**Service Portal development best practice guide**

This guide contains exercises that will teach you best practices for developing applications on Service Portal.

1. [Connect your instance to github](https://github.com/service-portal/CCW3956/wiki/Lab-0)
2. [Using option schema](https://github.com/service-portal/CCW3956/wiki/Lab-1)

1.5 [Using SP Instance](https://github.com/service-portal/CCW3956/wiki/Lab-1.5)

1. [Handling the empty State](https://github.com/service-portal/CCW3956/wiki/Lab-2)
2. [Embedded widgets and directives](https://github.com/service-portal/CCW3956/wiki/Lab-3)
3. [Sharing data and events](https://github.com/service-portal/CCW3956/wiki/Lab-4)
4. [Rest APIs & Server Script](https://github.com/service-portal/CCW3956/wiki/Lab-5)
5. [Localize & consider WCAG](https://github.com/service-portal/CCW3956/wiki/Lab-6)
6. [UI best practices](https://github.com/service-portal/CCW3956/wiki/Lab-7)

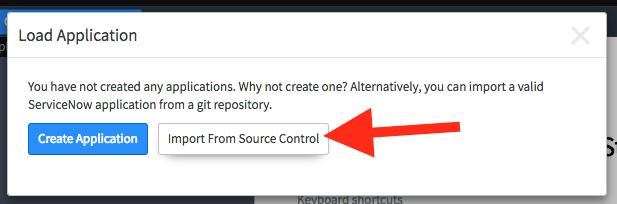
# Connect your instance to github

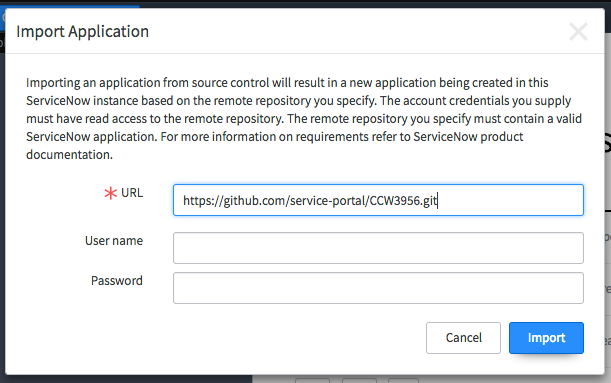
This document will guide you through a series of labs to ultimately create a card list and form interface that can be used to fulfill requests. Each lab will successively instruct a best practice for building widgets that are easy to use, maintain, and configure.

This lab guide will walk you through each of the labs but if you fall behind or want to jump to the solution for any of the exercises, just switch to the branch of the next lab.

## Import the lab playground from github

1. Go to STUDIO in your navigator or go to /$studio.do in your instance
2. You should see a dialog titled “Load Application”. If not click on the File menu.
3. Choose Import from Source Control

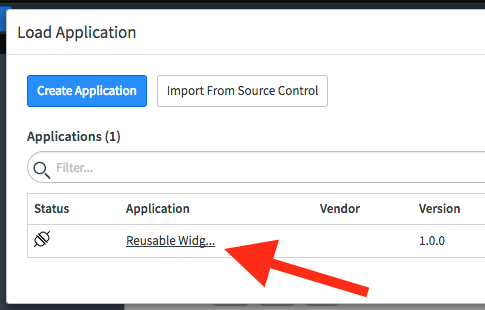




In URL type: https://github.com/service-portal/CCW3956.git

Leave user name and password blank

1. Click Import
2. When it’s done loading, click **Select Application**. In the application list select **Reusable Widgets**.



At any time you can switch to a branch by opening the Source Control menu and selecting **Switch Branch**. Switching branches will reset your local changes and automatically apply the changes from the branch to your instance.

# Using option schema

This lab starts out with a quick introduction to Widget Options. The Widget’s Options Schema is a way to define configuration options (or parameters) that your widget can accept. For example, the cool clock widget has two options defined, a timezone and second hand color. Whenever you use an instance of the cool clock widget, you can specify values for both of those options.

The options schema supports the following field types:

* String
* Boolean
* Integer
* Reference
* Choice
* Field\_list (depends on table)
* Field\_name (depends on table)
* Glide\_list

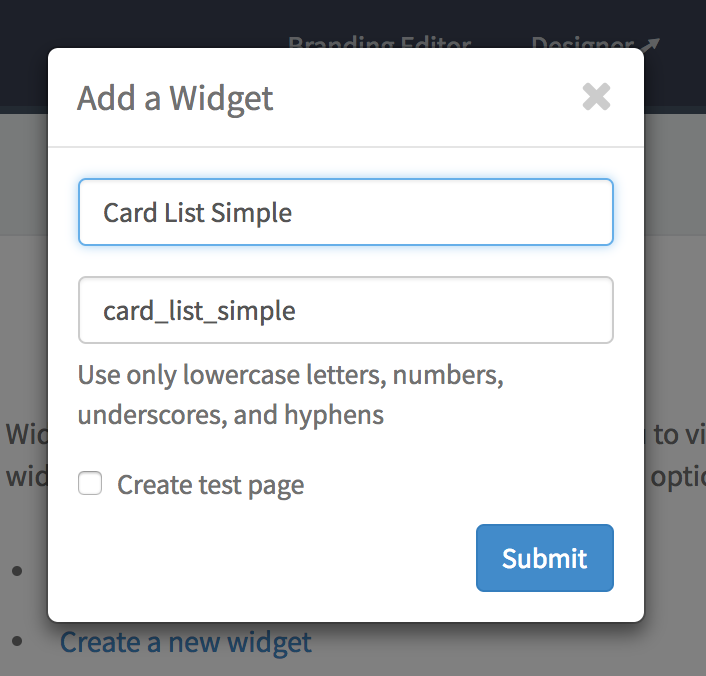
For more field types, or if your widget options require ui-policy or client scripts, you can create a table that extends sp\_instance and use any field type in the ServiceNow platform.You will see an example of this towards the end of this lab.

## Create a card list widget

1. From Studio, Click **Create Application File** and type Widget in the filter. Select Widget and click Create. Create new
2. Create a new widget with the following values:

Name: Card List Simple

ID: card\_list\_simple

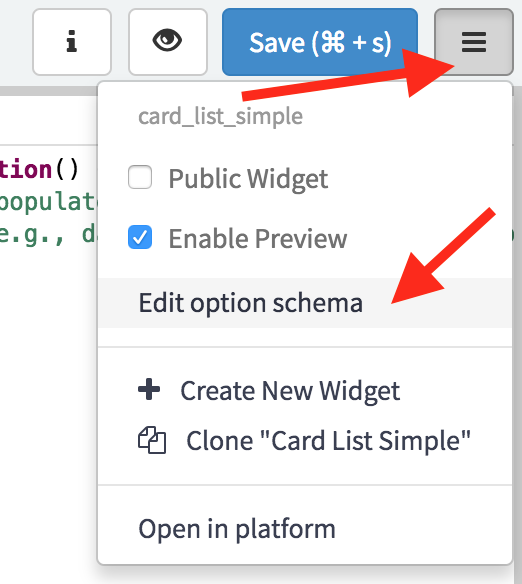


1. Click Submit

Optional: Open the properties menu and check **Enable Preview** With preview enabled, click on the preview button any time you want to see a real time preview of your widget in action.

## Define widget options

1. Select **Edit option schema** from the widget properties menu.



1. Click the [ + ] button and add the following values:

Label : **Title**

Name: **title**

Type: **String**

Hint: **The title for the card list**

1. Click [ + ] again add another option:

Label: **Table**

Name: **table**

Type: **string**

Hint: **The table to query**

1. Click [ + ] again add another option:

Label: **Filter**

Name: **filter**

Type: **string**

Hint: **The filter query string**

1. Click [ + ] again add a final option:

Label: **Card Fields**

Name: **fields**

Type: **string**

Hint: **Comma separated list of field names to include on the cards**

1. Click **Save**
2. Click **Save** on the widget editor again. This is very important!

## Use the options to render a list of records

You can refer to any option value in your widget client script or server script as **options.optionName**. Use the options to render a list of records.

