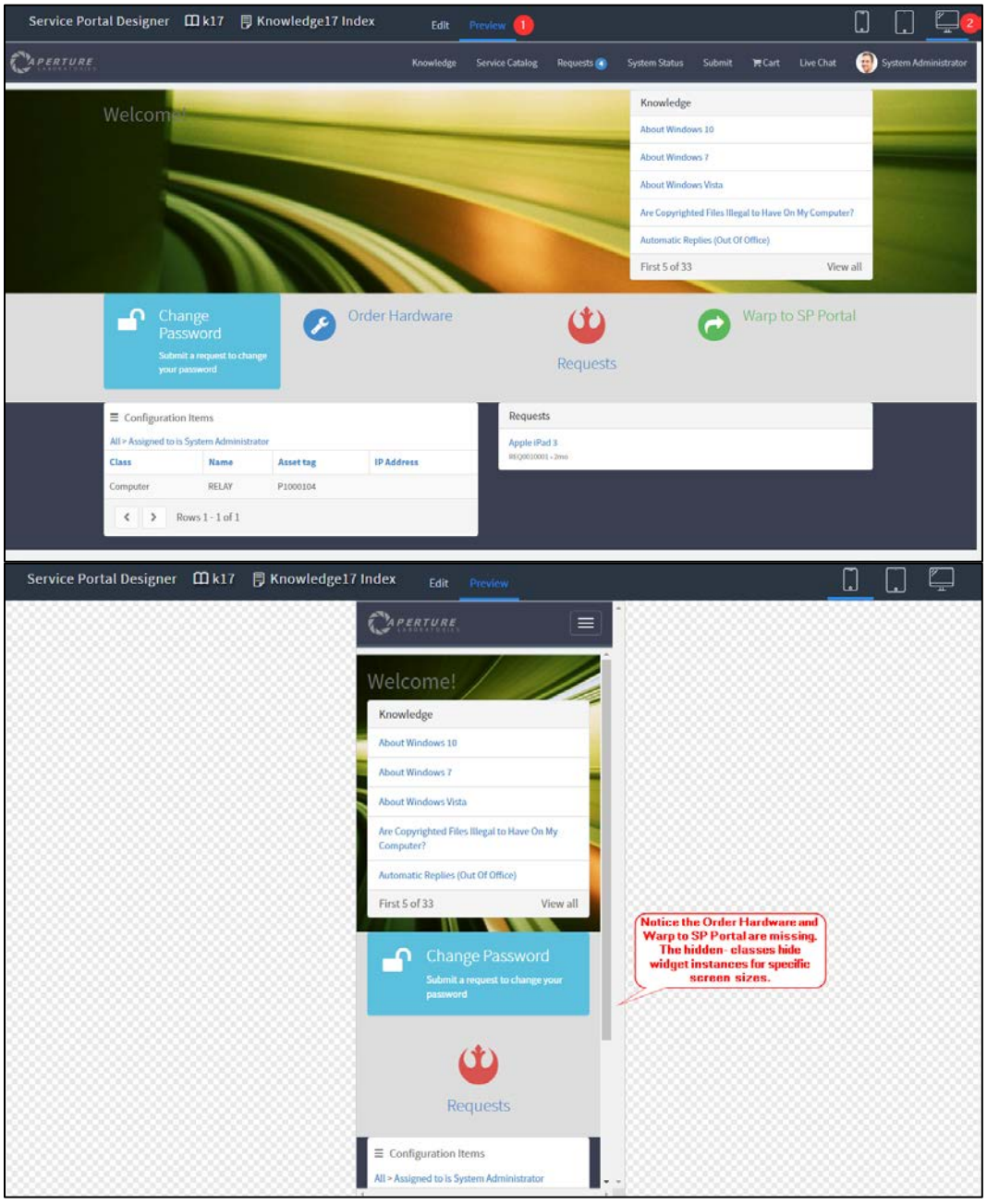


2. Drag the **My Requests** widget into the right column.

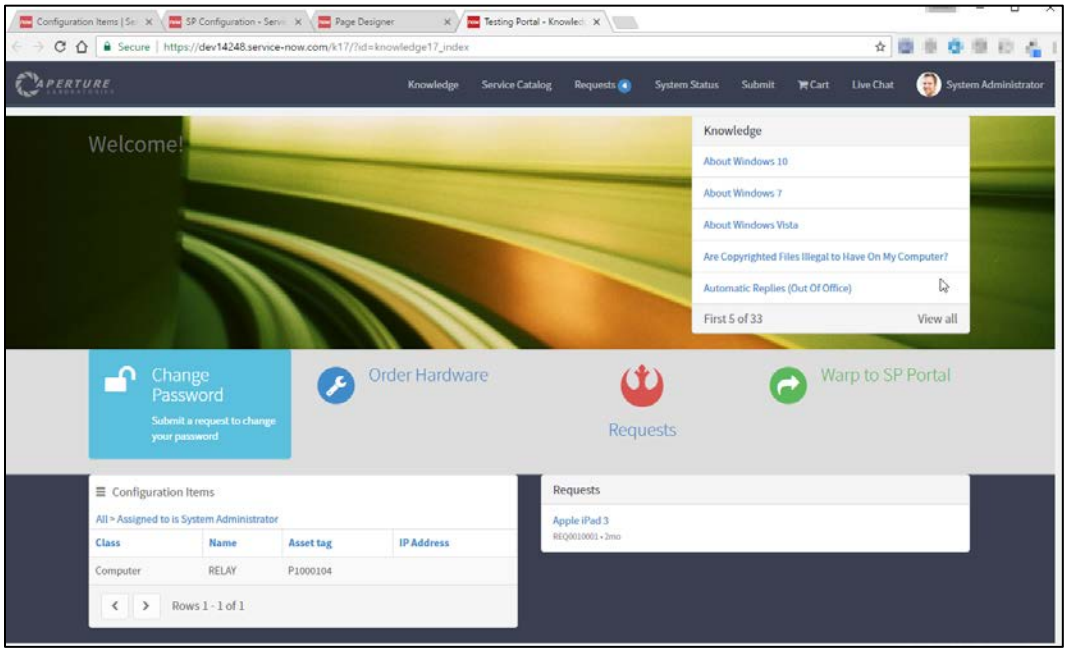
3. The page should look like this in Edit View:



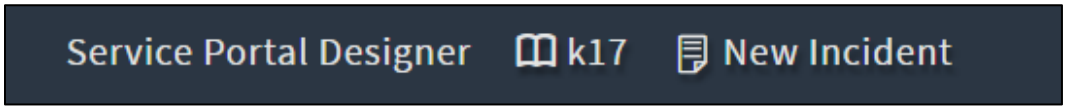
