Service Portal 200 Lab Guide

Preparation

Load, Preview & Commit the Backend Update Set:

- Name: **SP-Enablement-HackathonRegistrationComponents**
- Git: https://github.com/frankschuster/sp-enablement-training

This update set will provide you with the table & fields for the following labs.

Create a Hackathon Registration Page, a Container and e.g. a 12 column layout where you will display your widget in.

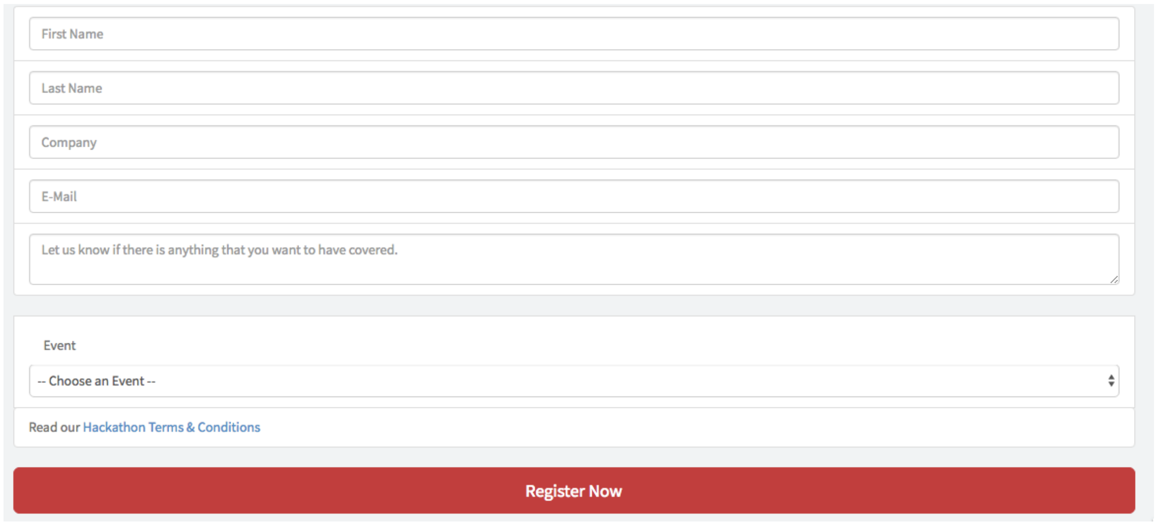
Lab 1.1: Create a Bootstrap Form

Create a Bootstrap form with the following fields:

- First Name (Text Input)
- Last Name (Text Input)
- Company (Text Input)
- E-Mail (Text Input)
- Event (Select Basic)
- Comments (Input Text Area)
- A Submit Button

Hint: Do not make your life to hard, use Bootstrap form builders to quickly create forms, e.g.: http://bootsnipp.com/forms

Lab Validation:
Your form should look something like this.


First Name
Last Name
Company
E-Mail
Let us know if there is anything that you want to have covered.
Event
-- Choose an Event --
Read our Hackathon Terms & Conditions
Register Now

Bonus: Add a Link to Terms & Conditions (you can add Terms & Conditions as a UI Page with simple HTML and link to that UI Page).

Lab 1.2: Adding AngularJS to the Bootstrap form

Now that you have built your form, let's add some AngularJS magic to bring that form to life.

When you've worked through tutorials or have seen some Angular examples you will know that everything in Anguglar is most of the time prefixed with **$scope**. Service Portal uses the so called **Controller As** notation per default, so instead of writing $scope.myVariable we will write **c.myVariable**.

Step 1 – ng-model

The first thing we do is that we will add the **ng-model** directive to all our input fields. That way we bind input fields to the Angular scope and make the value of those fields accessible to the Client and Server Script.

As we will add form validation later on you can already add an **ng-required="true"** to the input field as well. Make sure all of the input fields have an **id** attribute as well as the **name** attribute (Angular form validation will go against the **name** attribute).

Example:

```
<input type="text" ng-model="c.first_name" value="" class="form-control"
name="first_name" id="first_name" placeholder="First Name" ng-
required="true"/>
```

Step 2 – ng-submit

The next step is to add a function that should be executed on the client when the user submits the form.

We are using the **ng-submit** directive for this. Make sure that the **name** of your form is passed in as a parameter to the **submitForm** function.

$valid is an AngularJS property that tells whether an item is valid based on the rules that we will place. That means that our submitForm function will pass either **true** or **false** depending on if the form is valid or not.

We also add **novalidate** to the form tag since we do not want the standard HTML5 form validation, because we are building our own.