Add this to the widget **Server Script** code block:

(function() {

if (options.table) {

data.rows = [];

var gr = new GlideRecord(options.table);

gr.addEncodedQuery(options.filter);

gr.setLimit(25);

gr.query();

if (!data.primaryField) {

data.primaryField = gr.getDisplayName();

}

var fields = "sys\_id," + data.primaryField + "," + options.fields;

data.fields = $sp.getFieldsObject(gr, fields);

while(gr.next()) {

var row = {};

$sp.getRecordDisplayValues(row, gr, fields);

data.rows.push(row);

}

}

})();

Add this to the **Client Script** code block:

function() {

var c = this;

c.cardFields = (c.options.fields) ? c.options.fields.split(",") : "";

c.getPrimaryField = function getPrimaryField(row){

return row[c.data.primaryField];

};

}

Add this to the **CCS - SCSS** code block:

.fields {

$dl-horizontal-offset: 120px;

dt {

font-weight: normal;

color: $text-muted;

float: left;

width: ($dl-horizontal-offset - 20);

clear: left;

text-align: right;

@include text-overflow;

}

dd {

@include clearfix;

margin-left: $dl-horizontal-offset;

}

}

Add this to the **HTML Template** code block:

<div>

<div class="panel panel-default">

<div class="panel-heading">

<span class="h3 panel-title">{{::c.options.title}}</span>

</div>

<div class="panel-body">

<div class="list-group" ng-repeat="row in c.data.rows track by row.sys\_id">

<a href="javascript:void(0)" class="list-group-item">

<div class="h4 list-group-item-heading">{{c.getPrimaryField(row)}}</div>

<dl class="fields">

<span ng-repeat="f in c.cardFields"><dt>{{c.data.fields[f].label}}</dt><dd>{{row[f]}}</dd></span>

</dl>

</a>

</div>

</div>

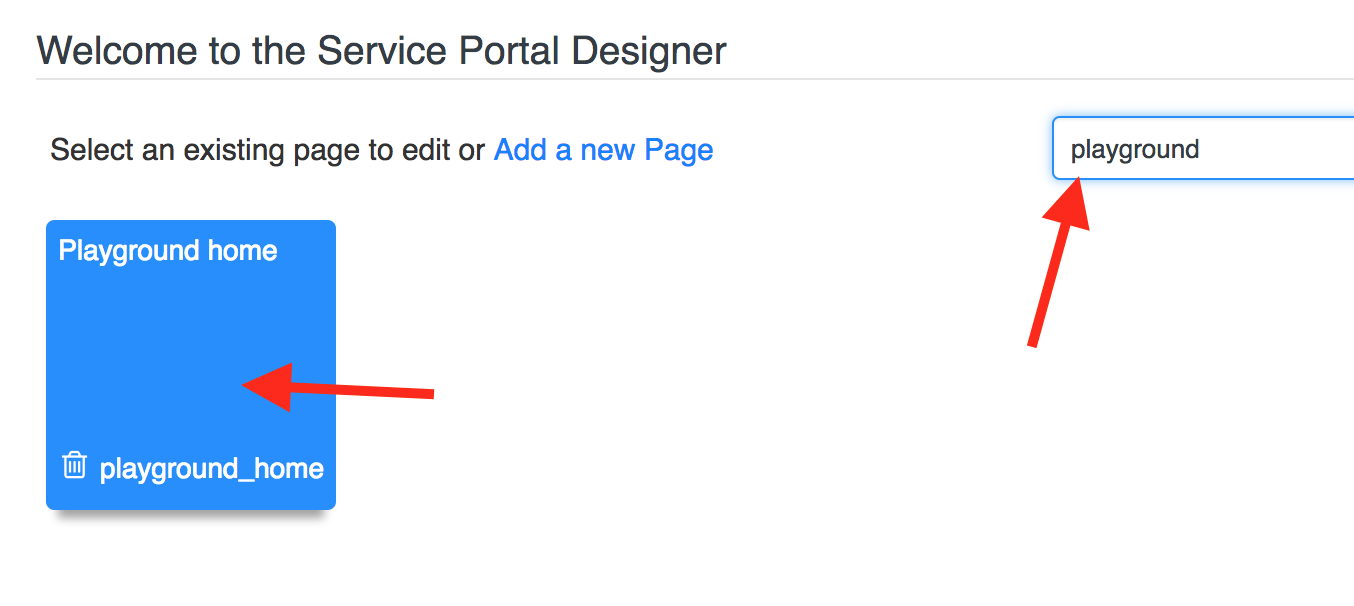
</div>

</div>

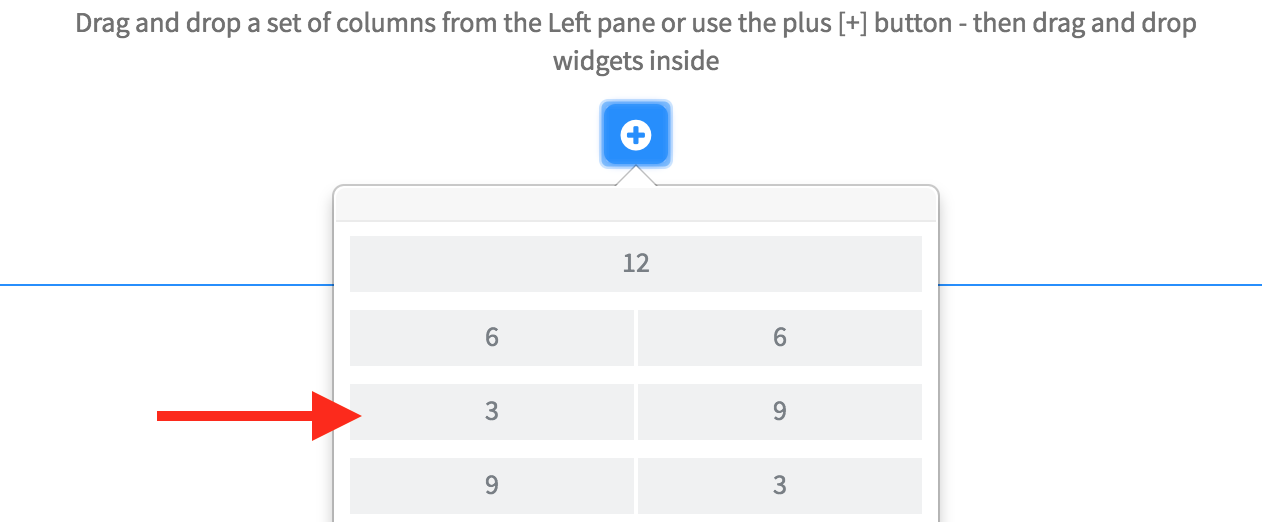
**Save the widget**

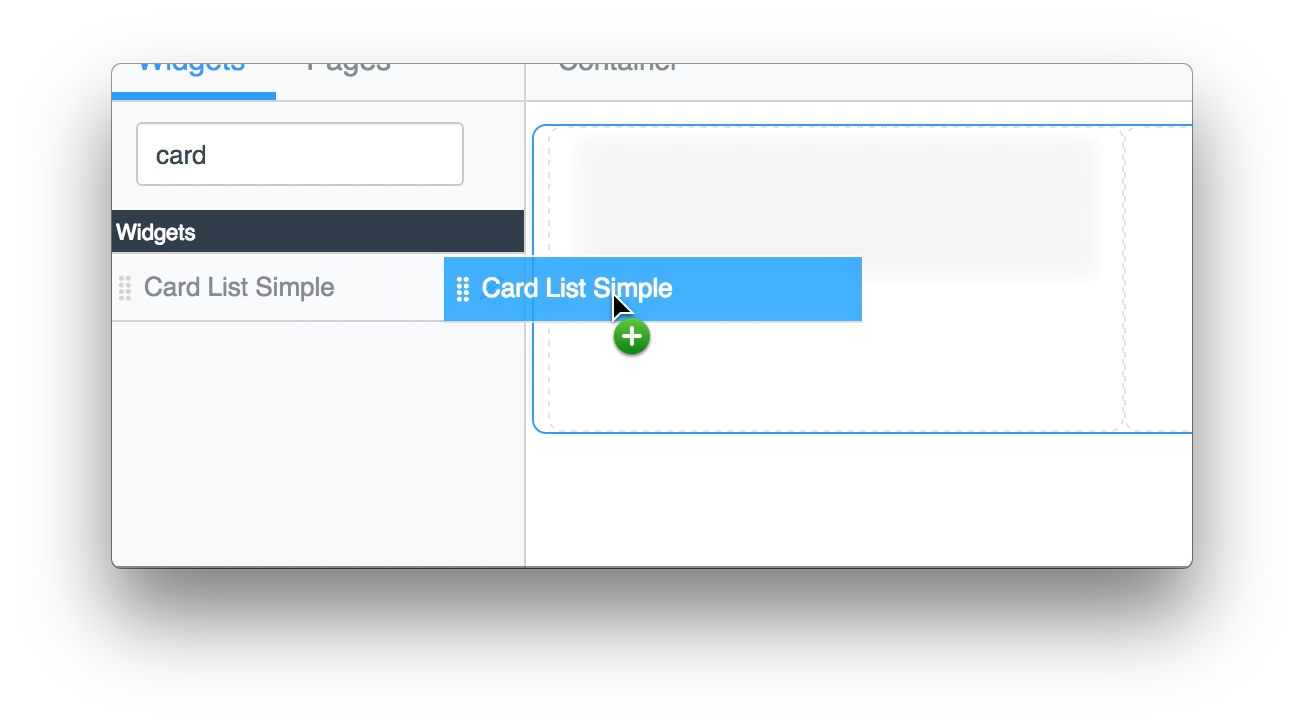
## Add the card list to a page using the Service Portal Designer.