4. Click **Preview** to view the new portal in Desktop, Tablet, and Mobile views.



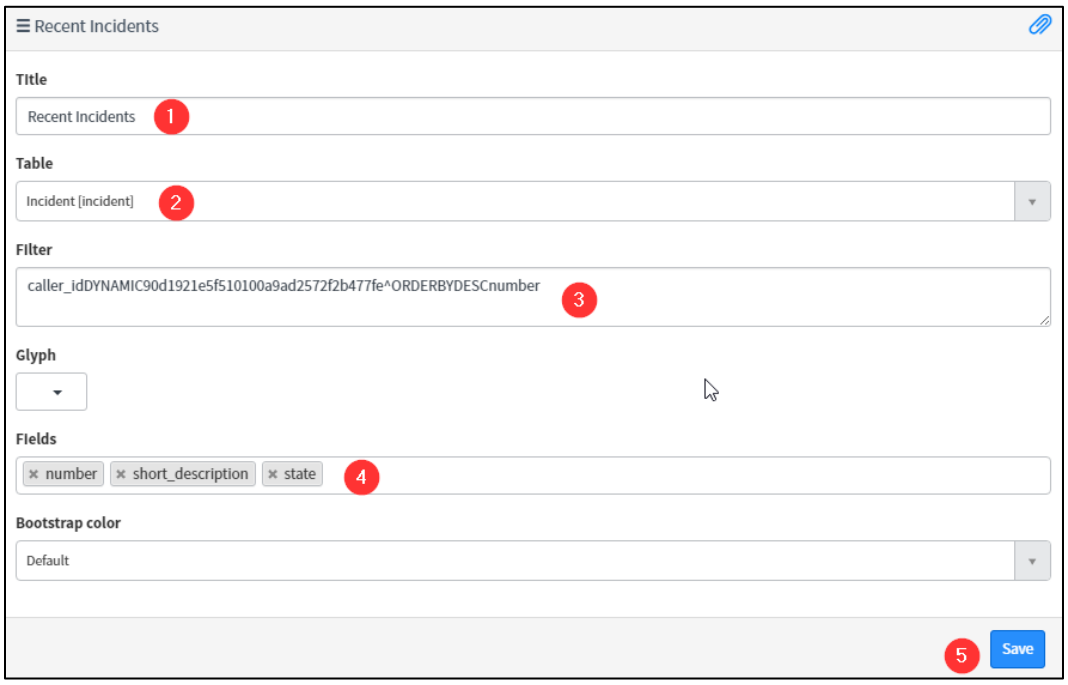
5. Click the **top-right button** (next to Edit Page Properties) to view the live portal. Return to the Edit mode tab.



