Contents

[**part 01 – NG16 Code Setup** 2](#_Toc141770185)

[**part 02 – Supporting Form Code** 4](#_Toc141770186)

[**part 03 – Form Models** 6](#_Toc141770187)

[**part 04 – Simple Validation** 7](#_Toc141770188)

[**part 05 – Adding Email** 10](#_Toc141770189)

[**part 06 – Select and Radio Buttons Form Control** 13](#_Toc141770190)

[**part 07 – Additional Template Form Features** 16](#_Toc141770191)

[**Appendix A – Angular Architectural Concepts** 19](#_Toc141770192)

[**Appendix B – Angular Directives** 21](#_Toc141770193)

[**Appendix C – Installation Issues** 22](#_Toc141770194)

[**Appendix D – @NgModule** 22](#_Toc141770195)

[**Appendix E – Form States** 23](#_Toc141770196)

Day01 Angular Template Driven Forms

# part 01 – NG16 Code Setup

This section assumes that you have already installed the latest Angular CLI. If you did not, please run the command **npm install -g @angular/cli** before proceeding. For this particular boot camp, I will use *skills* as the app folder, but you may use any other name you wish.

There is a starter file on GitHub that you may use to get started. This file is a snapshot of the code at the end of Part01, so it is NOT the beginning code, only the code after all the instructions were done in Part01.

1. From your root folder (Documents in my case), open a terminal window (or tab) to that folder and type the command **ng new skills**
2. Choose **N** for stricter type checking (if asked) and **N** for routing. Choose plain **CSS** for styles. To choose a CSS developer, use the arrow keys on the keyboard, however the default CSS should be auto selected, just hit **Enter**.
3. Use VS Code to open the skills folder and therefore the boiler plate application. navigate to **src->app->app.component.html**.
4. Remove all the code from the app.component.html template file except the following line (around line 333):

|  |
| --- |
| **<span>{{ title }} app is running!</span>** |

You may use any editor, you are not bound to VS Code

1. Add the following code in the template file to create a form with one input field:

|  |
| --- |
| **<hr /> <form>**  **<label for="fName">First name:</label>**  **<input type="text" name="fName"><br />**  **<input type="submit" value="Submit"><br />**  **</form>** |

The <hr /> is optional, it just adds a horizontal line for separation.

1. Angular supports forms with a special class (module), add this class in app.module.ts:

|  |
| --- |
| **import { AppComponent } from './app.component';**  **import { FormsModule } from "@angular/forms";**  **//**  **@NgModule({**  **… *other code here* …**  **imports: [**  **BrowserModule,**  **FormsModule**  **],** |

We will not be using this app.module file any time soon, so close it. Once we import FormsModule, it immediately recognizes all form tags and creates a top level FormGroup. This is done behind the scenes.

1. In app.component.ts file, we can start using the FormsModule. First import the NgForm module from @angular/forms:

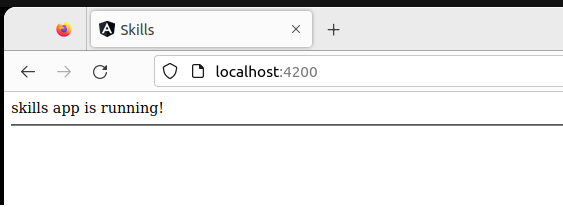
|  |
| --- |
| **import { Component } from '@angular/core';**  **import { NgForm } from "@angular/forms";**  **@Component({** |

1. Continuing from #5 above, add a template reference to the form tag and point it to ngForm (this gives access to the form and its controls):

|  |
| --- |
| **<form #userDetails = "ngForm" novalidate>**  **<label for="fName">First name:</label>**  **<input type="text" name="fName"><br />**  **<input type="submit" value="Submit"><br />**  **</form>** |

Now, userDetails represents this new FormGroup. Also add the novalidate attribute to prevent normal HTML validation. We will implement our own validation.

1. At this point, you may start the app. Open a terminal window and run the code ng serve. Once the app starts, open a browser and go to the localhost address port 4200:



1. Now, bind the HTML submit event of the form to Angular’s ngSubmit(). This ngSubmit() is called a **directive.** This directive will point to a function that will be written soon. Also pass the name of the form as a parameter, userDetails:

|  |
| --- |
| **<form #userDetails = "ngForm" (ngSubmit)="onSubmit()" novalidate>**  **<label for="fName">First name:</label>**  **<input type="text" name="fName"><br />**  **<input type="submit" value="Submit"><br />**  **</form>** |

Putting parenthesis (brackets) around ngSubmit means that we are listening for the form submission event. The onSubmit() method will be created in the component (.ts) file shortly. Your app will break since it cannot find the onSubmit() function.

# part 02 – Supporting Form Code

Angular has made available the FormsModule which was added to the *app* in Part 01. This action will create a JavaScript representation of our form once we declare one using HTML. It does not, however, add form controls. For that we need to add the ngModel directive to each control that we want to be part of the final submission. The ngModel directive together with the name of the control guarantees that the control will be part of the final form object that Angular controls. We imported the ngModel class in Part01 #6.