1. Open the designer by opening a new tab and going to [instance]/$spd.do
2. Type in the filter to find the page called: **Playground**. Open the page Playground home



1. Click on the [ + ] button to add a predefined layout of [ 3 | 9 ]



1. Under the widgets toolbox, find the widget Card List Simple and drag it to the left column. 

## Configure the widget using instance options

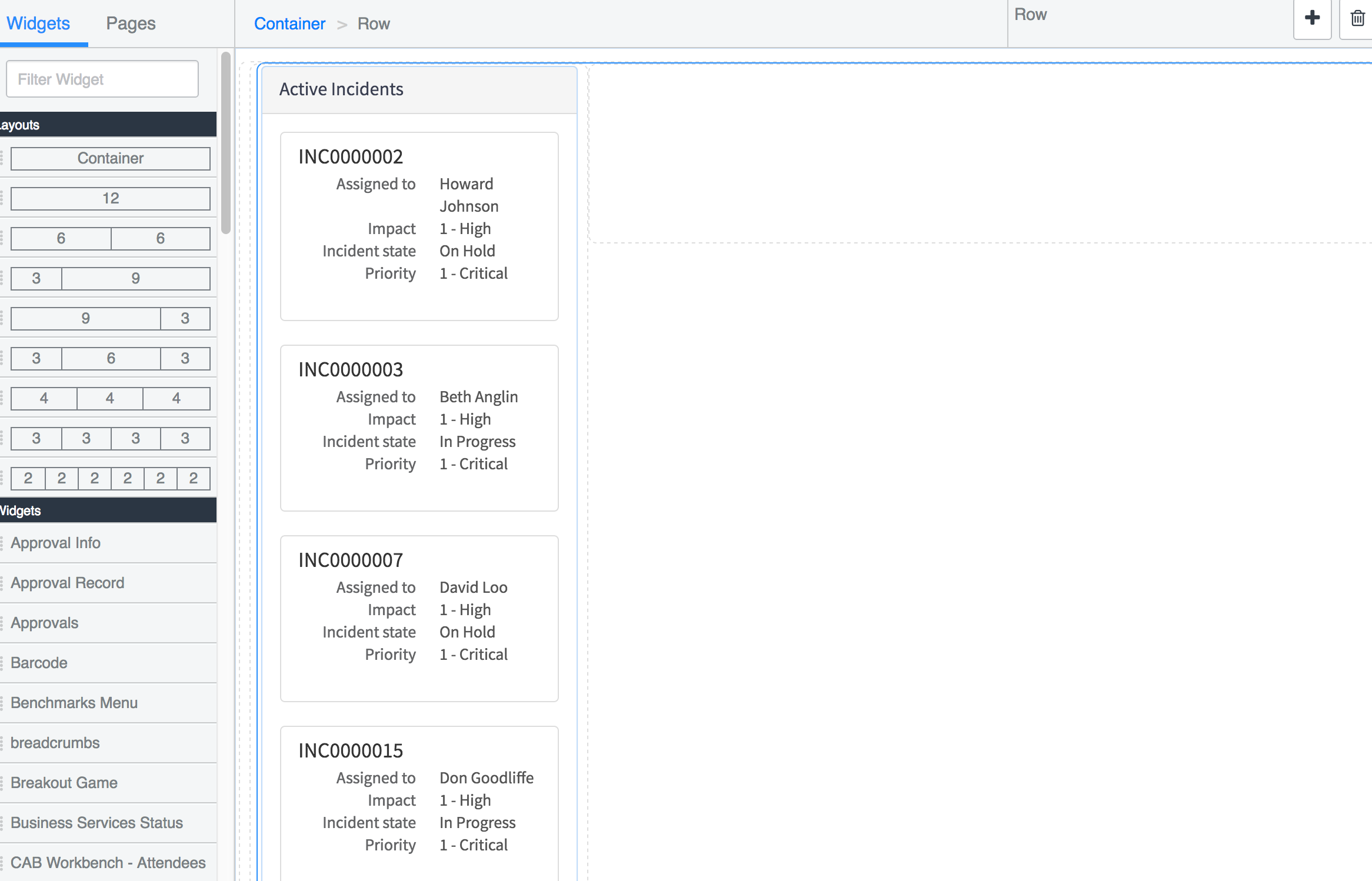
1. Hover over the widget and click the edit pencil
2. Enter the following values and click **Save**

Title: **Active Incidents**

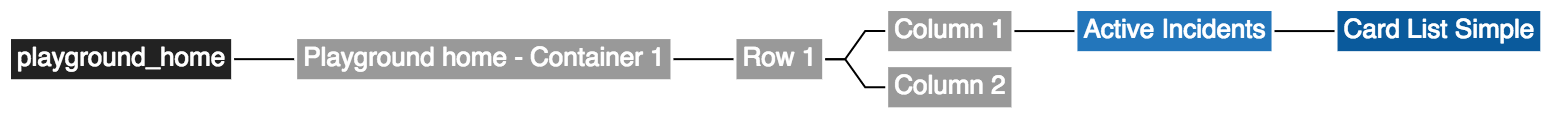
Table: **incident**(all lower case, this is the table name not the label)

Filter: **active=true**

Card Fields: **assigned\_to,impact,incident\_state,priority** (all lower case, no spaces)

1. You should see the following in the widget designer: 

### This is a breakdown of everything that happened behind the scenes:



When you added the 2 column layout using the designer it created 3 records:

sp\_row (1 record)

A reference to the sp\_container

sp\_column (2 records)

Left column with Size - md: 3 and a reference to sp\_row

Right column with Size - md: 9 and a reference to sp\_row

When you dropped the card list widget into the left column it created 1 record:

\*\*sp\_instance\*\*

A reference between the left column and the Card List Simple Widget

**Side note:** You can visualize this data tree map using the page editor:

[instance]/sp\_config?id=page\_edit&p=playground\_home

When you edited the instance options for the card list it wrote the following JSON object into the sp\_instance.widget\_parameters field:

{

"title": {

"value": "Active Incidents",

"displayValue": "Active Incidents"

},

"table": {

"value": "incident",

"displayValue": "incident"

},

"filter": {

"value": "active=true",

"displayValue": "active=true"

},

"fields": {

"value": "assigned\_to,impact,incident\_state,priority",

"displayValue": "assigned\_to,impact,incident\_state,priority"

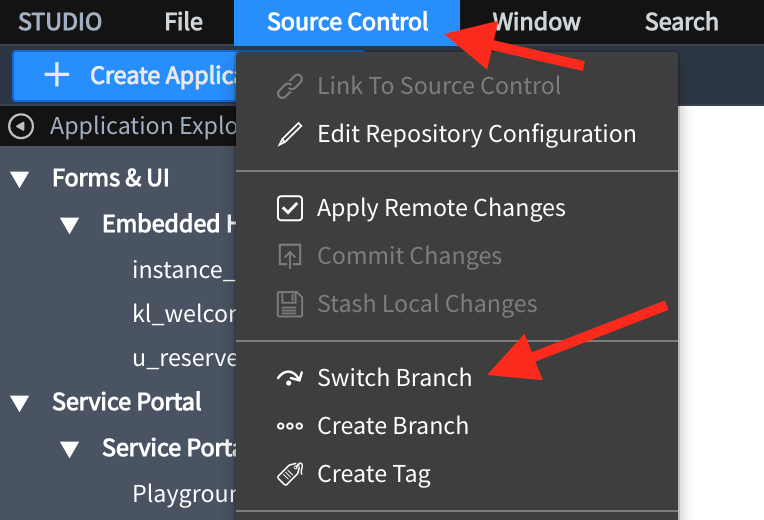
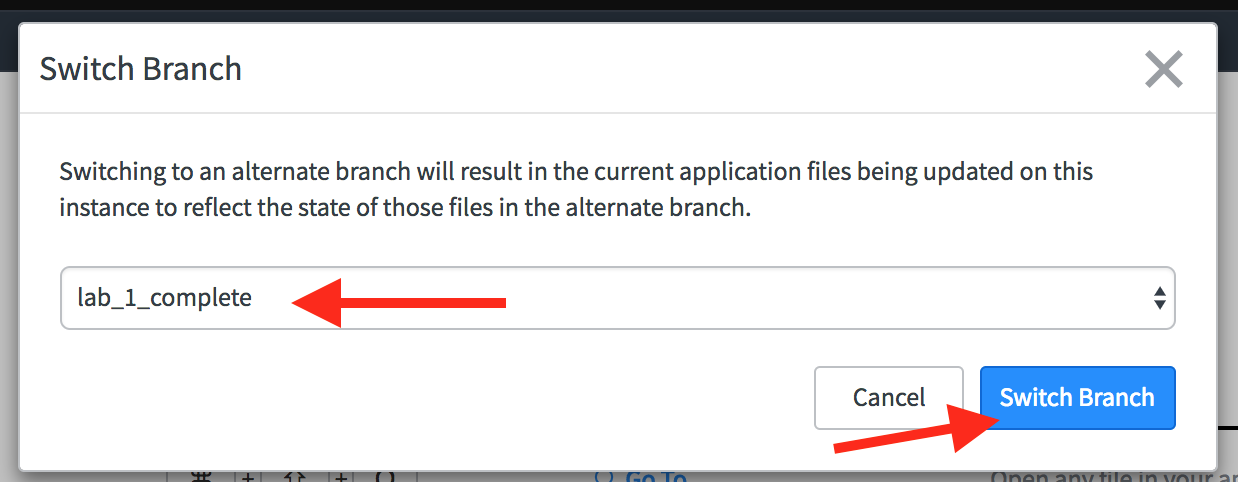
}