6. Change the page to **New Incident** in the Designer.



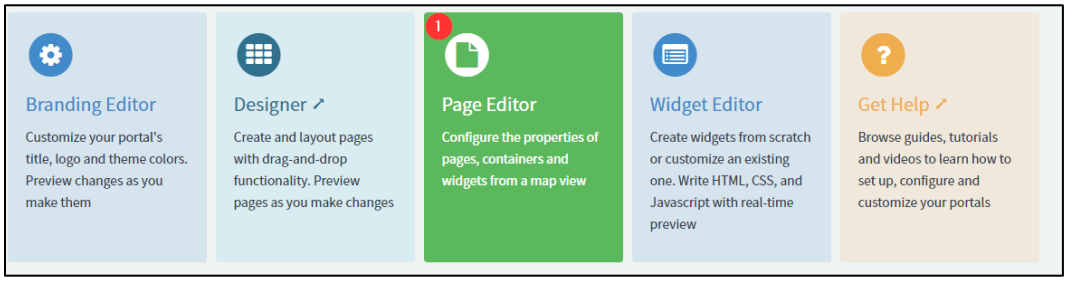
7. Drag the **Data Table from Instance Definition** widget into the top-left column. Click the pencil to edit the widget instance, enter the following values, and click **Save**.
   Title: **Recent Incidents**
   Table: **Incident**
   Filter:
   caller_idDYNAMIC90d1921e5f510100a9ad2572f2b477fe^ORDERBYDESCnumber
   Fields: **number, short_description, state**



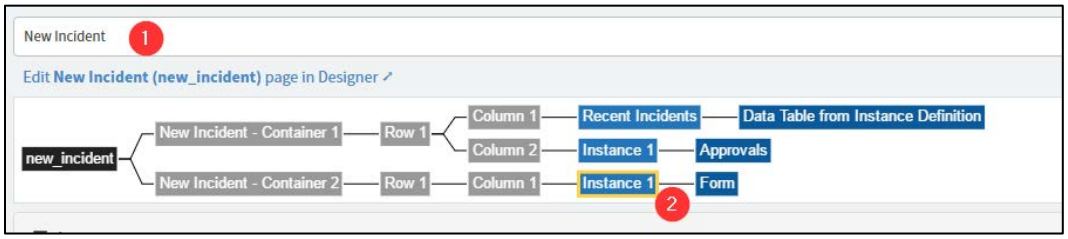
8. Drag an **Approvals** widget into the top-right column.

9. Drag the **Form** widget into the bottom container. There are 4 hidden options that you can modify.

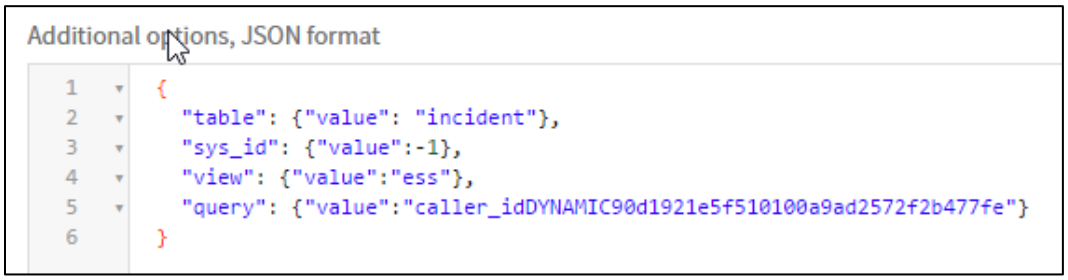
10. Navigate to **Service Portal > Service Portal Configuration ↗ > Page Editor.**