1. Create the onSubmit() method in the app.component.ts file, inside of the class:

|  |
| --- |
| **export class AppComponent {**  **title = 'skills';**  **onSubmit(){**  **console.log("Form Submitted");**  **}**  **}** |

At this point, the error from Part01 should go away and if you submit the form, the console.log() function will fire.

1. Not much is shown in the console window, but if we accept the form as an object, then we can log its value or any property of the form:

|  |
| --- |
| **onSubmit(userDetails : NgForm){**  **console.log("Form Submitted " + userDetails.value.fName);**  **}** |

The object being passed in here is of the type NgForm and it has a value property. This property contains those form controls that hold the values that the user types into the box.

1. Now that this method accepts an NgForm object, lets pass one during the submission of the form:

|  |
| --- |
| **<form #userDetails = "ngForm" (ngSubmit)="onSubmit(userDetails)" novalidate>**  **<label for="fName">First name:</label>** |

Just pass userDetails since it is a reference to the form. The novalidate attribute is the normal HTML that prevents the form submission accidentally.

1. If you logged just userDetails or userDetails.value, you get Objects in the console window. If you log fName you will get undefined.
2. So, to make any form control part of the final form object, simply add ngModel as an attribute to the form control code:

|  |
| --- |
| **<form #userDetails = "ngForm" (ngSubmit)="onSubmit(userDetails)" novalidate>**  **<label for="fName">First name:</label>**  **<input type="text" name="fName" ngModel><br />**  **<input type="submit" value="Submit"><br />**  **</form>** |

The value typed into the first name box is now showing in the console window. Notice the control has an HTML name attribute, it helps register the control.

1. Add a last name field in the same way as first name:

|  |
| --- |
| **<input type="text" name="fName" ngModel = "firstName"><br />**  **<label for="lName">Last name:</label>**  **<input type="text" name="lName" ngModel><br />**  **<input type="submit" value="Submit"><br />**  **</form>** |

1. At this point you can start experimenting with the onSubmit() method to see what kind of details you can get. If you tried to print just userDetails or userDetails.value you see objects and can’t go any further. You can however try the following to see what prints:

|  |
| --- |
| **onSubmit(userDetails : NgForm){**  **console.log("First name: " +**  **userDetails.controls["fName"].value +**  **", Last Name: " + userDetails.value.lName);**  **}** |

Soon we will start working with models, so getting values becomes a lot easier.

# part 03 – Form Models

If you think about it, a form is just a list of name/value pairs. Think of the field name as the *name* and what the user types into that field as the *value* so name/value. We can build a model to represent this arrangement.

1. Create a separate user.ts file in the app folder and insert the code below:

|  |
| --- |
| **export class User {**  **public firstName?: string;**  **public lastName?: string;**  **}** |

The ? signifies to TS that this field may be null. Name your file user.ts

1. To work with this new User class, we first import it into app.component.ts, create a property of type User and finally initialize the property inside of the constructor:

|  |
| --- |
| **import { User } from "./user";**  **@Component({**  **…other code…**  **})**  **export class AppComponent {**  **title = 'skills';**  **user : User = {};**  **constructor() {**  **this.user = new User();**  **};  onSubmit(userDetails : NgForm){** |

1. Update the template to reflect the User model:

|  |
| --- |
| **<form #userDetails = "ngForm" (ngSubmit)="onSubmit(userDetails)" novalidate>**  **<label for="fName">First name:</label>**  **<input type="text" name="fName" ngModel = "user.firstName"><br />**  **<label for="lName">Last name:</label>**  **<input type="text" name="lName" ngModel = "user.lastName"><br />**  **<input type="submit" value="Submit"><br />**  **</form>};** |

This sets up property binding which will come later

1. Also, now when we pass the form, we pass its value:

|  |
| --- |
| **<form #userDetails = "ngForm" (ngSubmit)="onSubmit(userDetails.value)" novalidate>**  **<label for="fName">First name:</label>**  **<input type="text" name="fName" ngModel = "user.firstName"><br />**  **<label for="lName">Last name:</label>**  **<input type="text" name="lName" ngModel = "user.lastName"><br />**  **<input type="submit" value="Submit"><br />**  **</form>};** |

1. Now in the class, when we accept the form object inside of onSubmit() we can specify that this is of the User type:

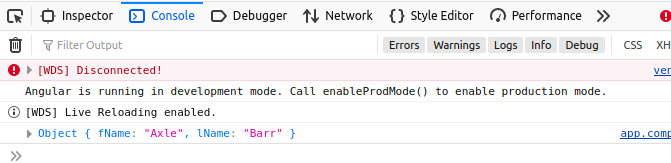
|  |
| --- |
| **constructor() {**  **this.user = new User();**  **};**  **onSubmit(userDetails : User){**  **console.log(userDetails);**  **}** |

The parameter here is of type User. It matches the form’s value.