Example:

```
<form name="registration" ng-submit="c.submitForm(registration.$valid)"
role="form" novalidate>
```

Step 3 – Form validation with AngularJS & Bootstrap

As we have already added **ng-required** to our inputs we can now add some Messages to the form that we will display when a field does not meet the validation criteria.

ng-required:

Let's assume you added ng-required to the First Name input field, we could then create something like this in the same <div> as your <input> field is:

```
<p ng-show="registration.first_name.$invalid && c.submitted" class="help-
block">Your First Name is required.</p>
```

Within that <p> tag we utilize the **ng-show** directive to determine when to show our error message. The conditions for showing that element are:

- The first name within the registration form is invalid (true if the field is empty since we set required to **true**)
- Our form has to be submitted (this is a value we are going to set in **Lab 1.3 – Writing the Client Script**, but you can already set this here)

We also add the Bootstrap **help-block** class to add some styling to our error message.

Hint: All our error messages will only appear when the user clicks the **Register Now** button (this will be controlled via the Client Script from **Lab 1.3**). Right now we are just adding the validations.

ng-minlength & ng-maxlength:
With using **ng-min/maxlength** we can determine the minimum or maximum characters for a field.

$error is an object hash, containing references to controls or forms with failing validators (e.g. minlength or maxlength).

Hint: you will find all $error properties here (search for $error):
https://docs.angularjs.org/api/ng/type/form.FormController

Example:

```
<!-- Company -->
<div class="list-group-item">
<input type="text" ng-model="c.company" value="" class="form-control"
id="company" name="company" placeholder="Company" ng-required="true" ng-
minlength="3" ng-maxlength="10"/>

<!—formName.fieldName.
<p ng-show="registration.company.$error.minlength && c.submitted"
class="help-block">Company is too short.</p>
<p ng-show="registration.company.$error.maxlength && c.submitted"
class="help-block">Company is too long.</p>
<p ng-show="registration.company.$invalid && c.submitted" class="help-
block">Your Company is required.</p>
</div>
```

Step 4 – Adding a Message Block

When the user submits the form and everything was good (or not) we want to show the result of the transaction immediately. For this we are going to use **Bootstrap Alerts**.

```
<div ng-if="c.message" id="errorMessage" class="alert alert-danger"
role="alert">{{c.message}}</div>

<div ng-if="c.success" id="successMessage" class="alert alert-success"
role="alert">{{c.success}}</div>
```

The **message** and **success** variables are both set on the client. We can access them by using the AngularJS **{{ c.myVariable }}** notation.

Hint: There is also another way of binding variables which can increase performance, but which is only a **one-way** binding (meaning that this variable is going to be undefined once we return to the server). The notation is: **::c.myVariable.**

For more information read up on this here:
https://www.binpress.com/tutorial/speeding-up-angular-js-with-simple-optimizations/135

Step 5 – Adding (CSS) Classes conditionally:

When we run into an error on a specific field we want to add according error handling classes to the input field. Do achieve that we use the **ng-class** directive.

What we are doing here is adding the class **has-error WHEN** the last_name field is invalid **AND** our form has been submitted.

```
<!-- Last Name -->
<div class="list-group-item" ng-class="{ 'has-error' :
registration.last_name.$invalid && c.submitted }">
<input type="text" ng-model="c.last_name" value="" class="form-control"
name="last_name" id="last_name" placeholder="Last Name" ng-required="true"
/>
<p ng-show="registration.last_name.$invalid && c.submitted" class="help-
block">Your Last Name is required.</p>
</div>
```

Step 6 - Loading our Events from the Server

To populate our Select Variable we make use of the **ng-options** directive.

```
<select id="select_event" name="select_event" class="form-control" ng-
model="c.select_event" ng-options="event.name for event in c.data.events"
ng-required="true">
<option value="" disabled>-- Choose an Event --</option>
</select>
```

This basically reads like **for every event** in **c.data.events** (which we will populate from our Server Script) print out the **event.name** of each event as a Select option.

This Select Box won't be populated with anything until we write our Server Script, but we can already prepare it.

Lab 1.3: Adding the CSS

Add the following lines into your CSS section since we are using some of the classes:

```
#registerNow {
  background: #D3232C;
  border-color: #D3232C;
}

.noresize {
  resize: none;
}

.vresize {
  resize: vertical;
}

.hresize {
  resize: horizontal;
}

.errorIcon {
  top: 10px;
  right: 10px;
}
```

Lab 1.4 – Adding the Client Script

The Client Script serves as the **Angular Controller**.

Step 1 – Initializing the Controller

The first thing we do is to initialize **c** by adding the following line:

```
var c = this;
```

Step 2 – Writing the Submit Form function

Our Submit function should take a parameter called **isValid**. The value in this variable is determined by the call of our submit function within the **ng-submit** that we defined in the HTML Template.

Our function call should therefore look like this:

```
c.submitForm = function(isValid) {
```

Now that we sent the form we actually want to trigger the validation.
We already set a variable in the HTML which was called **submitted**.
Let's set this one in the client script by adding the following line:

```
c.submitted = true;
```

After we have done this it's time to prepare the **data object** with our values from the form, but only in case our form is valid – if not, then we obviously do not want to do anything.