11. Select the **New Incident** page. Then select **Instance 1** in Container 2 – Row 1 – Column 1.



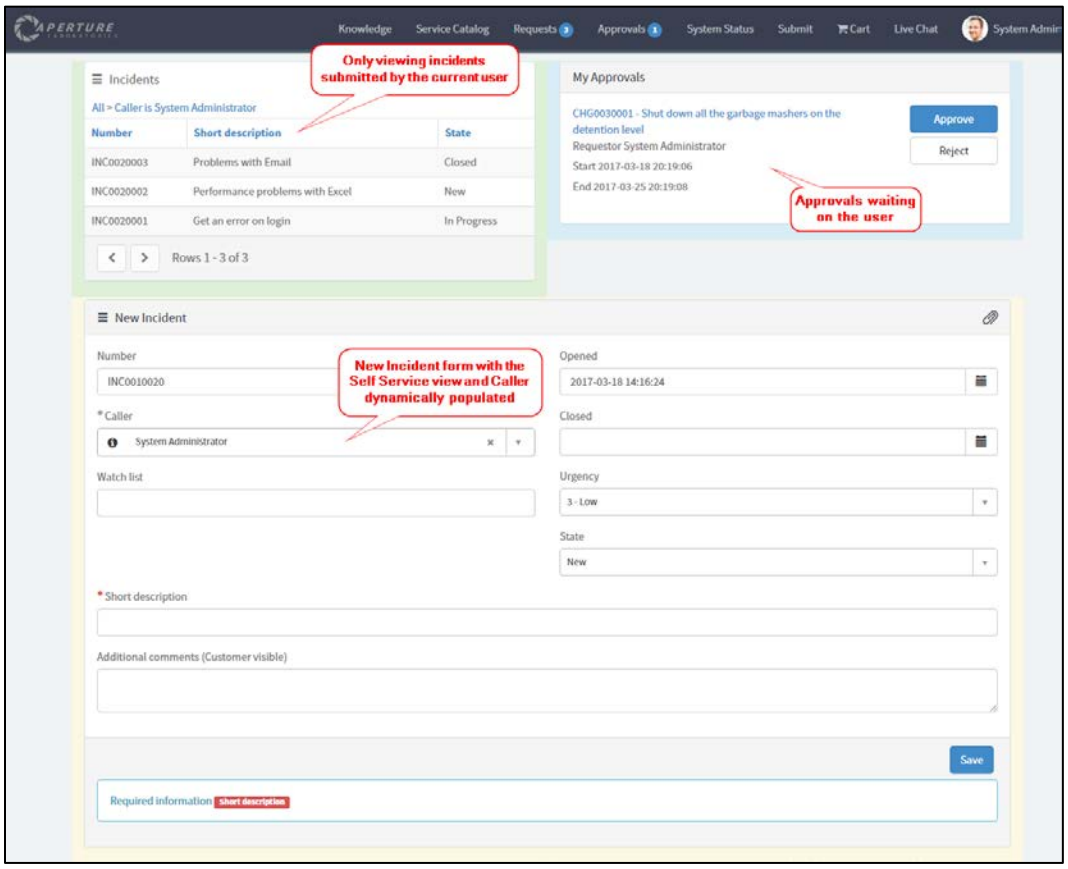
12. Replace the existing values in the **Additional options, JSON format** field with the following.




```
{
  "table"    : {"value": "incident"},
  "sys_id" : {"value":-1},
  "view"     : {"value":"ess"},
  "query"   :
{"value":"caller_idDYNAMIC90d1921e5f510100a9ad2572f2b477fe"}
}
```

https://pastebin.com/3HcDXtCX

13. Your page should look like the example. (Click **Submit** in the header.)



Lab Goal

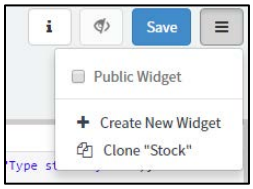
This lab explains how to clone an existing OOB widget (Stock) and apply some basic HTML/CSS changes (no scripting). It also explains how to abstract some features by using Widget Options. This lab is just a generic (and very simple) example to familiarize you with widget concepts.

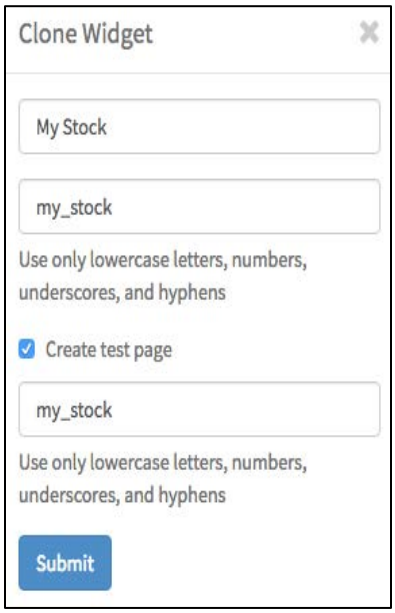
Lab 3.0 Custom Widgets

Clone Widgets

1. Navigate to **Service Portal > Service Portal Configuration ↗ > Widget Editor.** Select the **Stock** widget.

2. Click the menu and choose **Clone "Stock"**. Give the new widget the name **My Stock** and select **Create test page**. Click **Submit**.