1. The problem now is that we see the user and property in the input controls. To prevent this add square brackets, so property binding:

|  |
| --- |
| **<label for="fName">First name:</label>**  **<input type="text" name="fName" [ngModel] = "user.firstName"><br />**  **<label for="lName">Last name:</label>**  **<input type="text" name="lName" [ngModel] = "user.lastName"><br />**  **<input type="submit" value="Submit"><br />** |

Property binding also allows us to pass data from the class to the form directly



# part 04 – Simple Validation

We can achieve simple validation by adding required attributes to each field and disabling the submit button until the fields are filled out.

1. Make the following changes to the HTML template :

|  |
| --- |
| **<input type="text" name="fName" [ngModel] = "user.firstname" required><br />**  **<label for="lName">Last name:</label>**  **<input type="text" name="lName" [ngModel] = "user.lastname" required><br />**  **<input type="submit" value="Submit"><br />** |

1. Add the normal HTML disabled attribute wrapped as an Angular directive:

|  |
| --- |
| **<input type="text" name="lName" [ngModel] = "user.lastname" required><br />**  **<input type="submit" value="Submit" [disabled]='userDetails.invalid'><br />**  **</form>** |

When the browser refreshes, you will notice that the submit button is disabled. However, as you fill out both fields, the button is enabled again.

1. Something else we can do here is message the user. Add a <div> tag that is visible only if the input element is NOT valid, so when it is empty or if we add custom validation requirements in the future:

|  |
| --- |
| **<label for="fName">First name:</label>**  **<input type="text" name="fName" ngModel = "user.firstname" required><br />**  **<div [hidden]="fName.valid">Please tell us your name</div>**  **<label for="lName">Last name:</label>** |

Here, the fName.valid property is part of the overall form object. In this case the status of valid is **true** if the user did **not** enter anything in the field.

1. The app should now break. The problem is that fName is not valid at the moment. We need to add a local reference variable so that Angular can detect which control we are working with:

|  |
| --- |
| **<label for="fName">First name:</label>**  **<input type="text" name="fName" [ngModel] = "user.firstName" #fName = "ngModel" required>**  **<div [hidden]="fName.valid">Please tell us your name</div><br />**  **<label for="lName">Last name:</label>**  **<input type="text" name="lName" [ngModel] = "user.lastName" #lName = "ngModel" required><br />** |

1. At this point, I will stop for a moment to discuss the various form and control states. See Appendix F for more details on this topic.
2. Now we know that Angular gives us other states that we can leverage such as *pristine* and *touched* in addition to *valid/invalid*. At the moment the message shows even as the page is loaded. If you prefer a behavior where the message is shown **only if** the user interacts with form, then we will need additional criteria to show the message:

|  |
| --- |
| **<label for="fName">First name:</label>**  **<input type="text" name="fName" ngModel = "user.firstname" #fName = "ngModel" required><br />**  **<div [hidden]="fName.valid || fName.pristine">Please tell us your name</div>**  **<label for="lName">Last name:</label>** |

With this addition, the form will load without the message. The message will appear only if the user attempted to enter a value but then erased their value.

Alternatively: **<div [hidden]="fName.valid || (fName.pristine && !fName.touched)">Please tell us your name</div><br />**

1. Another simple feature of Angular is the *reset* feature, see below:

|  |
| --- |
| **<input type="text" name="lName" ngModel = "user.lastname" required><br />**  **<input type="submit" value="Submit" [disabled]='userDetails.invalid'>**  **<input type="submit" value="Reset" (click)='userDetails.reset()'><br />**  **</form>** |

If used the form will delete all the user values and may revert to default ones

**The rest of this part is optional, it deals with styles.**

1. Import the styles into our main styles.css file in the src folder:

|  |
| --- |
| **@import url('https://unpkg.com/bootstrap@5.3.0/dist/css/bootstrap.min.css');** |

Note: In this boot camp we will take a different approach to using Bootstrap. Also the latest version may have changed by the time you are taking this boot camp, but not significantly.

1. Once the bootstrap.min.css file is imported we can start using the CSS classes:

|  |
| --- |
| **<div class="container">**  **<form #userDetails = "ngForm" (ngSubmit)="onSubmit(userDetails.value)" novalidate>**  **<div class="form-group">**  **<label for="fName" class="form-label">First name:</label>**  **<input type="text" name="fName" class="form-control" ngModel = "user.firstname" #fName = "ngModel" required><br />**  **<div [hidden]="fName.valid || fName.pristine">Please tell us your name</div>**  **</div>**  **<div class="form-group">**  **<label for="lName" class="form-label">Last name:</label>**  **<input type="text" name="lName" class="form-control" ngModel = "user.lastname" required><br />**  **</div>**  **<input type="submit" class="btn btn-primary" value="Submit" [disabled]='userDetails.invalid'>**  **<input type="submit" class="btn btn-warning" value="Reset" (click)='userDetails.reset()'><br />**  **</form>**  **</div>** |

This is just one option, remember styles are subjective. Bootstrap *mb* means margin bottom.

