Service Portal Lab 4 - Guide

Lab 2.1: Prepare the Backend 2

Lab 2.2: Writing the first part of the Server Script 3

Step 1 – Loading the Contact Types from the Choice Table: 3

Lab 2.3: Add the Categories to the view (HTML) 4

Lab 2.4: Bind the other HTML types to variables 4

Lab 2.5: Adding form validation to the form 5

Step 1 – Preparing the <form> tag 5

Step 2 – Adding validation criteria to the first name & last name fields 5

Step 3 – Adding Bootstrap Error Classes to our inputs 7

Step 4 – Adding a Message Block 7

Lab 2.6: Submitting the form 8

Lab 2.7: Adding the Client Script 9

Step 1 – Initializing the Controller 9

Step 2 – Writing the Submit Form function 9

Lab 2.8: Inserting into the Database via the Server Script 11

Lab 2.9: Adding CSS 12

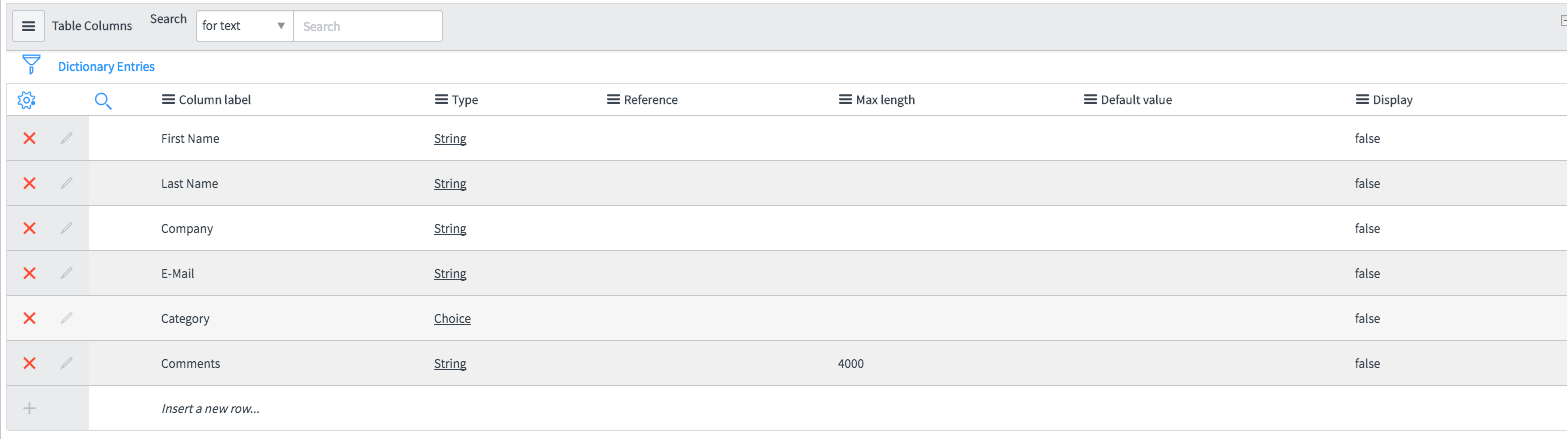
Remember Lab 2? Let’s add some complexity to it. We will load data from the server, write data back to the server and validate the form on submission.

# Lab 2.1: Prepare the Backend

Before we can modify the Widget, we need to make sure we do have have a table, that will contain our contact requests.

Create a new table called “Contact Request”.

Add the following fields to it, those will be the database fields, that we write our form values in:

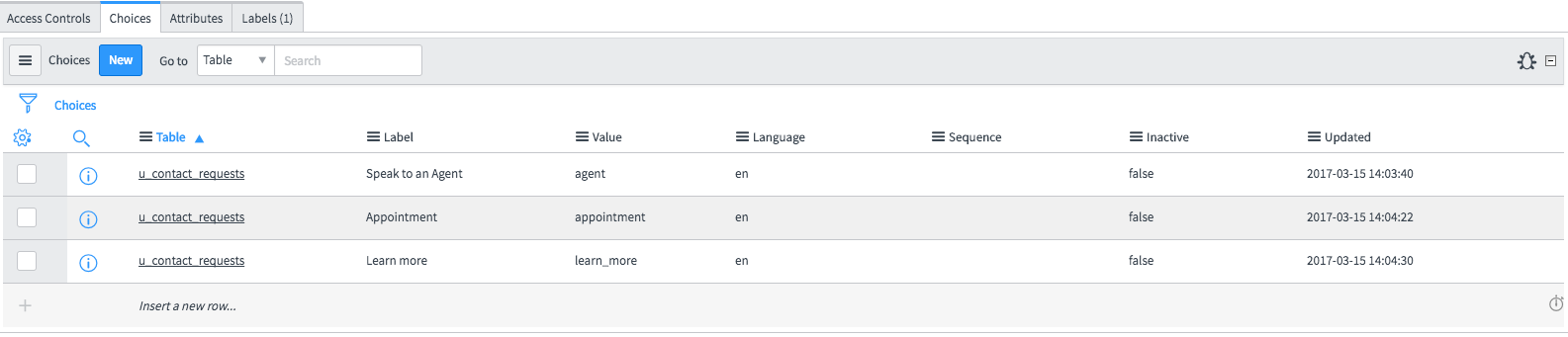


**We award bonus points** for making the company field a reference field and have your server script do a lookup of the form company value and match that value against the company table ☺

After you have added your fields, let’s add some choice values to the category field.

Open the Dictionary Record for the Category field or navigate to the **Contact Request Form -> right-click on the label -> Show Choice Lis**t (I always like to use this, since I have full control over the technical values, which is what I wouldn’t have when I use Configure Choices).

Here’s how it should look like:



# Lab 2.2: Writing the first part of the Server Script

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

**HINT:** Do not copy/paste this code from the Box preview, open up the Word document, copy it in the correct widget field, mark everything (**Cmd + A**) and then format the code with **Shift + Tab**.

## Step 1 – Loading the Contact Types from the Choice Table:

The first thing we want to replace are the static select values, with actual values from the contact request table.

To load the categories, we are using some standard 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 categories **initially**. When a user submits the form we do not need to load the categories 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 categories for the "I would like to" select box. Only load them initially \*/

if(!input) {

var categories = [];

var grCategory = new GlideRecord("sys\_choice");

grCategory.addQuery("name", "u\_contact\_requests");

grCategory.addQuery("element", "u\_category");

grCategory.query();

while(grCategory.next()) {

var category = {};

$sp.getRecordDisplayValues(category, grCategory, "sys\_id, value, label");

categories.push(category);

}

//make categories available to the client

data.categories = categories;

}

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

We leverage the **$sp** Service Portal Server API, to retrieve the display values of sys\_id, label & value. We write the result into a **category** object, which will then be pushed into the array.

# Lab 2.3: Add the Categories to the view (HTML)

To populate our Select Variable we make use of the AngularJS **ng-options** directive. We will also have to bind the field to a variable, which we will use later to populate the database, this is achieved with the **ng-model** directive. By writing the variable directly into the data object, we will have it available right away on the server via **input.selected\_category**.

Change the select box part in your HTML to:

<select id="category" name="category" class="form-control" ng-model="c.data.selected\_category" ng-options="category.label for category in c.data.categories">

<option value="" disabled>-- Choose a Category --</option>

</select>

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

# Lab 2.4: Bind the other HTML types to variables

Go through your HTML and add an AngularJS **ng-model** directive to all the input fields, i.e. ng-model=”c.data.first\_name”.

# Lab 2.5: Adding form validation to the form

## Step 1 – Preparing the <form> tag

In order to allow AngularJS form validation, we turn off HTML5 validation by adding the **novalidate** attribute to the form element. Change the very first line in your HTML to look like this:

<form class="form-horizontal" name="**contactForm**" ng-submit="c.submitForm(contactForm.$valid)" **novalidate**>

Then form name **contactForm** is extremely important, as we will use it in every field validation. The ng-submit function will be explained later, but essentially it will trigger a function on the client and contain a Boolean that is either true (form is valid) or false (form is invalid).

## Step 2 – Adding validation criteria to the first name & last name fields

In this step, we want to make the first name mandatory and we want to add a minimum length of 3 chars and a max length of 6 chars to the last name field.

<!-- Text input-->

<div class="form-group">

<label class="col-md-4 control-label" for="first\_name">First Name</label>

<div class="col-md-4">

<input id="first\_name" ng-model="c.data.first\_name" name="first\_name" type="text" placeholder="First Name" class="form-control input-md" **ng-required="true"**>

**<p ng-show="contactForm.first\_name.$invalid && !contactForm.name.$pristine" class="help-block">Your First Name is required.</p>**

</div>

</div>

First, we’ll add an **ng-required=”true”** to our input field, that will tell our form validation method from step 1, that this field needs to be filled.