This is how we do it:

```
// check to make sure the form is completely valid
if (isValid) {

//Write variables into the data object
c.data.first_name = c.first_name;
c.data.last_name = c.last_name;
c.data.email = c.email;
c.data.company = c.company;
c.data.event = c.select_event;
c.data.eventSysId = c.select_event.sys_id;
c.data.comments = c.comments;

}
```

Now that we assigned our form variables to the **data** object our Server Script will have access to this data the moment we add a simple line:

```
c.server.update();
```

If you do not want to do anything after that you are good, but in our example we also want to show the user if the registration was successful or not. Do achieve that we are adding a **Promise** to the update function.

```
c.server.update().then(function(response) {

if (response.status == 'success') {
$('#errorMessage').hide();

c.success = response.message;

setTimeout(function() {
    window.location.href = "?id=index";
}, 5000);


} else if (response.status == 'error') {
c.message = response.message;
}
});
```

The promise function evaluates a **status** variable that will also be set by the server script. In case we ran into an error before the first thing we do is hide the **errorMessage** <div> with **jQuery**.

After that we populate the **success** variable with the **message** that we also set in the Server Script. That will cause our success <div> to show up since we now have a value for this variable.

setTimeout is a method calls a function or evaluates an expression after a specified number of milliseconds. In our case we are using the **window** object of the Browser to redirect the user to the **index** page after 5 seconds.

Lab 1.5 – Writing the Server Script

Step 1 – Loading the Events from the Event Table:

To load events we are using some normal GlideRecord magic.
Be aware that the Server Script is already executed when you load a page (and the widgets). For us that means that we obviously only want to load the Events **initially**. When a user submits the form we do not need to load the events again, for this we are adding the **if(!input)** check – so only when the form loads and the input object is therefore not defined yet, we will execute the GlideRecord script.

```
/* Get Events for Event Select Box. Only load them initially */
    if(!input) {
          var events = [];

          var grEvent = new GlideRecord("u_hackathon_event");
          grEvent.addActiveQuery();
          grEvent.query();
```

```
        while(grEvent.next()) {
              events.push({
                    "name": grEvent.getDisplayValue("u_event_name"),
                    "sys_id": grEvent.getUniqueValue()
              });
        }

        //make events available to the client
        data.events = events;
    }
```

ng-repeat on the HTML side awaits an array so we are initializing an **events** array that we are populating by iterating over the GlideRecord result.

Quite contrary to that we have a piece of code that we only want to execute when we have some input defined, which is the case when the user submitted the form.
That piece of code should then do some validation against the Registration [u_hackathon_registration] and the User [sys_user] table – here we will only check if the email address is already present in either one of the two tables. If yes we will throw a duplicate error, if not we will actually start inserting the form data as a new record on the Registration table.

So before we start inserting add the following code to your server script for validation:

```
function validateRegistration(email) {
    var isNoDuplicate = true;

    //abort if user already exists
    if (userExists(email)) {
          data.message = gs.getMessage("There is already a user record with
that email address. Please pick another one or reset your password via the
'Login' form.");
          data.status = "error";
          isNoDuplicate = false;
    }

    //abort if registration already exists
    if (registrationExists(email)) {
          data.message = gs.getMessage("There is already a pending
registration request for that email address. Please pick another one or
reset your password via the 'Login' form.");
          data.status = "error";
          isNoDuplicate = false;
    }

    return isNoDuplicate;
}


function registrationExists(email) {
    var reg = new GlideRecord("u_hackathon_registration");
    reg.addQuery("u_email", email);
    reg.addQuery("u_state", "pending");
    reg.query();
    return reg.hasNext();
}

function userExists(email) {
    var usr = new GlideRecord("sys_user");
    usr.addQuery("email", email);
    usr.query();

    return usr.hasNext();
}
```

The final piece is writing the GlideRecord part where we actually create a new registration record.

```
/* Only execute when we have form input */
if(input) {

if(validateRegistration(input.email)){
    var grHackathonReg = new GlideRecord('u_hackathon_registration');
    grHackathonReg.initialize();
    grHackathonReg.setValue('u_first_name', input.first_name);
    grHackathonReg.setValue('u_last_name', input.last_name);
    grHackathonReg.setValue('u_email', input.email);
    grHackathonReg.setValue('u_company', input.company);
    grHackathonReg.setValue('u_event', input.eventSysId);
    grHackathonReg.setValue('u_state', 'pending');
    grHackathonReg.setValue('u_comments', input.comments);
    var userRegSysId = grHackathonReg.insert();

    if (!gs.nil(userRegSysId)) {
          data.status = "success";
          data.message = gs.getMessage("Your registration has been
submitted. You will be redirected to the homepage in a few seconds. If you
are not being redirected click on the Logo and you will get back to the
homepage.");
    } else {
          data.status = "error";
    }
}

return data;
}
```

For your convenience you can find the whole Widget here on Github:

- https://github.com/frankschuster/sp-enablement-training