1. Adding styles make it possible to highlight errors on the form. For example if we added the alert class to our hidden div, we can get the user to pay attention:

|  |
| --- |
| **<input type="text" name="fName" class="form-control" ngModel = "user.firstname" #fName = "ngModel" required><br />**  **<div [hidden]="fName.valid || fName.pristine" class="alert alert-danger">Please tell us your name</div>**  **</div>**  **<div class="form-group">** |

# part 05 – Adding Email

At the moment, the form has two form controls, two are required and have been given the *required* attribute. In this section we add an email field and email types have an additional attribute called simply *email*. In the background the value that the user types in will be checked for validity. Lets see how to implement this in Angular template forms.

1. Add a property for email in the User class

|  |
| --- |
| **export class User {**  **public firstname?: string;**  **public lastname?: string;**  **public email?: string;**  **}** |

1. Add a label and email fields to the form. Notice that *required* and *email* attributes are already in place

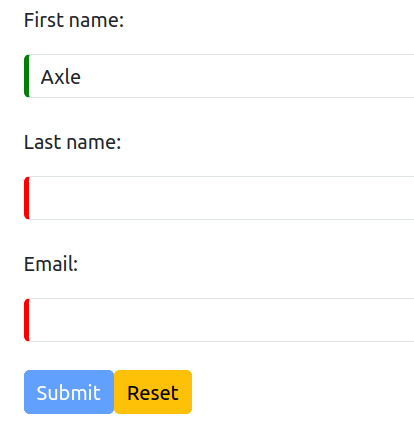
|  |
| --- |
| **<input type="text" name="lName" class="form-control" [ngModel] = "user.lastname" required>**  **</div>**  **<div class="form-group mt-3">**  **<label for="email" class="form-label">Email:</label>**  **<input type="email" name="userMail" class="form-control" [ngModel] = "user.email" required email>**  **</div>**  **<input type="submit" class="btn btn-primary" value="Submit" [disabled]='userDetails.invalid'>** |

1. At this point, if just one of the three fields is invalid, the Submit button will not show. But more importantly, Angular adds class attributes to its elements based on the state of the element and other objects (Part04 #5). Here we see the same form where the first name field is valid and invalid. The images capture both states:

|  |  |
| --- | --- |
|  |  |

1. We can use this information to add even more clarification to the user experience. Add the following classes to the styles.css file in src folder. The name of the class is already provided by Angular, we just have to define details like colour, border background and whatever is required by your company:

|  |
| --- |
| **input.ng-pristine {**  **border-left: 5px solid yellow;**  **}**  **input.ng-touched.ng-invalid {**  **border-left: 5px solid red;**  **}**  **input.ng-touched.ng-valid {**  **border-left: 5px solid green;**  **}** |



1. This is what this change looks like on my Firefox browser:
2. Some fields like email and text will take an additional configuration called *pattern*. With this attribute we can pass in an actual regular expression to test for a valid email. Change the input (email) field to the following:

|  |
| --- |
| **<label for="email" class="form-label">Email:</label>**  **<input**  **type="email"**  **name="userMail"**  **class="form-control"**  **[ngModel] = "user.email"**  **pattern="[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$"**  **required ~~email~~**  **>**  **</div>**  **<input type="submit" class="btn btn-primary" value="Submit" [disabled]='userDetails.invalid'>** |

So now in addition to being *required*, the value the user types in must also pass this *pattern* test. Note, you do not need the *email* attribute if using this pattern.

1. Now that we have an email field, we can demonstrate a different approach to displaying error messages. We will use the Angular structural directive \*ngIf to show/hide an error based on email input. Add the following pair of <div> tags with the two pairs of <span> tags inside:

|  |
| --- |
| **required**  **>**  **<div \*ngIf="(userMail.touched && !userMail.valid)" class="alert alert-danger">**  **<span \*ngIf="userMail.errors.required">Email is a required field!</span>**  **<span \*ngIf="userMail.errors.pattern">Please provide a valid email address!</span>**  **</div>** |

Note: The Bootstrap class has not been added in the above code. Also this approach is different from Part03 #4 where the <div> is hidden or shown based on Angular’s reporting if the field is valid or invalid.

1. These additional lines would probably show an error under userMail. This is because we need to associate this name with ngModel, so change the email code as show below:

|  |
| --- |
| **<input**  **type="email"**  **name="userMail"**  **class="form-control"**  **[ngModel] = "user.email"**  **#userMail="ngModel"**  **pattern="[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$"**  **required**  **>** |

1. Two different approaches and similar results:

|  |
| --- |
|  |

Note: if the code above does not work, you can try this approach:

**<span \*ngIf="userMail.touched && userMail.errors?.['required']">**

**Email is a required field!**

**</span>**

# part 06 – Select and Radio Buttons Form Control

In this section we add a *select* field and a *radio button* group.