}

## Optional

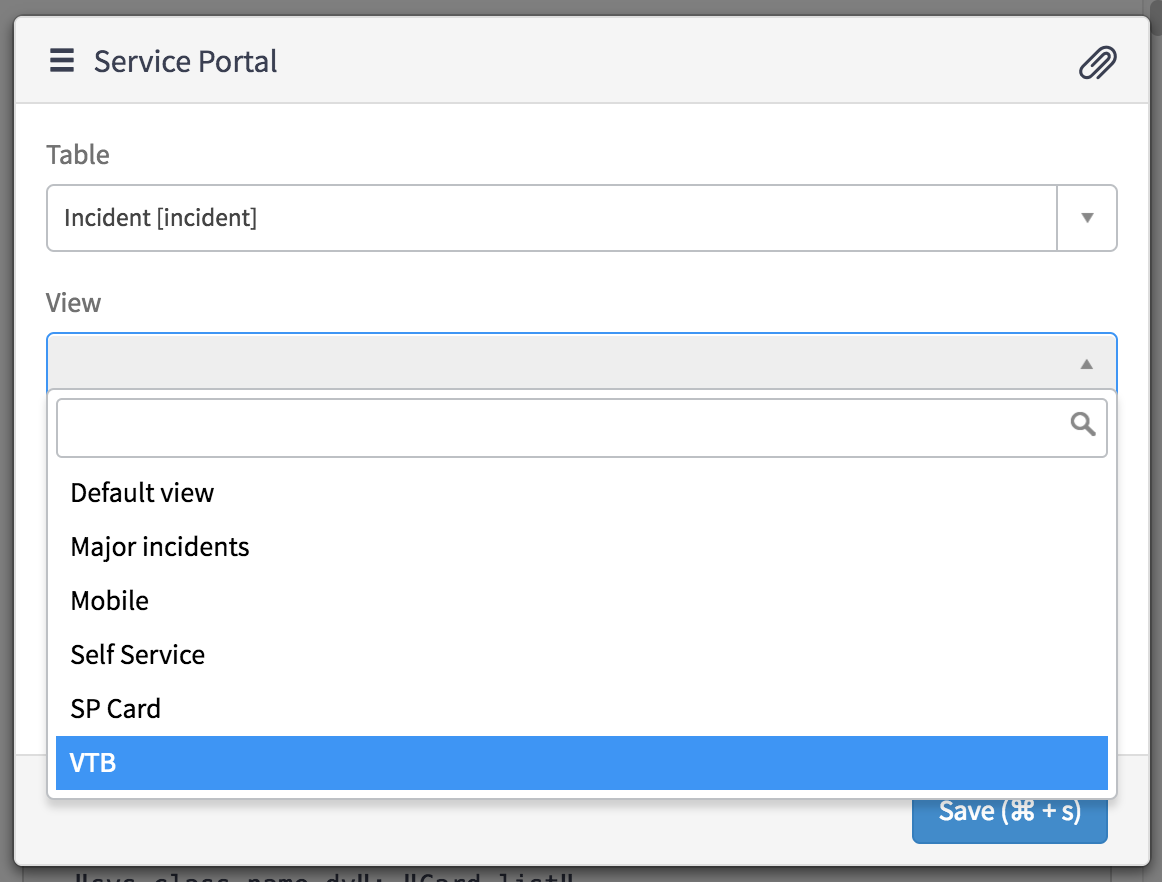
At this point, if you are behind or want to get your instance up to parity with the lab guide you can check out the branch lab\_1\_complete using STUDIO.

### This is optional and you will lose all of your local changes.

1. To check out that branch, open STUDIO and click on the Source Control menu. 
2. Select lab\_1\_complete from the list and click Switch Branch. 

# Getting started with custom sp\_instance extension tables

Widgets like the card list are rarely useful with such basic configuration. A list (whether cards or tables) usually has many requirements depending on the kind of data you want to show. For instance, it can be tedious and arcane to type in a list of fields to show on each card. What if your widget has options that only apply under certain circumstances? In this section, you will learn how to create an sp\_instance extension table that allows for more types of options and configurability of your widgets. To give a fair comparison, we will create a similar widget. But when you configure this widget you will be able to pick a pre-built view for the list of fields on your cards. If you already have a VTB card layout, you could just re-use it like this:



## Create a table that extends sp\_instance\_table

1. Using STUDIO click Create Application File.

create application file

1. Select: **Data Model**- > **Table**. Click **Create**.
2. Create the table with the following values:

Label: **Card List Instance** Name: [automatically created] Extends table: **Instance with Table**Create Module: **false**

1. Click Submit

## Create a script include to help build a reference qualifier

1. Using STUDIO click Create Application File.

create application file

1. Select: **Server Development** -> Script Include. Click **Create**.
2. Use the following values for the script include:

Name: **ListViewHelper**

Script:

var ListViewHelper = Class.create();

ListViewHelper.prototype = {

initialize: function() {

},

getViewsForList: function(table) {

var views = [];

var gr = new GlideRecord("sys\_ui\_list");

gr.addEncodedQuery("viewNOT LIKErpt^sys\_userISEMPTY");

gr.addQuery("name", table);

gr.query();

while(gr.next()) {

var view = gr.getValue("view");

if (views.indexOf(view) == -1) {

views.push(view);

}

}

return views.join(",");

},

type: 'ListViewHelper'

};

1. Click Submit.

## Create a reference field to select a view

1. Using STUDIO click on the table **Card List Instance** (If you don’t see a place to add a new column, close the tab and open the Card List Instance table again)
2. Add a new column  
   Type: **Reference**  
   Column Label: **View**  
   Column Name: **view**  
   Reference: **UI View** (sys\_ui\_view)  
   Display: **checked**
3. Right click the header and select **Save**.
4. Near the bottom of the form, under Related Links, click **Advanced view**
5. In the section Reference Specification. Set the following values:

Use reference qualifier: Advanced Reference qual: javascript:"sys\_idIN" + new x\_snc\_reusable\_wid.ListViewHelper().getViewsForList(current.table)  
6. Click **Update**.

## Clone Card list widget

1. Using STUDIO click on the widget Card List Simple
2. Open the widget properties menu and select **Clone** **“Card List Simple”**.

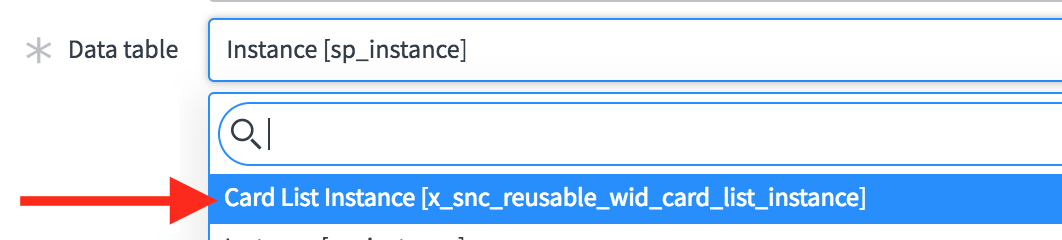
![Clone](2. Open the widget properties menu and select Clone “Card List Simple”.)

1. Name the new widget: **Card List**
2. Click **Submit**.

## Edit Card List widget in the platform

The widget editor doesn’t currently offer a way to do this step. Open the new Card List widget using the platform by using the widget properties menu and selecting **Open in platform**.

