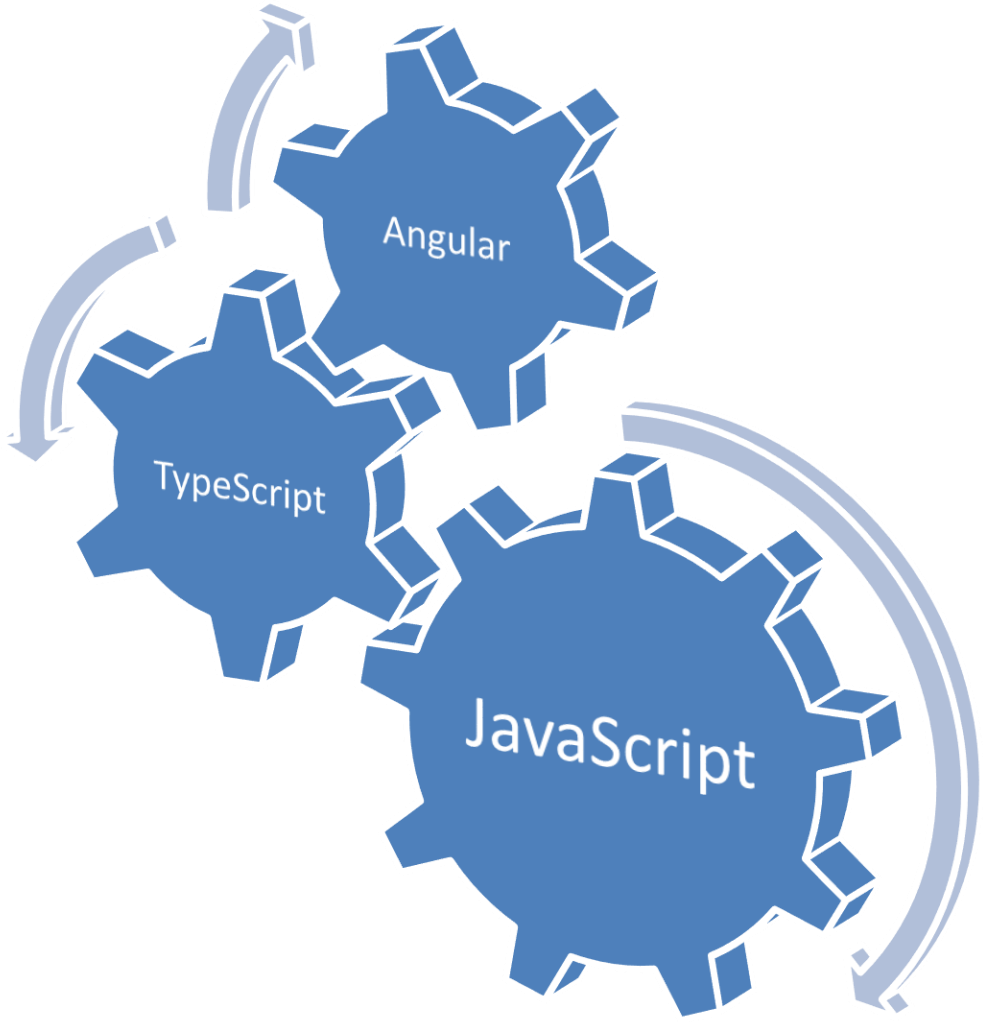
WD-530 Angular for TypeScript exercises 2

Client-side application building

Centriq Training



530 Angular exercises v3.0.docx
last saved March 13, 2020

Table of Contents

Angular for TypeScript

Events

24. Click event

- Let's go back to the dog-panel exercise and create a new template to use for it. Copy the **dog-panel \ app.component2.html** to a new file and rename it to **dog-panel\app.component4.html**.
- Update the bootstrap module:

/ main.ts

```
import { AppModule } from './dog-panel/app.module';
```

- Then, update the template in your DogPanel component to

dog-panel / app.component.ts

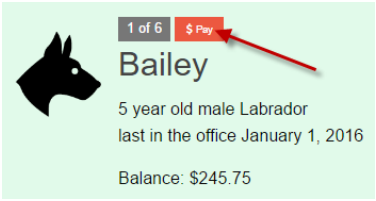
```
    selector: 'app-root',
    templateUrl: app.component4.html',
```