1. Add the following code to include a drop-down list box:

|  |
| --- |
| **export class User {**  **public firstname?: string;**  **public lastname?: string;**  **public email?: string;**  **public department?:string;**  **public prizePreference?:string;**  **}** |

We will use department for the select form control and prizePreference for the radio button group later in this section.

1. Add the following code to include a drop-down list box:

|  |
| --- |
| **<div class="form-select mt-5">**  **<label class="form-label" for="department">Department:</label>**  **<select name="department" ngModel = department class="form-control" >**  **<option value="sales">Sales</option>**  **<option value="accounting">Accounting</option>**  **<option value="customerSupport">Customer Support</option>**  **<option value="contentCreation">Content Creation</option>**  **</select></div>** |

1. If you test the form now you will see that it works, however we will be moving forward with a slightly different approach. In the component, add an array of all the departments:

|  |
| --- |
| **export class AppComponent {**  **title = 'skills';**  **user : User;**  **departments : string[] = ["Sales", "Accounting", "CustomerSupport", "ContentCreation"];**  **constructor() {** |

1. Back on the *view*, use the \*ngFor directive to build the select form control. Replace the select code from #3 above with the code from below:

|  |
| --- |
| **<select class="form-control" name="department" [(ngModel)]="user.department" required>**  **<option \*ngFor="let dept of departments"></option>**  **</select>** |

This will get us the basic structure but in order to see the options we need to extract them from the array.

1. The value of each option is held in dept, which can also be used as the display:

|  |
| --- |
| **<select class="form-control" name="department" [(ngModel)]="user.department" required>**  **<option \*ngFor="let dept of departments" [value]="dept">**  **{{dept}}**  **</option>**  **</select>** |

1. Now, we will add a prompt and make its value undefined:

|  |
| --- |
| **<select class="form-control" name="department" [(ngModel)]="user.department" required>  <option value="undefined">--Choose an Option--</option>**  **<option \*ngFor="let dept of departments" [value]="dept">**  **{{dept}}**  **</option>**  **</select>** |

1. Radio buttons can be added in a similar way. Create an array of options in the component:

|  |
| --- |
| **user : User;**  **departments : string[] = ["Sales", "Accounting", "CustomerSupport", "ContentCreation"];**  **prizes : string[] = ["Cash", "Voucher", "Lunch"];**  **constructor() {**  **this.user = new User();** |

We code the *options* that the competition winner may choose as a prize.

1. Add the following code to build the radio button group:

|  |
| --- |
| **<div class="form-group">**  **<label class="form-label" for="prizes">Prize Preference:</label>**  **<div class="radio" \*ngFor="let prize of prizes">**  **<label>**  **<input**  **type="radio"**  **name="prizePreference"**  **[value]="prize"**  **required>**  **{{ prize }}**  **</label>**  **</div>**  **</div>** |

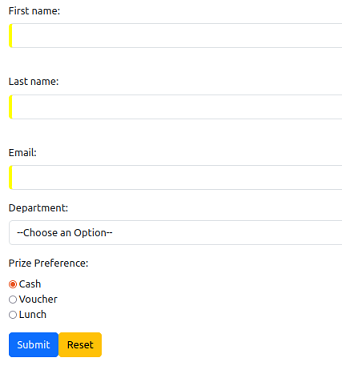
1. If you wanted to have one of the options selected by default, we would have to go back to the component and add two extra lines of code:

|  |
| --- |
| **departments : string[] = ["Sales", "Accounting", "CustomerSupport", "ContentCreation"];**  **prizes : string[] = ["Cash", "Voucher", "Lunch"];**  **selectedPrize : string = "";**  **//**  **constructor() {**  **this.user = new User();**  **this.selectedPrize = this.prizes[0];**  **};** |

The above can be done in one line but it creates a property that we can bind to in the view.

1. Bind to the selectedPrize property:

|  |
| --- |
| **<div class="form-group">**  **<label class="form-label" for="prizes">Prize Preference:</label>**  **<div class="radio" \*ngFor="let prize of prizes">**  **<label>**  **<input**  **type="radio"**  **name="prizePreference"**  **[value]="prize"**  **[(ngModel)]="selectedPrize"**  **required>**  **{{ prize }}**  **</label>**  **</div>**  **</div>** |



# part 07 – Additional Template Form Features

1. We can access a template form via the @ViewChild feature of Angular. Continue using the code from Part06. Change the *form* tag to the following:

|  |
| --- |
| **<div class="container">**  **<form #userDetails = "ngForm" (ngSubmit)="onSubmit(~~userDetails~~)" novalidate>**  **<div class="form-group">**  **<label for="fName" class="form-label">First name:</label>** |

With @ViewChild, we do not pass the form on submit, we access it via the component code using its template reference. Remember to import ViewChild from @angular/core.