1. Ensure the widget name is “Card List”  
   If not you must go to [instance]/sp\_widget\_list.do and select the **Card List** widget
2. Change the field Data Table: **Card List Instance**



1. Set Fields: Title, Table, View, Filter, Display Field
2. Set the Option Schema to:

[{"hint":"A field that contains a value 1 -4 to indicate priority.",

"name":"priority\_field",

"default\_value":"priority",

"label":"Priority Field",

"type":"string"}]

5.Set the HTML Template to:

<div>

<div class="panel panel-default">

<div class="panel-heading">

<span class="h3 panel-title">{{::c.options.title}}</span>

</div>

<div class="panel-body">

<div class="list-group" ng-repeat="row in c.data.rows track by row.sys\_id">

<a href="javascript:void(0)" class="list-group-item" ng-style="::c.getPriority(row)">

<div class="small">{{::c.getPrimaryField(row)}}</div>

<div class="h4 list-group-item-heading">{{::row[c.options.display\_field]}}</div>

<dl class="fields">

<span ng-repeat="f in c.cardFields"><dt>{{c.data.fields[f].label}}</dt><dd>{{row[f]}}</dd></span>

</dl>

</a>

</div>

</div>

</div>

</div>

1. Set Server script to:

(function() {

/\* populate the 'data' object \*/

/\* e.g., data.table = $sp.getValue('table'); \*/

if (options.table) {

data.cardFields = $sp.getListColumns(options.table, options.view\_dv);

data.rows = [];

var gr = new GlideRecord(options.table);

gr.addEncodedQuery(options.filter);

gr.setLimit(25);

gr.query();

if (!data.primaryField) {

data.primaryField = gr.getDisplayName();

}

var fields = "sys\_id," + data.primaryField + "," + data.cardFields;

data.fields = $sp.getFieldsObject(gr, fields);

while(gr.next()) {

var row = {};

$sp.getRecordDisplayValues(row, gr, fields);

data.rows.push(row);

}

}

})();

1. Set the Client Controller to:

function() {

/\* widget controller \*/

var c = this;

if (c.data && c.data.cardFields) {

c.cardFields = getCardFields(c.data.cardFields, c.data.primaryField);

}

c.getPrimaryField = function getPrimaryField(row) {

return row[c.data.primaryField];

};

function getCardFields(allFields, primaryField) {

var cardFields = [];

allFields = allFields.split(",");

var exclude = [primaryField, c.options.display\_field, c.options.priority\_field];

for (var i = allFields.length-1; i>= 0; i--){

if (exclude.indexOf(allFields[i]) == -1) {

cardFields.push(allFields[i]);

}

}

return cardFields;

}

c.getPriority = function(row) {

if (typeof row[c.options.priority\_field] == "undefined") {

return;

}

var p = row[c.options.priority\_field];

var color;

if (p.indexOf('4') > -1) {

color = 'green';

} else if (p.indexOf('3') > -1) {

color = 'yellow';

} else if (p.indexOf('2') > -1) {

color = 'orange';

} else if (p.indexOf('1') > -1) {

color = 'red';

}

if (color) {

return {'border-left': '3px solid ' + color};

}

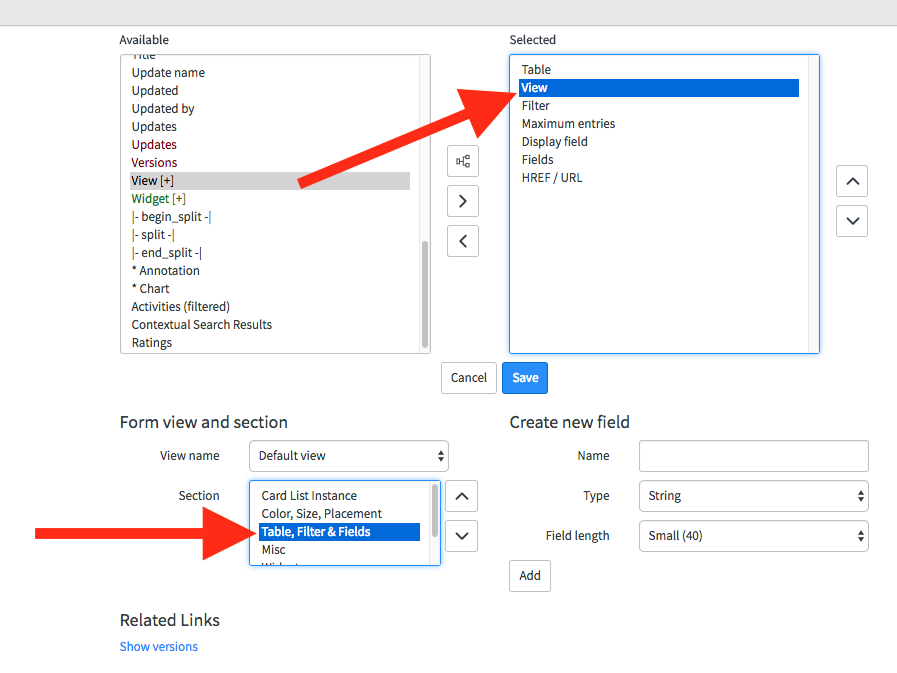
}

}

1. Click **Update**

The primary difference with this widget is that when you drag it onto your page from the designer it will create the instance record in the table \*card\_list\_instance. Editing records in that table will apply the reference qualifier on the view field and make it super simple to define a view that can be shared across different card lists on the platform.

## Configure the card\_list\_instance form

1. In a new tab go to [instance]/x\_snc\_reusable\_wid\_card\_list\_instance.do
2. Right click on the form header. Choose **Configure** -> **Form Layout**.
3. Choose the Section “**Table, Filter & Fields**”.
4. Move the field **View** between **Table** and **Filter**. 
5. Click **Save**.

## Add the new card list to a page

1. Using STUDIO click Create Application File.
2. Select Service Portal -> Service Portal Page. Click Create.
3. For page name and ID use the following values:  
   Page title: Incident Workspace Page ID: iw
4. Add the same [ 3 | 9 ] column layout to the default container.
5. Drag the widget Card List to the left column.
6. Click on the edit icon and set the following instance options:  
   Title: **Active Incidents**  
   Table: **Incident**  
   View: **Mobile**  
   Filter: **active=true**  
   Display field: **Short Description**  
   Priority Field: [**use default**]
7. Click **Save**.

# Providing an empty state

When you dropped the card list widget onto the page using the designer it was nothing more than a box with an outline. Even editing the instance options only slightly revealed what the widget does. When practical, your widgets should provide an empty state and tip users off about how to proceed and what it will look like when it has data.

Using the Demo data field on a widget is good for previewing a widget in the widget editor but won’t work in the designer. This is because the demo data is overwritten as soon as there is an instance record. Here are a couple of ways you can provide an empty state for a good experience in the designer.

## Show a default title

1. From STUDIO, Open the Card List widget.
2. Add the following code to the client controller:

function isConfigured() {

if (!c.options) {

return false;

}

if (typeof c.options.title === "undefined")

return false;

return true;

}

1. Now you can use that function to set an example title right in the controller script. Find the lines:

if (c.data && c.data.cardFields) {

c.cardFields = getCardFields(c.data.cardFields, c.data.primaryField);

}

and replace it with this code:

if (!isConfigured()) {

c.options.title = "My active incidents";

} else if (c.data && c.data.cardFields) {

c.cardFields = getCardFields(c.data.cardFields, c.data.primaryField);

}

## Add some demo data

1. If you want to go for a richer experience, change the client script to this:

if (!isConfigured()) {

// Provide demo data if options.title is empty

c.options.title = "My active incidents";

c.options.priority\_field = "priority";

c.options.display\_field = "short\_description";

c.data = {

"primaryField": "number",

"cardFields": "category",

"rows": [{"sys\_id":"1",

"number":"INC0000002",

"short\_description":"Network file shares access issue",

"category":"Network",

"priority":"1 - Critical"},

{"sys\_id":"2",

"number":"INC0000003",

"short\_description":"I need a mouse",

"category":"Hardware",

"priority":"4 - Low"}]

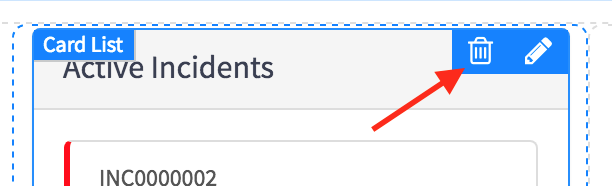
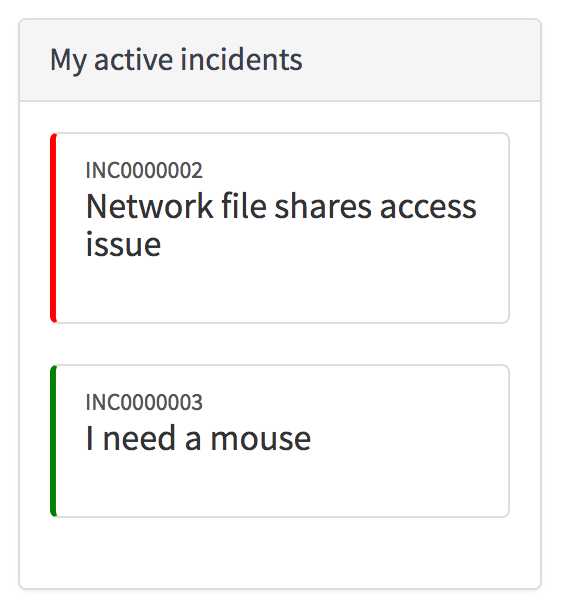
}

} else if (c.data && c.data.cardFields) {

c.cardFields = getCardFields(c.data.cardFields, c.data.primaryField);

}

## Try it out

1. Open the designer in a new tab [instance]/$spd.do
2. Edit the Incident Workspace page 
3. Delete the Card List widget from the left column by clicking on the delete button when hovering over the widget. When it asks you to confirm, click **Yes**
4. Drag a new instance of the Card List widget from the widget list to the left column again.
5. Now you will see a much better example of the card list before it’s even configured: 

# Embedded Widgets and Directives

As your widgets become more complex you will want to break them up into smaller manageable components. Those components could then be reused elsewhere or just help define the connections between parts of your application.