Hint: Save the widget by pressing **Ctrl + S** (Windows) or **Command + S** (Mac).

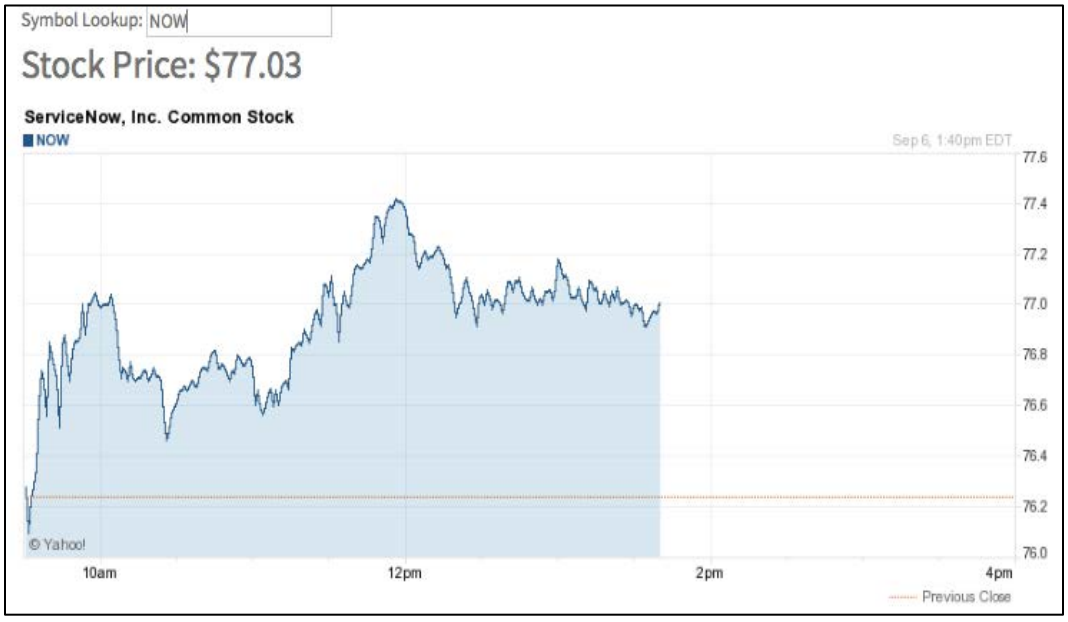
3. Modify the Server Script to look like this then click **Save**.

```
data.typeSymbolMsg = gs.getMessage("Type stock symbol");
if (input) {
   var r = new RESTMessage('Yahoo Finance', 'get');
   r.setStringParameter('symbol', input.symbol);
   var response = r.execute();
   data.price = response.getBody().split(',')[0];
}
```

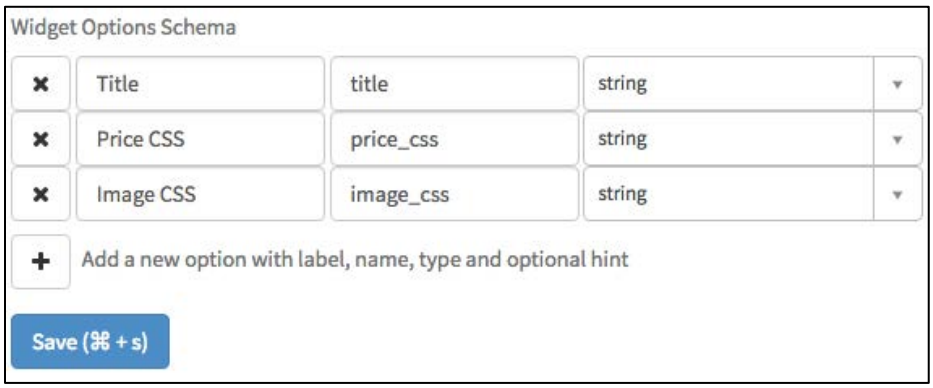
https://pastebin.com/72LnNLgJ

3. Open the test page in a new window or tab without a portal associated. Navigate to **$sp.do?id=my_stock**.

4. The new test page with the cloned widget will look something like the example. Test it by adding a known stock code (e.g., NOW).



5. Navigate to **Service Portal > Service Portal Configuration ↗ > Widget Editor.** Select the **My Stock** widget. Select **Edit option schema** and add three new options. Click **Save** then reload the test page.



To get access to those options within the HTML Template section, use double curly brace notation. Example: **{{::options.title}}**.