1. Add the following code to access our form via its handle, userDetails. It is still of the type NgForm:

|  |
| --- |
| **export class AppComponent {**  **@ViewChild('userDetails') userDetailsForm!: NgForm**  **//**  **title = 'skills';**  **user : User;** |

1. If you wanted to have a look at what this produces:

|  |
| --- |
| **onSubmit(){**  **console.log(this.userDetailsForm.value);**  **}** |

Note: remember if you want to see this code work, you will need to remove the parameter from the HTML template, the <form> tag, so pass nothing. Also remove the incoming parameter from the onSubmit() method.

1. It is also possible to group certain form controls that make sense in your project. For example, in this project we may want to group first and last names into a **full name group**. This gives you access to all the validation properties of the group. Here we add a full name group:

|  |
| --- |
| **<form #userDetails = "ngForm" (ngSubmit)="onSubmit()" novalidate>**  **<div ngModelGroup="fullName">**  **<div class="form-group">**  **<label for="fName" class="form-label">First name:</label>**  **<input type="text" name="fName" class="form-control" [ngModel] = "user.firstname" #fName = "ngModel" required>**  **<div [hidden]="fName.valid || fName.pristine" class="alert alert-danger">Please tell us your name</div>**  **</div>**  **<div class="form-group">**  **<label for="lName" class="form-label">Last name:</label>**  **<input type="text" name="lName" class="form-control" [ngModel] = "user.lastname" required>**  **</div>**  **</div>**  **<div class="form-group">**  **<label for="email" class="form-label">Email:</label>** |

1. If you now look at the console window, you will see that new group being shown based on the name of the ngModelGroup. In our case it is fullName:

A screenshot of a computer

Description automatically generated

1. Angular offers two methods of initializing form fields, setValue() and patchValue(). The setValue() method is easy to code so in this section we will focus on patchValue(). We will use the users value for firstName and lastName and create a suggested email address. With patchValue() we will initialize an email address and allow the user to accept it or change it. On the template add parenthesis to ngModel, around both firstName and lastName fields:

|  |
| --- |
| **<label for="fName" class="form-label">First name:</label>**  **<input type="text" name="fName" class="form-control" [(ngModel)] = "user.firstname" #fName = "ngModel" required>**  **<div [hidden]="fName.valid || fName.pristine" class="alert alert-danger">Please tell us your name</div>**  **</div>**  **<div class="form-group">**  **<label for="lName" class="form-label">Last name:</label>**  **<input type="text" name="lName" class="form-control" [(ngModel)] = "user.lastname" required (blur)="suggestEmail($event)">**  **</div>** |

The setValue() method is used to initialize ALL the fields in a form and is coded exactly as the patchValue()method.

1. Once the user fills out both first and last name fields, on exiting the lastName field we can activate the *blur* event and call a method on the component:

|  |
| --- |
| **<label for="fName" class="form-label">First name:</label>**  **<input type="text" name="fName" class="form-control" [(ngModel)] = "user.firstname" #fName = "ngModel" required>**  **<div [hidden]="fName.valid || fName.pristine" class="alert alert-danger">Please tell us your name</div>**  **</div>**  **<div class="form-group">**  **<label for="lName" class="form-label">Last name:</label>**  **<input type="text" name="lName" class="form-control" [(ngModel)] = "user.lastname" required (blur)="suggestEmail($event)">**  **</div>** |

Also remove the *pattern restriction* on the email field. Other validation signals can remain for now.

1. On the component side, all we need to do is code the suggestEmail() method. For this method, we call the patchValue() method of userDetails.form property. Also we need to pass the suggested value as an object:

|  |
| --- |
| **suggestEmail(event: any){**  **this.userDetailsForm.form.patchValue(**  **{userMail : this.user.firstname + "." + this.user.lastname + "@skillsoft.com"}**  **);**  **}** |

Remember that patchValue takes an object

# Appendix A – Angular Architectural Concepts

Angular uses the concept of modules (Ng Modules) into which components are placed. There are built-in modules that come with the installation of Angular. Some of these modules we will be using in the course include the HttpClientModule and the FormsModule. An Ng Module is just a TypeScript class with an @NgModule decorator. Most decorators add metadata to the class and in come cases functionality. By default we get the AppModule to help us kickstart our customized development.

Decorators may contain declarations, exports, imports, providers and bootstrap classes. Declarations handle views like component views and directive views. Export classes ensure that a class can be accessed by other classes. Imports exposes modules required by a class. Providers handle Services which are mostly logic required by some class. Bootstrap is in the root component and provides the initial view.

There are several JS modules used as libraries in an Agular application. Libraries such as @angular/core, @angular/router and Material are used to add functionality. These libraries are simply imported.

Components comprise of a TypeScript class, some kind of HTML template for display and a stylesheet. A component will have the @Component decorator to define it as a component.

A customized component will usually have a selector which is an instructor to Angular to insert this particular component where ever it finds the selector. The selector tag within the HTML is usually written as <app-root></app-root>.

The templateUrl will point to an html file which acts as the template for a component. styleUrls of course does the same for CSS files.