## Move the card template to a directive

We can separate the card from the card list by moving the card template to a directive.

1. Open the widget Card List in the platform UI by going to [instance]/sp\_widget\_list.do
2. Scroll down to the bottom of the widget find the related list Angular Providers  
   3.Click **New**
3. Use the following values for the new Angular Provider:  
   Type: **Directive**  
   Name: **card**  
   Client Script:

function() {

var template = "<div class='list-group'>";

template += "<a href='javascript:void(0)' class='list-group-item' ng-style='::ctrl.getPriority()'>";

template += "<div class='small'>{{::row[primaryField]}}</div>";

template += "<div class='h4 list-group-item-heading'>{{::row[displayField]}}</div>";

template += "<dl class='fields'>";

template += "<span ng-repeat='f in cardFields'><dt>{{fields[f].label}}</dt><dd>{{row[f]}}</dd></span>";

template += "</dl>";

template += "</a>";

template += "</div>";

return {

restrict: 'E',

scope: {

row: '=',

primaryField: '=',

displayField: '=',

priorityField: '=',

fields: '=',

cardFields: '='

},

template: template,

controllerAs: 'ctrl',

controller: function($scope) {

var ctrl = this;

var row = $scope.row;

ctrl.getPrimaryField = function getPrimaryField() {

return row[$scope.primaryField];

};

ctrl.getPriority = function() {

if (typeof row[$scope.priorityField] == "undefined") {

return;

}

var p = row[$scope.priorityField];

var color;

if (p.indexOf('4') > -1) {

color = 'green';

} else if (p.indexOf('3') > -1) {

color = 'yellow';

} else if (p.indexOf('2') > -1) {

color = 'orange';

} else if (p.indexOf('1') > -1) {

color = 'red';

}

if (color) {

return {'border-left': '3px solid ' + color};

}

};

}

};

}

Notice how the template is embedded right in the directive. Understand the pros and cons:

Pros:

* Keeps the template close to the directive code
* Allows you to pass in a different template as a scope parameter
* Can be reused without providing an angular template

Cons:

* Template isn’t cached
* Directive can get very large

This lab is ultimately about componentizing and so we want to isolate the rendering of the card from the card list widget.

## Use <workspace-card> in the widget template

1. Open the **Card List** widget
2. Set the HTML Template to:

<div>

<div class="panel panel-default">

<div class="panel-heading">

<span class="h3 panel-title">{{::c.options.title}}</span>

</div>

<div class="panel-body">

<workspace-card ng-repeat="row in c.data.rows track by row.sys\_id"

row="::row"

primary-field="::c.data.primaryField"

display-field="::c.options.display\_field"

priority-field="::c.options.priority\_field"

fields="::c.data.fields"

card-fields="::c.cardFields"></workspace-card>

</div>

</div>

</div>

1. Set the client script to:

function() {

/\* widget controller \*/

var c = this;

if (!isConfigured()) {

// Provide demo data if options.title is empty

c.options.title = "My active incidents";

c.options.priority\_field = "priority";

c.options.display\_field = "short\_description";

c.data = {

"fields": {"category": {"label": "Category"}, "opened\_by": {"label": "Opened By"}},

"primaryField": "number",

"cardFields": "category,opened\_by",

"rows": [

{"sys\_id":"1", "number":"INC0000002","short\_description":"Network file shares access issue","category":"Network","priority":"1 - Critical","opened\_by":"admin"},

{"sys\_id":"2", "number":"INC0000003","short\_description":"I need a mouse","category":"Hardware","priority":"4 - Low","opened\_by":"Garfield"}

]

}

}

c.cardFields = getCardFields(c.data.cardFields, c.data.primaryField);

c.getPrimaryField = function getPrimaryField(row) {

return row[c.data.primaryField];

};

function getCardFields(allFields, primaryField) {

var cardFields = [];

allFields = allFields.split(",");

var exclude = [primaryField, c.options.display\_field, c.options.priority\_field];

for (var i = allFields.length-1; i>= 0; i--){

if (exclude.indexOf(allFields[i]) == -1) {

cardFields.push(allFields[i]);

}

}

return cardFields;

}

function isConfigured() {

if (!c.options) {

return false;

}

if (typeof c.options.title === "undefined")

return false;

return true;

}

}

## Embed filter widget

You can embed a widget inside your html template using the directive. To read more about embedded widgets go here: <https://github.com/service-portal/documentation/blob/master/documentation/widget_embedded.md>

What about embedding a widget inside another widget? The modal widget is nothing more than a modal dialog wrapper that loads another widget inside of it. The modal widget has a few options that help with loading and closing:

Options:

* embeddedWidgetId : string
* embeddedWidgetOptions : object
* afterOpen : callback
* afterClose: callback
* beforeRender: callback

Follow along to see how you can use the modal widget to embed the sn-desktop-filter widget in the card list heading:

1. Open the **Card List** widget
2. Use the following code for the HTML Template:

<div>

<sp-widget widget="c.filterModal" ng-if="c.filterModal"></sp-widget>

<div class="panel panel-default">

<div class="panel-heading">

<a href="javascript:void(0)" ng-click="c.showFilter($event)" class="pull-right"><span class="glyphicon glyphicon-search"></span></a> <span class="h3 panel-title">{{::c.options.title}}</span>

</div>

<div class="panel-body">

<workspace-card ng-repeat="row in c.data.rows track by row.sys\_id"

row="::row"

primary-field="::c.data.primaryField"

display-field="::c.options.display\_field"

priority-field="::c.options.priority\_field"

fields="::c.data.fields"

card-fields="::c.cardFields"></workspace-card>

</div>

</div>

</div>

1. Add the following 2 functions anywhere inside your client controller:

c.showFilter = function showFilter(event) {

var filterModalCtrl;

event.preventDefault();

event.stopPropagation();

var unregister = $scope.$on("snfilter:update\_query", function(e, query) {

e.stopPropagation();

e.preventDefault();

console.info("new Query", massageEncodedQuery(query));

// Todo: call webservice for data

filterModalCtrl.close();

});

var filterModal = angular.copy(c.data.filterModal);

filterModal.options.afterOpen = function(ctrl){

filterModalCtrl = ctrl;

$scope.$broadcast("snfilter:initialize\_query", massageEncodedQuery(c.options.filter));

};

filterModal.options.afterClose = function() {

unregister();

c.filterModal = null;

filterModalCtrl = null;

};

c.filterModal = filterModal;

}

function massageEncodedQuery(query) {

return (query) ? query.replace(/CONTAINS/g, "LIKE").replace(/DOES NOT CONTAIN/g, "NOT LIKE") : query;

}

1. Your server script should look like this:

(function() {

/\* populate the 'data' object \*/

/\* e.g., data.table = $sp.getValue('table'); \*/

if (options.table) {

data.filterModal = $sp.getWidget('widget-modal', {'embeddedWidgetId': 'sn-desktop-filter', 'embeddedWidgetOptions': { table: options.table, initialQuery: options.filter }});

data.cardFields = $sp.getListColumns(options.table, options.view\_dv);

data.rows = [];

var gr = new GlideRecord(options.table);

gr.addEncodedQuery(options.filter);

gr.setLimit(25);

gr.query();

if (!data.primaryField) {

data.primaryField = gr.getDisplayName();

}

var fields = "sys\_id," + data.primaryField + "," + data.cardFields;

data.fields = $sp.getFieldsObject(gr, fields);

while(gr.next()) {

var row = {};

$sp.getRecordDisplayValues(row, gr, fields);

data.rows.push(row);

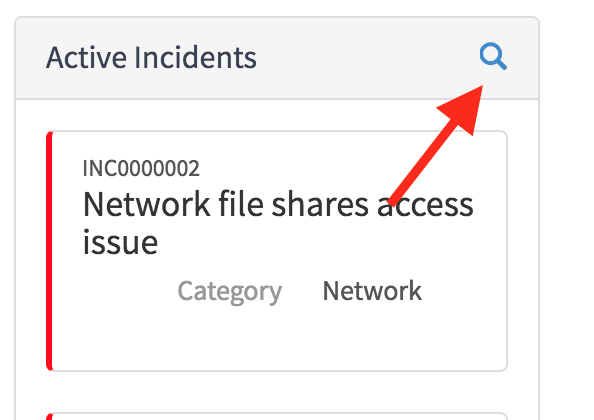
}

}

})();

1. Save the widget
2. Test the widget using the incident workspace page in a new tab. Go to: [instance]/$sp.do?id=iw

You should see a new glyph icon in the header of the card list

1. Click on the filter icon. The filter builder from the platform UI List v3 will be loaded in a modal.  
   
2. Make changes to the filter and click Run. Nothing happens yet beyond a console log statement showing the updated query. In the next section, you will see how to fetch new data for the card list using a web service and the new query.