- View in the browser and the component should still look the same as the last time. But update your template to show a change:

dog-panel / app.component4.html

```
 <h1>Dog Panel v4</h1>
```

Pay the bill



- We'll let the label of our balance notification be our button so that when we click on it, it will set the balance to zero.
- We first bind a click event to a payBill function that we will write in the Dog Panel component and put it in the balance notification label. We'll also switch the label to a tiny button to get that UI look.
- Update your code to the label in **app.component4.html** like this:

dog-panel / app.component4.html

```
<span *ngIf ="dog.balance > 200" class='tiny alert button'
(click)='payBill($event)'><i class="fas fa-dollar-sign"></i> Pay</span>
```

- Refresh your browser and you can see that the bottom alignment is off because of extra margin on the button that's not on the label. We'll fix this in the stylesheet like this

dog-panel / app.component.css

```
}
.tiny.button {
    margin:0;
    padding: .6em 1em;
}
```

- Now checking the browser again, it should line up better.

- If you click the button now, you'll see a wealth of red messages in the Console of your Dev Tools complaining that payBill( ) is not a function. That's so true. Let's add a stub in the Dog Panel component.

dog-panel / app.component.ts

```
    get dog() { return this._dog; }
    public payBill(param) {}
```

- You should be able to click on the button and get no red complaints now. Test it.
- To see what we're working with a little more, let's add some information in the console for debugging

dog-panel / app.component.ts

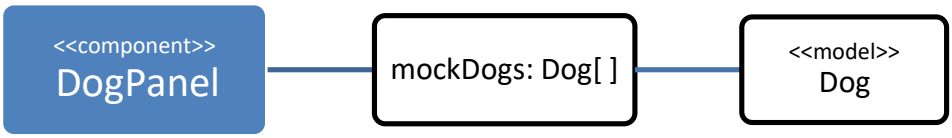
```
        public payBill(event: MouseEvent) {
            console.info('payBill() received event', event );
            var dogIndex: number =
event.target.parentElement.querySelector('#dogIndex').innerText;
            console.info('scraped Dog index = ', dogIndex);
            this.dog.balance = 0;
            console.info(this.dog);
        }
```

- The dog model requires a set method for the balance so we can set it to zero. Check to make sure this code is in your class:

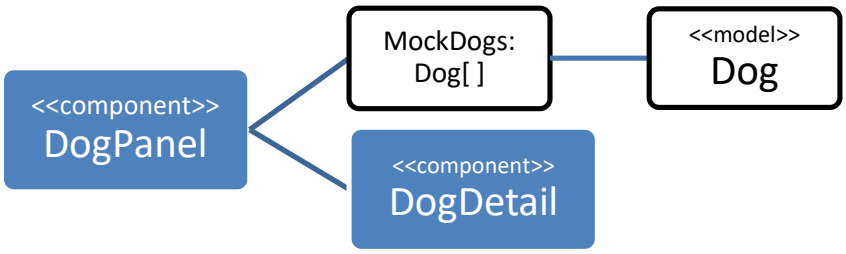
dog-panel/model/dog.ts

```
        get balance(): number { return this._balance; }
        set balance(inBalance: number){this._balance = inBalance;}
```

- We now have a problem. The dog panel component is keeping track of a dog made in the constructor which it does nothing with. It also has the fake data for which the template is displaying. Our component should really be handling the layout only and send a dog at a time for another child component that can remember which one it's dealing with. That will prevent us from doing the scraping that would have been necessary in jQuery or pure JavaScript like you see in the payBill( ) function unless you did some fancier coding.
- The solution is to break out the dog display component on its own. The component diagram currently looks like the one below.



- But there is no memory of an individual dog when the DogPanel pay button is clicked to tell the dog from the MockDogs that the balance is zero. That reason is important because the MockDogs represents our database that should be updated. Our panel would have to map DOM dogs to the MockDogs by some key.
- We will have to alter the component diagram to the one below and pass the data.



Refactor the DogPanel

- Start by creating the empty files for **dog-panel/app.component.ts** and **dog-panel /app.component.html.**
- Then copy what you can over to the files and update them so that they look like the following. This is the bare bones version with no comments or modal stuff.

dog-panel / dog-detail.component.ts

```
    import { Component, Input } from '@angular/core';
    import {Dog}                from './model/dog';

    @Component({
        selector: 'dog-detail',
        templateUrl: './dog-detail.component.html',
        styles: [
            '.c-imagePlacement1 {margin:1em 1em 3em 0; max-height:5em}',
            '.c-imagePlacement2 {margin:.5em .5em 1.5em 0; max-
height:2.5em}',
            '.tiny.button { margin:0; padding: .6em 1em;}'
        ]
    })
    export class DogDetail {
        @Input('dogIn') _dog: Dog;

        get dog() { return this._dog; }
        set dog(dogIn: Dog) {
            console.info('Updating from', this.dog);
            this._dog = dogIn;
            console.info('Updated to', dogIn);
        }
        public payBill(event: MouseEvent) {
            console.info('PayBill received event', event);
            this.dog.balance = 0;
            console.info('Paid bill for', this.dog);
        }
    }
```