Directives:

Directives are instructions that instruct the DOM as to how to place your components and business logic in the Angular project. Directives are just JS class which are declared as @directive. There are 3 directives in Angular: Component Directives, Structural Directives and Attribute Directives.

Component Directives look like this @Component. They contain the detail of how the component should be processed, instantiated and used at runtime.

Structural directives start with a \* sign. These directives are used to manipulate and change the structure of the DOM elements. For example, \*ngIf and \*ngFor.

Attribute directives are used to change the look and behavior of the DOM elements. For example: ngClass, ngStyle etc.

The main building blocks of Angular are:

* Modules
* Components
* Templates
* Services
* Metadata
* Directives
* Data binding
* Dependency injection

Here are a few Angular CLI commands that we will be using

|  |  |  |
| --- | --- | --- |
| *add* |  | Used to add support for an external library to your project. |
| *build* | Will compile an Angular app into an output directory named dist/ at the given output path. |
| *generate* | Generates and possibly modifie files based on a schematic. |
| *new* | Creates a new workspace and a boilerplate Angular app. |
| *run* | Runs an Architect target |
| *serve* | Builds and serves your app via http, also re-compiles when it detects changes. |
| *test* | Executes unit tests in a project |
| *update* | Updates your application and its dependencies |

Angular 14 File Explanation

* src folder: all the action takes place here
* app folder: all the files, that support app components.
* app.component.css: the cascading style sheets code for your app component.
* app.component.html: the template html file connected to app component and is used by angular to do any data binding.
* app.component.spec.ts: use the command ng test to see this file in action. It is a unit testing file related to app component. All files that have .spec in the middle is a test file
* app.component.ts: probably the most important typescript file which contains the view logic driving the component.
* app.module.ts: a file which includes all the dependencies for the entire website. This file defines any modules to be imported, components to be declared and the main component to start the app
* karma.config.js: This file specifies the config file for the Karma Test Runner, Karma has been developed by the AngularJS team which can run tests for both AngularJS and Angular 2+
* main.ts: As defined in angular.json file, this is the main ts file that will first run. This file bootstraps (starts) the AppModule from app.module.ts , and it can be used to define global configurations.
* polyfills.ts: This file is a set of code that can be used to provide compatibility support for older browsers. Angular 7 code is written mainly in ES6+ language specifications which is getting more adopted in front-end development, so since not all browsers support the full ES6+ specifications, pollyfills can be used to cover whatever feature missing from a given browser.
* styles.css:/ This is a global css file which is used by the angular application.
* tests.ts: This is the main test file that the Angular CLI command ng test will use to traverse all the unit tests within the application and run them.
* tsconfig.json: This is a typescript compiler configuration file.
* tsconfig.app.json: This is used to override the tsconfig.json file with app specific configurations.

tsconfig.spec.json: This overrides the tsconfig.json file with app specific unit test configurations

# Appendix B – Angular Directives

Directives are functions used reinforce HTML, make it do much more than what it was designed for. These directives have names like \**ngFor* and *ngStyles* but can be any name you make up and they are specific to an HTML element, an attribute or class

DOM manipulation directives are called attribute or structural directives.

Attribute directives manipulate the DOM by changing its behavior and appearance.

Using the Existing Angular Directives in an example:

|  |
| --- |
| **<div [**[**ngStyle**](https://angular.io/api/common/NgStyle)**]="myStyles">**  **Content goes here**  **</div>** |

You can now define myStyles somewhere in your .ts file as a function.

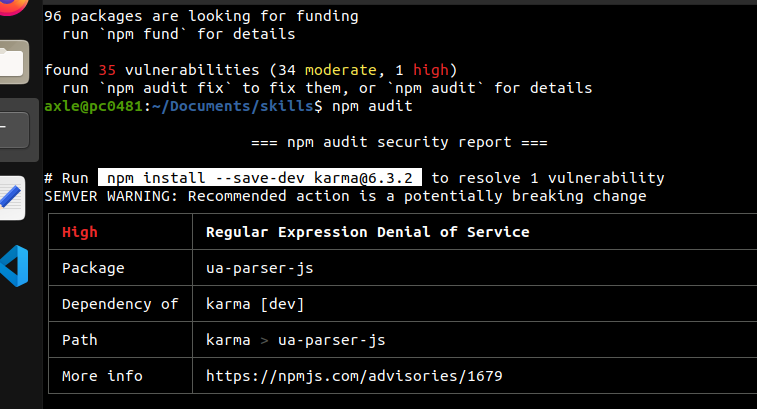
Structural directives are meant to create and destroy DOM elements and usually start with the \* character such as *\*ngIf*

|  |
| --- |
| **<div \*ngIf="condition">Content to render when condition is true.</div>** |

Components are also special directives

# Appendix C – Installation Issues

If you get issues while installing json-server run npm audit to see what might be stopping the installation and how you might be able to fix it:



So in this case, I installed [karma@6.3.2](mailto:karma@6.3.2)

# Appendix D – @NgModule

**Declarations** are used to declare components, directives, pipes that belong to the current module. Think of a namespace, declarations create a namespace so all the components in this @NgModule are available to each other in a public but protected way.

