Service Portal Lab 3 - Guide

Lab 3.1: Create a Bootstrap Table 2

Lab 3.2: Writing the Server Script 4

Lab 3.3: Adding AngularJS to the Bootstrap form 6

Step 1 – ng-repeat over the table headings 6

Step 2 – ng-repeat over the table rows 6

Step 3 – ng-repeat over the table columns 6

Lab 3.4: Adding a Record Watcher to the list 7

Optional - Lab 3.5: Adding dynamic CSS classes 8

Optional - Lab 3.6: Adding information from the recordWatch result 8

# Lab 3.1: Create a Bootstrap Table

This lab will teach you how to load data, in that case active Incident records, into a table. Let’s start with building the HTML for a Bootstrap table, before we load data to populate the table with.

You can find documentation on how to create Bootstrap tables here:

* <https://getbootstrap.com/css/#tables>
* <http://www.w3schools.com/bootstrap/bootstrap_tables.asp>
  + **Hint:** when you click on **Try it yourself**, you will be able to copy/paste the full HTML of a table

You will create that table in a new Widget. Go to **Service Portal -> Service Portal Configuration**, open the **Widget** **Editor** and create a new Widget in which you will write your code. Provide a name for your new widget, create a new test page with your widget and then start creating the table.

The table heading (<th>) should contain the following fields:

* Number, Short Description, Caller ID, State, Priority, Created on

Also create a table row (<tr>) of demo data, the actual data should reside in table cells, represented by the <td> (column) tag.

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

In case you do not want to create your own table, here is a snippet that you can copy & paste.

<!-- Table without AngularJS -->

<div class="container">

<div class="table-responsive">

<table class="table">

<thead>

<tr>

<th>Number</th>

<th>Short Description</th>

<th>Caller ID</th>

<th>State</th>

<th>Priority</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>2016-07-08 14:30:00</td>

</tr>

</tbody>

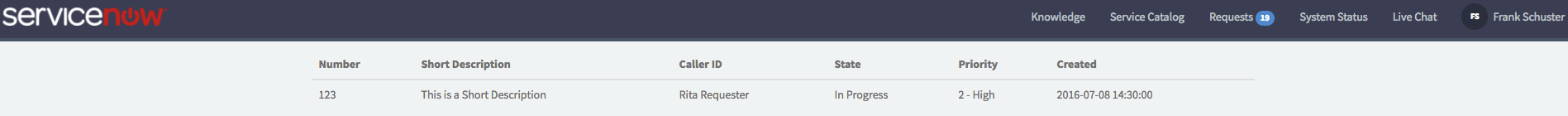
</table>

</div>

</div>

**Lab Validation:**

Your table should look something like this.

****

Now we have a table and a row with some data, but you probably do not want to create HTML for hundreds, maybe even thousands of records (I wouldn’t for even 2 records ☺). Let’s change this by loading data and adding AngularJS after.

# Lab 3.2: Writing the Server Script

Before we start adding AngularJS to our form, let’s make sure we have all the data we need.

To load the incidents we use the same API that we would also use on the Backend side of things: the GlideRecord object.

A pseudo code description on how to write the code, could sound like this:

Initialize a new, blank, array called **incidentsArr**, which is part of the data object.

After that we create a new GlideRecord on the Incident table to retrieve only active Incidents, ordered by the **number** field.

We will make use of the **$sp Service Portal Server Side API** to retrieve values, display values and even a set of field labels.

For our table we need the following data:

* Table Headings
  + Which represent the labels of all the fields that we want to see in the table
* Table Rows
  + Which represent the actual Incident records with their values

**ng-repeat** is the AngularJS directive that we will use. It awaits an array, so we are initializing an **incident** object (*var incident = {}*) that we are populating by iterating over the GlideRecord result.