Second, we’ll add a <p> tag, which will show an error message below the input field.

We use the **ng-show** directive, to determine two things if the item has been used and if all rules apply or not.

Here is a list of all available properties, that you can add:

|  |  |  |
| --- | --- | --- |
| Property | Class | Description |
| $valid | ng-valid | *Boolean* Tells whether an item is currently valid based on the rules you placed. |
| $invalid | ng-invalid | *Boolean* Tells whether an item is currently invalid based on the rules you placed. |
| $pristine | ng-pristine | *Boolean* True if the form/input **has not** been used yet. |
| $dirty | ng-dirty | *Boolean* True if the form/input **has** been used. |
| $touched | ng-touched | *Boolean* True if the input **has** been blurred. |

**ACCESSING ANGULAR FORM PROPERTIES**

* **To access the form:** <form name>.<angular property>
* **To access an input:** <form name>.<input name>.<angular property>

Now we’ll add some rules to the last name field:

<!-- Text input-->

<div class="form-group">

<label class="col-md-4 control-label" for="last\_name">Last Name</label>

<div class="col-md-4">

<input id="last\_name" ng-model="c.data.last\_name" name="last\_name" type="text" placeholder="Last Name" class="form-control input-md" **ng-minlength="3" ng-maxlength="6" ng-required=”true”**>

**<p ng-show="contactForm.last\_name.$error.minlength" class="help-block">Last Name is too short.</p>**

**<p ng-show="contactForm.last\_name.$error.maxlength" class="help-block">Last Name is too long.</p>**

**<p ng-show="contactForm.last\_name.$invalid" class="help-block">Your Last Name is required.</p>**

</div>

</div>

We will leverage the **ng-minlength / ng-maxlength** directives for the length of the string. Then we will add two messages, one for the min length and one for the max length.

## Step 3 – Adding Bootstrap Error Classes to our inputs

Because we are using Bootstrap, we can use the **has-error** class, to add a red border and red text around our <form-group>.

**ng-class**allows us to add classes based on an expression. In this case, we want to add a has-error class to our **form-group** if an input is $invalid and not pristine.

The way it works is ng-class="{ <class-you-want> : <expression to be evaluated > }".

For our first name, we will change the outer <div class=”form-group”> to:

<div class="form-group" **ng-class="{ 'has-error' : contactForm.first\_name.$invalid && !contactForm.first\_name.$pristine }"**>

And the outer <div> of last name would change to:

<div class="form-group" **ng-class="{ 'has-error' : contactForm.last\_name.$invalid && !contactForm.last\_name.$pristine }"**>

Save your widget and notice that no error messages should appear right now.

If you start typing in the last name field, you will notice that the error message will appear right away (string is too short).

## Step 4 – Adding a Message Block

When the user submits the form and the request has been sent succesfully 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="message" class="alert" ng-class="{ 'alert-success': c.message == 'success', 'alert-danger': c.message == 'error'}" role="alert">{{c.message\_text}}</div>

The **message** and **message\_text** variables, will both be set once we inserted data into the database later. We can access them by using the AngularJS **{{ c.myVariable }}** notation.

Notice that we also use the ng-class directive, to contextually apply a **success** or **error** class, depending on the value of the **message** variable.

**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 would be: {{**::c.myVariable}}.**

For more information, read up on it here:

<https://www.binpress.com/tutorial/speeding-up-angular-js-with-simple-optimizations/135>

# Lab 2.6: Submitting the form

To submit our form, we will have to adjust the <button> tag and transform our button into a button of type **submit**.

<button **type="submit"** id="send\_request" class="btn btn-lg btn-primary btn-block">${Send Request}</button>

The **${Send Request}** notation is the markup you would use for translating widget content. This will try to find a Message with the key **Send Request** within the Messages [sys\_ui\_message] table.

Now, if you reload and click Send, our error messages would not appear: why? Because we added a $pristine check to them, meaning a user would have to touch them first – let’s make them appear for both cases (somebody types, somebody submits it empty without touching the field).

Remember the **ng-submit** function we defined in the <form> tag? Let’s use it.