6. Now add those newly created options to the HTML template.

Add an **h2** tag to the top of the HTML to display the title.
<h2>{{::options.title}}</h2>

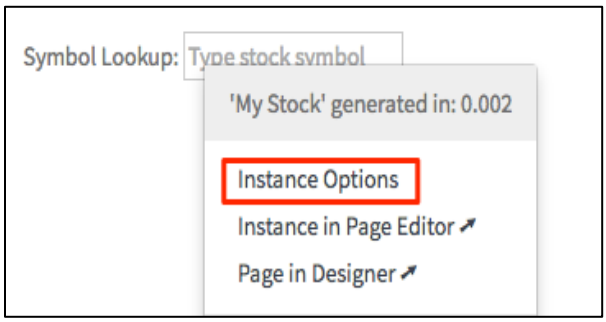
Locate the **span** element where the price is displayed and add a new style attribute with the Price CSS option.
{{c.data.price | currency:"$"}}
Locate the img element where the stock image is displayed and add a new style with the Image CSS option.
****

Click **Save**. Open or reload the test page again and notice how there's not really a big change. Why is that? Well, if you remember, this is just an instance of the widget so the options are there but they have not been set for this widget instance yet.

```
<h2>{{::options.title}}</h2>
  <div>${Symbol Lookup}:
    <input ng-model="c.data.symbol"
       ng-model-options="{debounce: 750}"
       ng-change="c.update()"
       placeholder="{{::data.typeSymbolMsg}}" />
    <div ng-show="c.data.symbol" style="font-size: 2em;">
      <p>${Stock Price}:
         <span ng-if="!c.data.price">${Requesting stock price}</span>
         <span style={{::options.price_css}}>
            {{c.data.price | currency:"$"}}
         </span>
       </p>
      <img style={{::options.image_css}}
ng-src="http://chart.finance.yahoo.com/z?s={{c.data.symbol}}&t=1d&z=l"/>
</div>
</div>
```

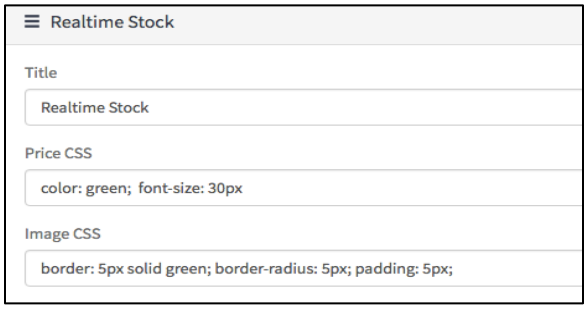
https://pastebin.com/FnTi1Epw

7. On the test page, right-click + CTRL on the widget instance and select Instance **Options**.



8. Enter text for the **Title** and then some CSS style to **Price CSS** and **Image CSS** option. Click **Save**.

   Title: **Realtime Stock**
   Price CSS: color: **green**; font-size: **30px**
   Image CSS: border: **5px solid green**; border-radius: **5px**; padding: **5px**;



9. Open the test page and again test it. This time, open your test page with an actual portal associated (e.g., **sp**). The page should look like the example.



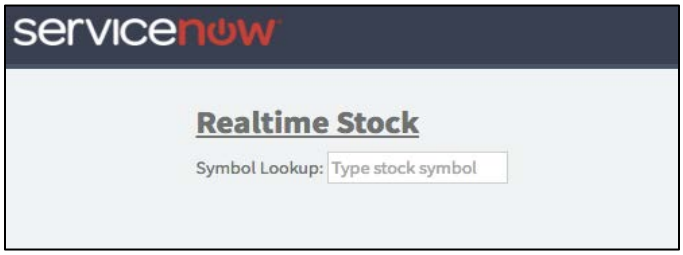
10. Navigate to **Service Portal > Service Portal Configuration ↗ > Widget Editor.** Select the **My Stock** widget. Check the **CSS-SCSS** box to enable editing the CSS of the widget. Add the following CSS then click **Save**.

```
.myTitleClass {
  font-size: 25px;
  color: $sp-tagline-color;
  font-weight: bold;
  text-decoration: underline;
}
```

11. Add the new class to the **h2** element tag. Click **Save**.

```
<h2 class="myTitleClass">{{::options.title}}</h2>
```

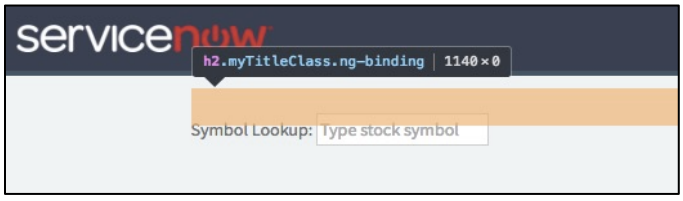
12. Open the test page within a portal (**https://instance.service-now.com/sp?id=my_stock).** The widget should look like the example with the new class applied.