Here is how that code would look like:

(function() {

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

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

//Initialize the 'incidentsArr' array in the data object

data.incidentsArr = [];

//Iterate over the Incident table to retrieve all active incidents ordered by number (descending - newest ones first)

var grIncident = new GlideRecord("incident");

grIncident.addActiveQuery();

grIncident.orderByDesc("number");

grIncident.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\_id", "state", "priority", "sys\_created\_on"];

//make use of the $sp API and get all field labels for the provided fields

data.labels = $sp.getFields(grIncident, 'number,short\_description,caller\_id,state,priority,sys\_created\_on');

while(grIncident.next()) {

var incident = {};

//get all display values for the fields below and write them into the incident object

$sp.getRecordDisplayValues(incident, grIncident, 'number,short\_description,caller\_id,state,priority,sys\_created\_on');

//get the value for sys\_id and also write it into the incident object

$sp.getRecordValues(incident,grIncident,'sys\_id');

//push the whole incident object into the incident array

data.incidentsArr.push(incident);

}

//log the data object to the browser console, so we can click through it and see our results

$sp.log(data);

return data;

})();

# Lab 3.3: Adding AngularJS to the Bootstrap form

Now that we have our data object populated, let’s access that data in the HTML section.

## Step 1 – ng-repeat over the table headings

At first we are going to replace all the <th> tags by repeating over a single <th> tag.

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

Within that property labels we have an attribute called **label**, that 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.

**This is the only <th> line that you should have now:**

<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

Our rows represent the number of records. Let’s say we have 20 records on the server, then we will need 20 rows. You probably already figured it out, but that’s how the <tr> repeat looks like:

<tr **ng-repeat**="incident in data.incidentsArr">

All our incidents are stored in the **incidentsArr** 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 we got changing values in each <td>. So unless you want to write a <td> for each field, here’s where the **data.fields** object comes into play. We have all technical field names stored in that object. Because of that we can now say for eachincident get the value of the actual field name that we are currently processing.

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

If you would not do that you would have to write:

<td>{{incident.number}}</td>

<td>{{incident.short\_description}}</td>

… etc.

# Lab 3.4: Adding a Record Watcher to the list

Wouldn’t it be awesome if our list would update itself any time a record is deleted or inserted into the Incident 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", "active=true", 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();

});

}

The Service Portal **spUtil** API includes the so called Record Watcher. We are basically just telling it to watch the Incident table and only records with the defined filter. If the watcher detects a change (like an insert, update or delete), then we want to execute our 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** classto the top <div> element.

<https://getbootstrap.com/components/#panels-tables>

**With that you successfully completed Lab 2 ☺**

The fully functional widget, including the code for the optional labs (**2.5 & 2.6**), can be found in the Box folder.

# Optional - Lab 3.5: Adding dynamic CSS classes

In this optional lab we want to utilize an AngularJS directive called **ng-class**. We will use this directive to conditionally apply styles to the **Priority** field. If an Incident has a Priority of 1, we want the background of the field to be **red**. For a Priority of 2, we will provide a **yellow** background. For everything elsethe background should be **green**.

Here are the components you will need:

* **ng-class** in the HTML part, which will call a function on the client script side
* 3 new CSS classes for the colors (with two attributes each for background and text color)
* One new function in the client script part to determine the right color for a field (it should have two input parameters: **field** and **fieldValue**). This function should only be triggered for the **priority** field. If the field is priority, we want to check the **fieldValue** and return the correct class.

# Optional - Lab 3.6: Adding information from the recordWatch result

Our list is already updating itself real-time. Let’s assume we want to display the information of the last recordWatch result to the user.

RecordWatch returns two objects, the **name** and **data** of the record.

Before you start, log the data object and inspect it in your browser to find out where we find the following information:

* The Time the record has been updated
* The action (was it an insert or a change?)
* The number of the record that has been updated or inserted

Once you found those values, make sure to make them available to the HTML part.

E.g.

c.recordNumber = data.name\_of\_the\_number\_field ☺

If you have done this, create a <div> in the HTML part, which should only be displayed if we have a recordWatch result (Hint: utilize **ng-if and set the value in the client script**).

Inside that <div> element, create <p> elements for the Last Update, the Action and the Incident Number. Have them display the values, which you determined in your client script.