Open the **Client Script** section of your widget and add the following lines:

c.submitForm = function(isValid) {

c.submitted = true;

};

The **submitForm** function is the one we call in the HTML form tag. We provide the result of the AngularJS form validation check as a Boolean to the function constructor.

We’ll use that in a second, but first let’s set a variable **c.submitted** to **true**.

Once we have done this, we will have to revisit our form validation and add that variable to the check and classes.

**First Name:**

<div class="form-group" ng-class="{ 'has-error' : contactForm.first\_name.$invalid && !contactForm.first\_name.$pristine **|| (c.submitted && contactForm.first\_name.$invalid)** }">

<p ng-show="contactForm.first\_name.$invalid && !contactForm.first\_name.$pristine **|| (c.submitted && contactForm.first\_name.$invalid)"** class="help-block">Your First Name is required.</p>

**Last Name:**

<div class="form-group" ng-class="{ 'has-error' : contactForm.last\_name.$invalid && !contactForm.last\_name.$pristine **|| (c.submitted && contactForm.last\_name.$invalid**)}">

<p ng-show="contactForm.last\_name.$error.minlength || c.submitted && contactForm.last\_name.$error.minlength" class="help-block">Last Name is too short.</p>

<p ng-show="contactForm.last\_name.$error.maxlength || c.submitted && contactForm.last\_name.$error.maxlength" class="help-block">Last Name is too long.</p>

<p ng-show="contactForm.last\_name.$invalid && c.submitted" class="help-block">Your Last Name is required.</p>

# Lab 2.7: Adding the Client Script

Now, that our HTML is prepared, we can start providing our data to the server – we will do this through 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 (if it is not already there):

var c = this;

## Step 2 – Writing the Submit Form function

We already the submit function, now we will add a bit more code to it.

c.submitForm = function(isValid) {

c.submitted = true;

if(isValid) {

c.server.update();

}

};

This will do two things: we will only send data to the server, if the form is valid, meaning all Angular validations have passed their tests.

We will send the data object to the server, by writing **c.server.update()**.

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

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

if (response.message == 'success') {

c.message = "success";

c.message\_text = "Thank you for contacting us, we will get back to you as soon as possible.";

} else if (response.message == 'error') {

c.message = "error";

c.message\_text = "Oops, something went wrong on our side. Apologies!";

}

});

The promise function evaluates a **message** variable that will also be set by the server script.

After that we populate the **message\_text** variable with some generic text. That will cause our success <div> to show up since we now have a value for this variable.

# Lab 2.8: Inserting into the Database via the Server Script

At this time our data object is populated, so we can start inserting the data into the server.

For this we’ll add a new snippet to the server script, this time we will check for **if(input)**, since the data object will be transformed into **input**. Make sure that all your field names are correct (they don’t necessarily have to match mine). Same counts for the **input**.variable values, you might have declared them differently in the HTML (**c.data.**yourVarName).

if(input) {

var grContactRequest = new GlideRecord("u\_contact\_requests");

grContactRequest.initialize();

grContactRequest.setValue("u\_first\_name", input.first\_name);

grContactRequest.setValue("u\_last\_name", input.last\_name);

grContactRequest.setValue("u\_company", input.company);

grContactRequest.setValue("u\_email", input.email);

grContactRequest.setValue("u\_category", input.selected\_category.value);

grContactRequest.setValue("u\_comments", input.comments);

var reqId = grContactRequest.insert();

if(reqId) {

data.message = "success";

} else {

data.message = "error";

}

}

Save and submit your form – the result should be a new record in your custom table and the display of the success message.