Challenge – Work with Unspecified Options

There may be a situation in which the widget instance does not have 1 or more options set. In this instance, add another check to ensure the option has a value.

1. Modify the instance options and clear the text for the Title. Notice how even though the title is not set, the h2 tag is still rendered and occupies a visible "empty" space on the page.
   Hint: You can use your browser's Inspect functionality for this.



2. Look at how the span element contains the localized text ${Requesting stock price}. This element is uses a special AngularJS directive called **ng-if**. There are alternatives (**ng-hide** and **ng-show**) that could provide comparable results.

3. Using **ng-if** directive, see if you can remove the h2 element altogether from the page if the Title option is not set.

4. The resulting page should look like the example with **NO visible empty space**. Remember to inspect the HTML to check that the h2 tag is no longer there.



Apply Dynamic Classes to Widgets

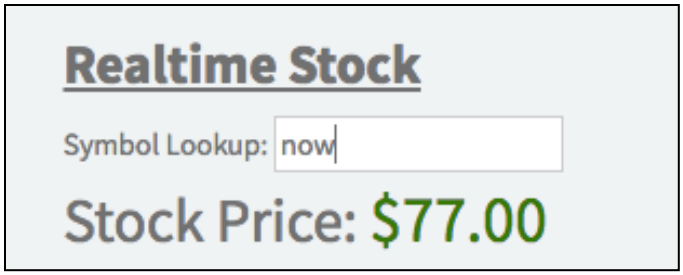
For a challenge, modify the widget so that the price is displayed in **red** if the price is 60 or below or **green** for anything higher. Feel free to create a new option for the price threshold to make it even more configurable.

Hints:

- You need to work with another AngularJS directive called **ng-class**
- You need to specify new classes in the CSS-SCSS section of the widget
- You need to remove the inline style for “color” of the price option

Once completed, the widget should look like the examples.





Answer:

1. Navigate to **Service Portal > Service Portal Configuration ➚ > Widget Editor.** Select the **My Stock** widget.

2. Edit the **HTML Template** to use this span for the price.

   <span ng-class="{ high : c.data.price > 60, low :c.data.price <=60 }">{{c.data.price | currency:"$"}}

3. Edit the CSS-SCSS and add these two classes.

   .high {color: green;}
   .low {color: red;}

Lab Goal

Lab 4.0 Advanced Custom Widgets

This lab explains how to load data, specifically active HR Case records, into a table. You begin with building the HTML for a Bootstrap table, before loading data to populate the table.

Use the links below to find documentation on how to create Bootstrap tables:

- https://getbootstrap.com/css/#tables
- http://www.w3schools.com/bootstrap/bootstrap_tables.asp

Hint: When you click **Try it yourself**, you can copy/paste the full HTML of a table.

Create a Bootstrap Table

1. Navigate to **Service Portal -> Service Portal Configuration**, open the **Widget Editor**, and create a new Widget in which you write the code. The table heading (<**th**>) should contain the following fields:

   Number, **Short Description**, **Opened for**, **State**, **Priority**, **Task Type**, **Created on**

2. Create a table row (<**tr**>) of demo data, the actual data should reside in a <**td**> (column) tag.

   Hint: All copied code from Word can be reformatted quickly by marking the code (Command + A) and then pressing Shift + Tab, which will prettify the code ☺

Lab Validation

1. Your table should look like the example.



That is all great, but you probably do not want to create HTML for hundreds, maybe even thousands of records. Change this by loading data and adding AngularJS after. Here is an example HTML Template to get the above example.

```
<!-- Table without AngularJS  -->
<div class="container">
  <div class="table-responsive">
    <table class="table">
      <thead>
        <tr>
          <th>Number</th>
          <th>Short Description</th>
          <th>Opened for</th>
          <th>State</th>
          <th>Priority</th>
          <th>Task Type</th>
          <th>Created</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>123</td>
          <td>This is a Short Description</td>
          <td>Rita Requester</td>
          <td>In Progress</td>
          <td>2 - High</td>
          <td>HR Benefits</td>
          <td>2016-07-08 14:30:00</td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

https://pastebin.com/8WT3DCyx

Write the Server Script

Before adding AngularJS to the form, make sure you have all the necessary data.

To load the cases, use the same API that you use on the Backend side of things: the GlideRecord.

A step-by-step description on how to write the code, could sound like this:

1. Initialize a new, blank, array called **incidents**, which will live in the data object.
2. After that, create a new **GlideRecord** on the Incidents table to retrieve only active Incidents.
3. Use the **$sp Service Portal Server Side API** to retrieve values, display values, and a set of field labels.
4. For the table, include the following data:

   Table Headings: the labels of all the fields to appear in the table
   Table Rows: the actual HR Case records with their values
   ng-repeat is the AngularJS directive to use. It awaits an array, so you must initialize an **Incidents** array that is populated by iterating over the GlideRecord result.

5. The resulting code looks the example.