**Imports** (and exports) work just like in other programming languages. They are used to import supporting modules like FormsModule, RouterModule and the CommonModule.

**Providers** are used by modules for accessing the services required by components, directives. The process is known as injecting services into the component.

The **bootstrap** property simply points to a component that will be used to start the application.

# Appendix E – Form States

Pristine - The user did not interact the form control

Dirty - The user interacted with the form control somehow

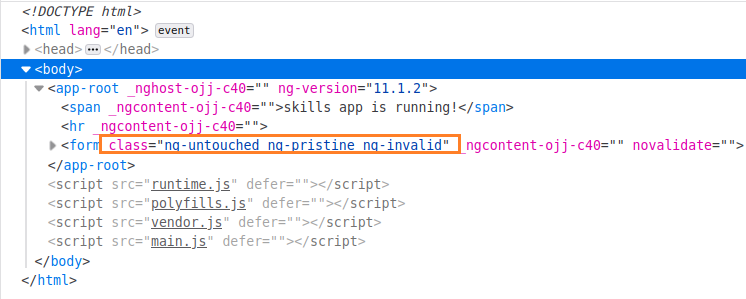
Touched - The user has done something with the form control

Untouched - The form control has not been interacted with by the user.

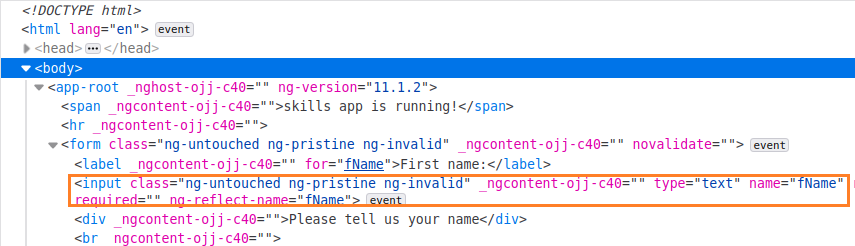
Valid - The users value is in accordance with validation rules defined in the application.

Invalid - The users value does not meet the validation rules defined in the TS code

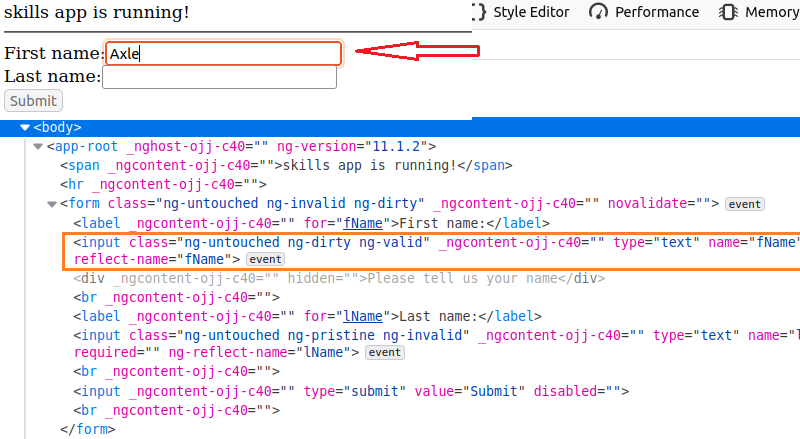
In the image below, this is when the form first appeared in the browser window. It is un-touched and pristine. It is also invalid until the form fields are filled in by the user:



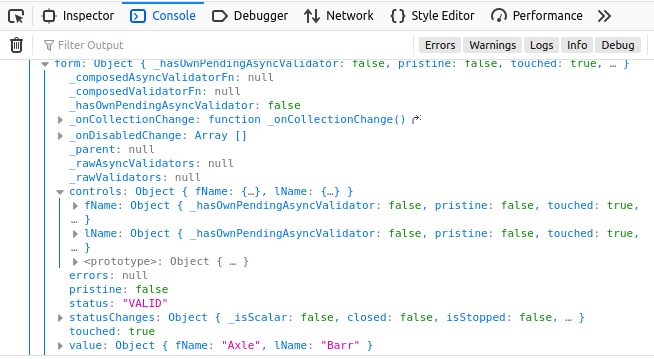
The form control itself gives the details similar to above:



Below, we see that the first name field is filled in, so that control is now untouched, dirty but valid:



Another method to view form and control status is via the console window itself:



If you wanted to see this then in the template you must pass the form itself:  
  
**<form #userDetails = "ngForm" (ngSubmit)="onSubmit(userDetails)" novalidate>**

Then in the onSubmit() method, accept as a parameter the form as type NgForm. Then you can simply print the parameter:

**onSubmit(userDetails : NgForm){**

**console.log(userDetails);**

**}**