# Lab 2.9: Adding CSS

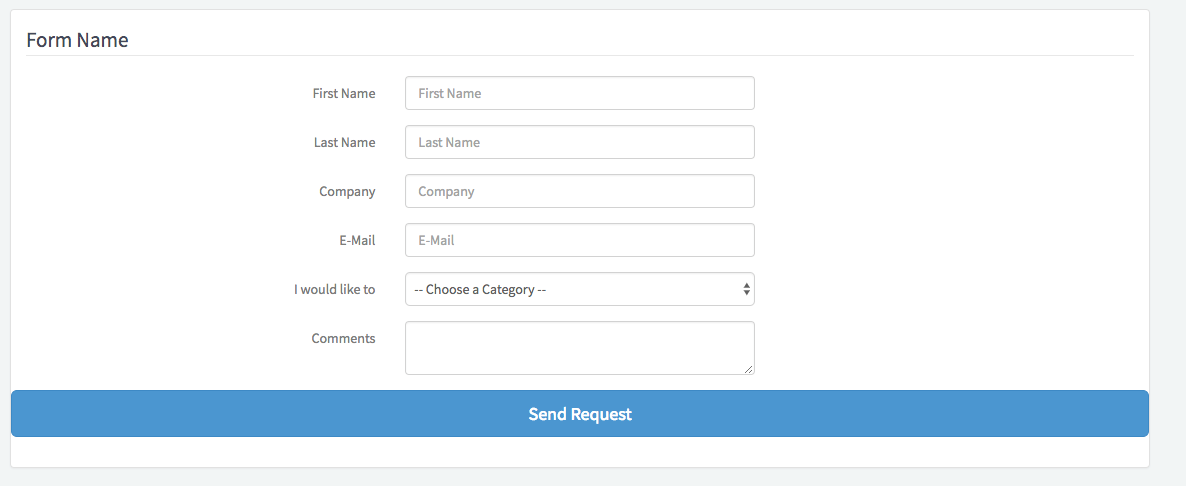
We do have a form now, but let’s be honest – that doesn’t look to good does it? ☺

We can improve this by adding Out of the Box Bootstrap classes and using custom CSS.

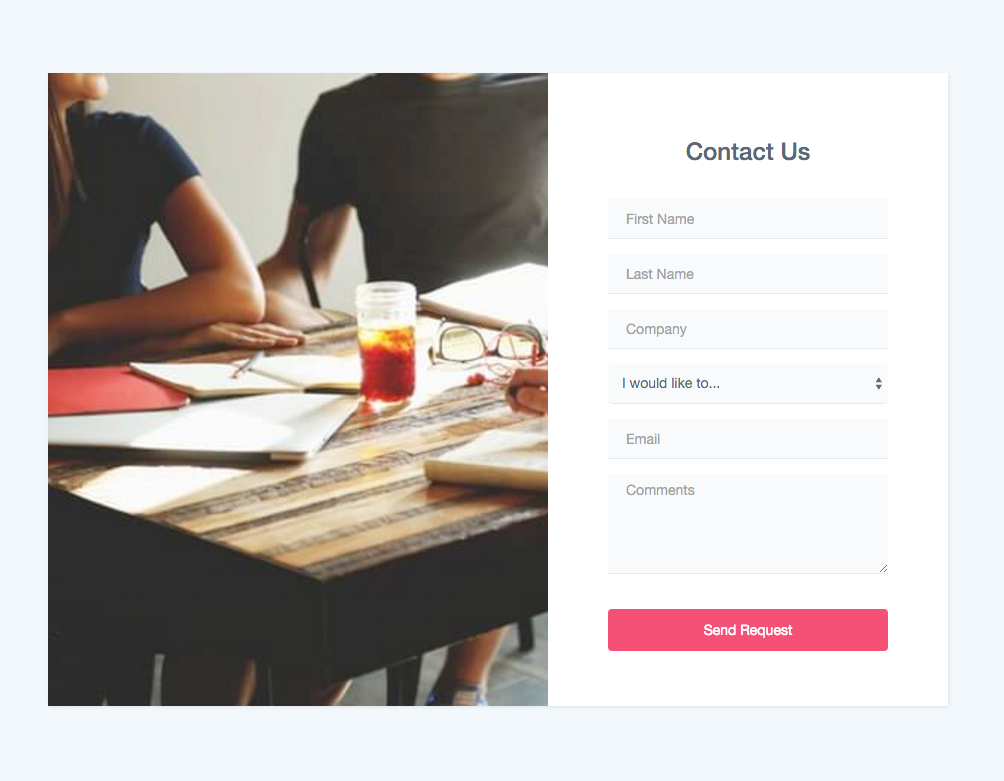
The first thing we will do is wrap a panel around the whole form.

Have a look at the Basic Example on: <http://getbootstrap.com/components/#panels> and try it out.

This is the expected result, after the panel has been added.



This is a little bit better, isn’t it? How about this:



The **only** thing we did here, is leverage what’s already out there and bring in the **CSS** of this form. You will certainly have to add those classes to your HTML input and form fields as well.

Some sources are:

<http://bootsnipp.com>

<http://wrapbootstrap.com>

<http://startbootstrap.com>

<http://bootstrapstudio.io> (Tool, no templates on the web)

You can find the widget we created in this exercise in the Box folder.