# Using data & events across widgets

So far the incident workspace page only has one widget but this is the perfect time to think about sharing data and interactions between widgets. As soon as you add another widget to the page you will probably ask yourself one of these questions:

* How do I keep my widgets in sync when changing records or filters?
* How can my widgets share context?
* How do I maintain and persist state?

You can easily drop widgets onto a page and have them all do their own thing. A portal homepage is a good example of that. In this case, all the widgets on the page are ultimately going to react to a single table&filter or a selected record.

To share data between widgets you can use a custom javascript dependency or an angular provider called a service. In this lab you will create a toolbar widget, data service, and then use that data service across your widgets.

## Add a toolbar widget

1. Using STUDIO click Create Application File.
2. Select Service Portal -> Widget. Click Create.
3. On the widget editor click **Create a new widget**. Use the following values:

Widget name: **Workspace header**  
Widget ID: **workspace\_header**

Click **Submit**.

1. Edit the option schema with the following values:  
   Label: **Title**  
   Name: **title**  
   Type: **string**

Label: **Table**  
Name: **table**  
Default Value: **incident**  
Type: **string**

Label: **Count expressions**  
Name: **count\_expressions**  
Hint: **Label,query;Label,query;...**  
Default Value: **Unassigned Incidents,assigned\_toISEMPTY;New Incidents,state=1**  
Type: **string**

1. Click **Save**
2. Enter this for the HTML Template:

<div class="workspace-header clearfix">

<div class="pull-right">

<div class="count" ng-repeat="counter in c.counters">

{{counter.count}}

<span class="count-label">{{::counter.label}}</span>

</div>

</div>

<div class="h2">

{{::c.options.title}}

</div>

</div>

1. Enter this for CSS:

$workspace-header-background: $gray-light !default;

$workspace-header-padding: 5px !default;

.workspace-header {

background-color: $workspace-header-background;

margin: 0px;

margin-bottom: 5px;

padding: $workspace-header-padding;

@include border-top-radius(3px);

@include border-bottom-radius(3px);

.count {

font-size: 3rem;

text-align: center;

display: inline-block;

margin-right: 20px;

.count-label {

font-size: 1.8rem;

display: block;

}

}

}

1. Enter this for Client Script:

function($http) {

/\* widget controller \*/

var c = this;

var countExpressions = c.options.count\_expressions.split(";");

c.counters = [];

for(var i=0;i<=countExpressions.length-1;i++) {

var parts = countExpressions[i].split(",");

var counter = makeCounter(parts[0], parts[1]);

runCounter(counter);

c.counters.push(counter);

}

function runCounter(counter) {

var url = "/api/now/stats/"+ c.options.table +"?sysparm\_query="+ counter.filter +"&sysparm\_count=true";

$http.get(url).then(function(response) {

counter.count = response.data.result.stats.count;

});

}

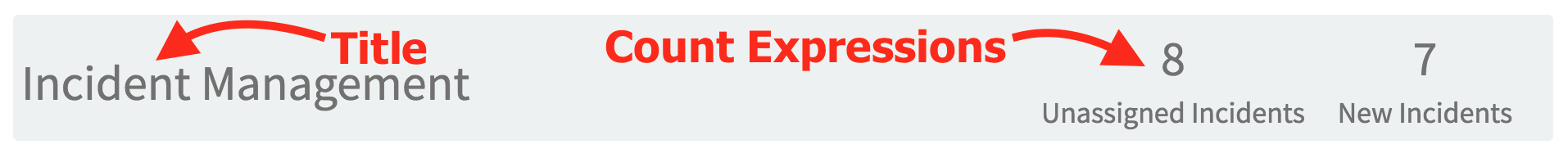
function makeCounter(label, filter) {

return {label: label, filter: filter, count: 0};

}

}

1. Save the widget.

In summary, this widget can show a heading and counts.  


When you configure the instance options for this widget you can provide count expressions in the following format:

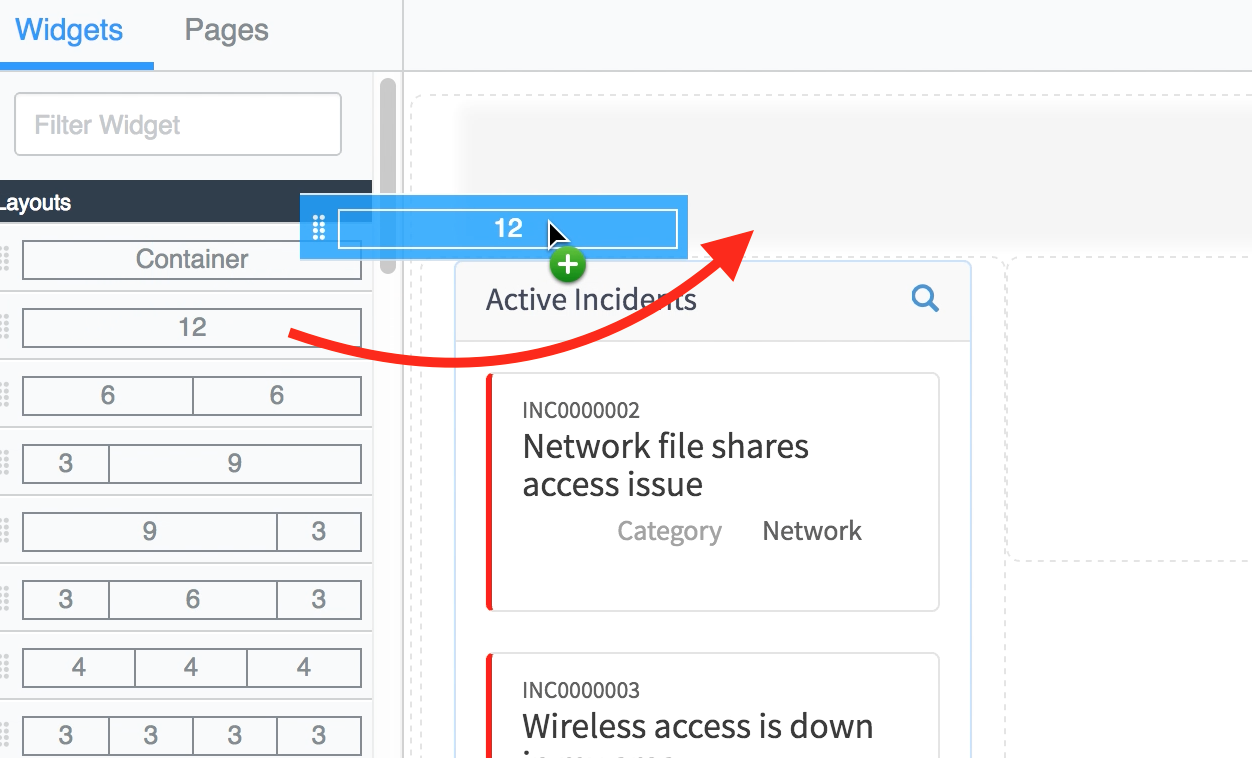
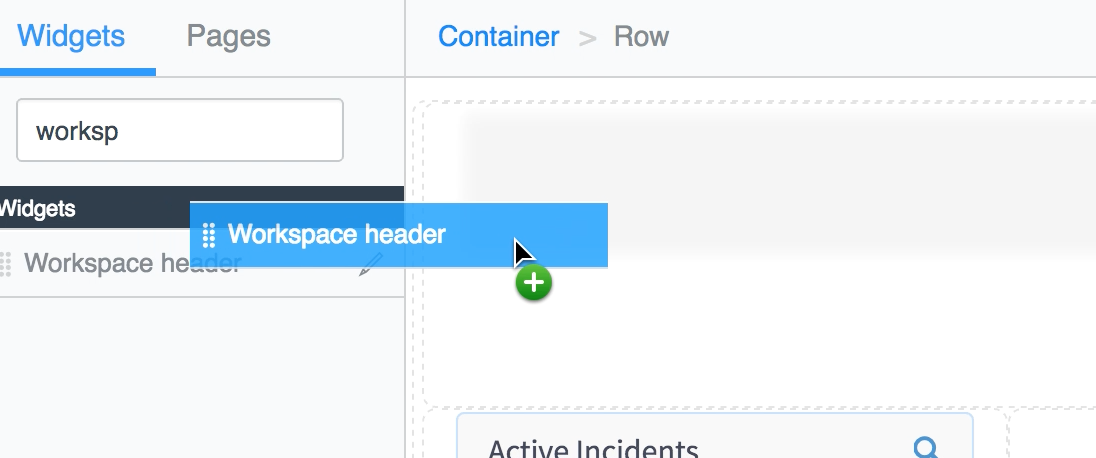
Label,filter

Label: **A string to show below the count**  
Filter: **An encoded query string**

Separate multiple count expressions with a semi-colon.