```
 (function() {
      /* populate the 'data' object */
      /* e.g., data.table = $sp.getValue('table'); */

      //Initialize the 'incident' Property in the data object
      data.incidents = [];
      //Iterate over the incident table to retrieve all active cases
ordered by number (descending - newest ones first)
      var gr = new GlideRecord("incident");
      gr.addActiveQuery();
      gr.orderByDesc("number");
      gr.query();

      //We will use the fields array for ng-repeat on the <tr> level since
we want to have all <tr>'s created automatically instead of manually
assigning values
      data.fields = ["number", "short_description", "caller", "state",
"priority", "sys_class_name", "sys_created_on"];

      //make use of the $sp API and get all field labels for the provided
fields
      data.labels = $sp.getFields(gr,
'number,short_description,caller,state,priority, sys_class_name,
sys_created_on');

      while(gr.next()) {
            var incidentArray = {};

            //get all display values for the fields below and write them
into the incident object
            $sp.getRecordDisplayValues(incidentArray, gr,
'number,short_description,caller,state,priority, sys_class_name,
sys_created_on');
            //get the value for sys_id and also write it into the incident
object
            $sp.getRecordValues(incidentArray,gr,'sys_id');

            //push the whole incident object into the incidentsArray array
            data.incidents.push(incidentArray);
      }
      //log the data object to the browser console, so we can click through
it and see our results
      $sp.log(data);

      return data;

})();
```

https://pastebin.com/vqywqpAd

Add AngularJS to the Bootstrap Form

Now the data object is populated. Access that data in the HTML section.

Step 1 – ng-repeat over the table headings

First, eliminate the double **<th>** tags by repeating over a single <th> tag.

In the server script, you populated a property called **labels** with all the label names. As everything is in the **data** object, you must use **data.labels** to access them.

Within that property label is the attribute called **label**, which holds the actual label name. **fieldLabel** is a key that represents each label object and is only defined here in the HTML – that is part of the notation of the ng-repeat AngularJS directive.

Example:

```
<th ng-repeat="fieldLabel in data.labels">{{fieldLabel.label}}</th>
```

Notice the double curly brackets, opening and closing. This is the actual AngularJS markup to retrieve a value.

Step 2 – ng-repeat over the table rows

The rows represent the number of records. If there are 20 records on the server, then you need 20 rows. You probably already figured it out, but this is how the <tr> repeat looks:

```
<tr ng-repeat="case in data.incidents">
```

All cases are stored in the hrCases array as separate objects.

Step 3 – ng-repeat over the table columns

Now, this is a bit trickier than the heading and rows. The reason for this is that there are changing values in each <**td**>. So unless you want to write a <td> for each field, here is where the **data.fields** object comes into play. All technical field names stored in that object. Because of that you can now say for each case get the value of the actual field name currently being processed.

```
<td ng-repeat="field in data.fields">{{case[field]}}</td>
```

If you don’t do that, you would have to write:

```
<td>{{case.number}}</td>
<td>{{case.short_description}}</td>
… etc.
```

Add a Record Watcher to the List

Wouldn't it be awesome if the list would update itself any time a record is deleted or inserted into the HR Case table? You bet it would be! And how hard can it be? Well, not at all ☺

Just replace your client script with the following code snippet:

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

  spUtil.recordWatch($scope, "incident", "", function(name, data) {
          //If there is a new record we want to re-render our list
//(which is simply executing the server script again)
          c.server.update();
  });
}
```

https://pastebin.com/Nkpm4Ex3

The Service Portal **spUtil** API includes the Record Watcher. You are just telling it to watch the HR Case table and only records with the defined filter. If the watcher detects a change (like an insert, update or delete), then you must execute the server script again – because that is the script that builds the list of records. This can be achieved by simply writing **c.server.update()**.

Now you can insert a record in the backend and then come back to your list, that record should now be on top of the list without having to reload the page.

If you want you can add some styling to your table (e.g., by adding the **panel** class to the **top ** element).