For each count expression, it will display the label and call the stats rest api to get a count of records matching the filter. This isn’t very intuitive to configure or efficient to execute but it can be easily modified later to use a shared data source. The takeaway here is that we aren’t using Server Script at all. You will soon see how to provide data to your widgets via rest endpoints.

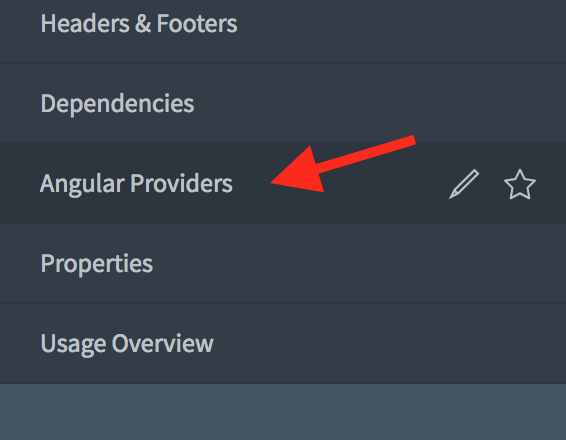
## Add the toolbar to the Incident Workspace page

1. Open the Service Portal designer in a new tab by going to [instacnce]/$spd.do
2. Open the Incident Workspace page.
3. Drag a new 12 column container to the very top of the page above the existing [3 | 9] layout like this:  
   
4. Drag the Workspace header widget into that new container:  
   

## Create an angular provider

Now that you have 2 widgets on a page using nearly the same data source, it’s time to create a data service that both widgets can use. It’s important to remember that since widgets are just angular directives, all the widgets on a page will use the same instance of an angular service. It’s just an instance of an object that is accessible by any widget that chooses to use it.

Angular Providers can’t be created from Studio so you need to create it using the platform.

1. Open your instance in a new tab:  
   [instance]/
2. Type **Service Portal** in the navigator and select **Angular Providers**  
   
3. At the top of the list click **New**
4. Add a new Widget Angular Provider with the following values:  
   Type: **Service**  
   Name: **workspaceData**  
   Client Script:

function(amb) {

var watcher;

var dataUpdatedHandlers = [];

function init(table, filter) {

if (watcher) {

watcher.unsubscribe();

}

if (table && filter) {

var watcherChannel = amb.getChannelRW(table, filter);

amb.connect();

watcher = watcherChannel.subscribe(function(message) {

if (!message.data) {

return;

}

dataUpdatedHandlers.forEach(function(fn) { fn.call(fn); });

});

}

}

return {

onDataUpdated: function(callbackFn) {

dataUpdatedHandlers.push(callbackFn);

},

initRecordWatcher: function(table, filter) {

init(table, filter);

}

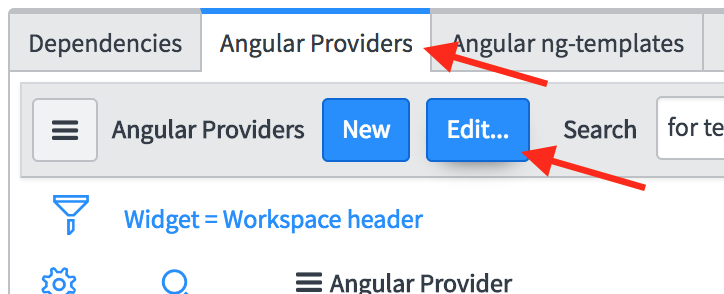
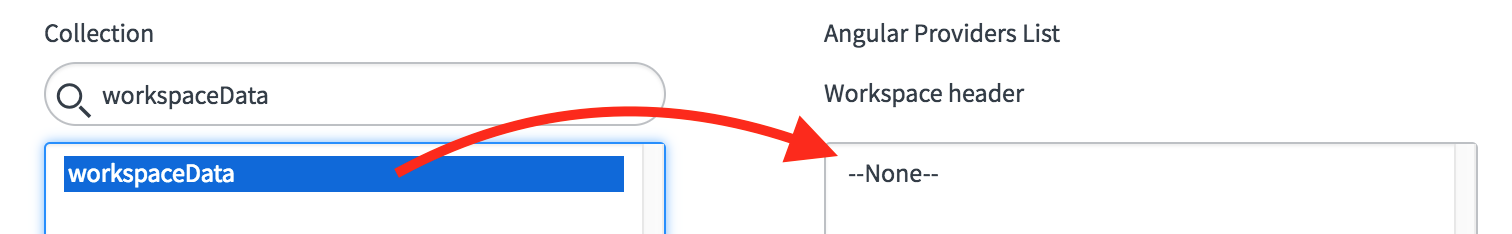
};

}

This service has two methods:

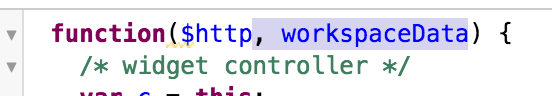
* onDataUpdated(callbackFn)  
  Use this method to subscribe to data updates. For the filter defined in initRecor dWatcher, when a record is added, updated or removed, the callback functions will be executed.
* initRecordWatcher(table, filter)  
  Use this method to start watching a table & filter for changes.

## Connect widgets together using data service

1. Using the navigator, open the Widgets list
2. Edit the **Workspace Header** widget
3. Scroll to the bottom of the form, open the **Angular Providers** related list, and click **Edit**  
   
4. Find **workspaceData** in the collection of angular providers. Double-click to move it to the right-side slush bucket.  
   
5. Click **Save**

Now repeat the same steps for the Card List widget so both widgets will import the workspaceData service.

## Use the workspaceData service inside the Card List widget

1. Using the widget editor, open the Card List widget
2. Inject the workspaceData service into the Client Script by adding it as a parameter to the controller function:  
   
3. Add the following code just below var c = this;

workspaceData.initRecordWatcher(c.options.table, c.options.filter);

workspaceData.onDataUpdated(function() {

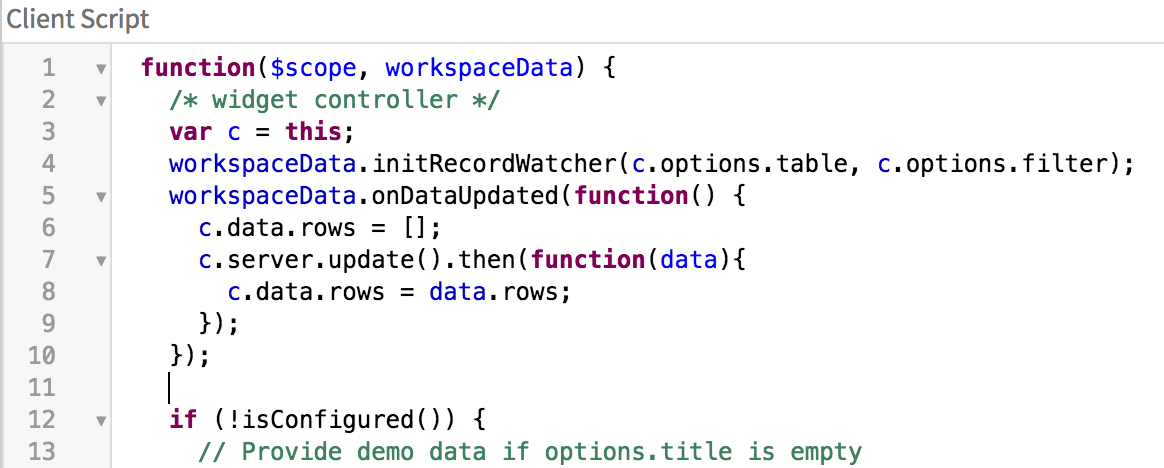
c.data.rows = [];

c.server.update().then(function(data){

c.data.rows = data.rows;

});

});

Your Client Script should look like this:  


1. Save the widget

Now when the record watcher fires, the widget will run c.server.update() and refresh the card list.***This isn’t optimal though*** and in the next section you will learn how to get the card list data from a webservice which utilizes a data stream and is much more performant than re-running the server script and serializing 25 records every time a record change is observed.

## Use the workspaceData service inside the Workspace Header widget

1. Using the widget editor, open the Workspace Header widget
2. Just like you did before, inject the workspaceData service into the Client Script controller function.
3. Add the following code just below var c = this;

workspaceData.onDataUpdated(function(){

c.counters.forEach(function(counter){ runCounter(counter); });

});