- and

dog-panel / dog-detail.component.html

```
        <span *ngIf ="dog.isSenior" class='warning label'>Senior</span>
        <span *ngIf ="dog.balance > 200" class='tiny alert button'
(click)='payBill($event)'><i class="fas fa-dollar-sign"></i> Pay</span>
        <span *ngSwitch = "dog.examType">
            <template *ngSwitchCase = "'checkup'"><span class='label'><i
class="fas fa-check"></i></span></template>
            <template *ngSwitchCase = "'routine'"><span class='label'><i
class="fas fa-stethoscope"></i></span></template>
            <template *ngSwitchCase = "'major'"><span
class='label'>Major</span></template>
            <template *ngSwitchCase = "'complete'"><span
class='label'>Complete</span></template>
            <template *ngSwitchDefault></template>
        </span>
        <img src="{{dog.imageUrl}}" title="{{dog.name}}" alt="{{dog.name}}"
class='float-left c-imagePlacement1'>
        <h3>{{dog.name}}</h3>
        <div style='font-size:95%'>{{dog.age}} year old {{dog.gender}}
{{dog.breed || 'unknown breed'}}
        last in the office {{(dog.lastOfficeVisit | date:'longDate') ||
'never'}}
        <span #balance *ngIf = "dog.balance > 0">Balance: {{dog.balance  |
currency:'USD':'symbol':'1.2-2' }}</span></div>
```

- This leaves the dog-panel files much leaner. Here are those new versions:

dog-panel / app.component.ts

```
    export class AppComponent {
        mockDogs: Dog[] = MockDogs.SIX;
    }
```

- and

dog-panel / dog-panel.component4.html

```
    <div class='row'>
    <div *ngFor="let dog of mockDogs; let i = index" class="success callout
small-4 columns ">
        <span class='secondary label'><span>{{i+1}} of {{mockDogs.length}}
dogs</span>
        <dog-detail [dogIn]='dog'></dog-detail>
    </div>
    </div>
```

- The last step is to update the declarations with the new component which requires changes to two lines:

dog-panel / app.module.ts

```
    import {DogDetail} from './dog-detail.component';
```

```
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ DogPanel, CurlyQuotesPipe, DogDetail ],
```

- Build and view in your browser. Click on the pay button to pay the bill. Read the console messages.

25. Click event talking to parent

- The child panel is paying the bills, but we'd like to show a status message for feedback on the parent template which means we have to tell them so they know what to display.
- We'll add a status panel to the Dog panel when there is a message from the Dog detail component. The status message is just a div above the panel like this

dog-panel \ app.component4.html.

```
   <div id='status' [ngClass]="{callout: panelStatus, alert :
panelStatus}">
        {{panelStatus}}
   </div>
   <div class='row'>
```

- The property of panelStatus must be added now to the component in the AppComponent class.

dog-panel \ app.component.ts

```
   export class AppComponent{
       mockDogs: Dog[] = MockDogs.SIX;
       panelStatus : string;
```

- The other thing we want to do is listen for some messages coming from the child component of the details so we'll add a method to capture all the info we want from there like this:

dog-panel \ app.component.ts

```
   panelStatus : string;

   onMessageFromDetail(event: any[]) {
     var message: string = event[0];
     var dogActedOn: Dog = event[1];
     var paidAmount: number = event[2];
     console.info("Received message", event[0], event[1]);
     this.panelStatus =
       message + " of $" + paidAmount + " for " + dogActedOn.name;
   }
```

- Look for the console messages when you are using the panel.
- Now we need to open the door to the messages coming from the child component through the element selector that holds that information. That's in the template file and it gets a small update here:

dog-panel \ app.component4.html

```
   <dog-detail [dogIn]='dog'
(messageFromDetail)='onMessageFromDetail($event)' ></dog-detail>
```

- It's always fun to see if you get any error messages and get used to knowing that you may not get any information back at all or a blank template. Check often to see how things look on the browser.
- The last thing to get updated is the DogDetail class in the **dog-detail.component.ts** file. There's one @Output decorator to declare and the message to send. Here's the outputting EventEmitter:

dog-panel \ dog-detail.component.ts

```
export class DogDetail{
    @Input('dogIn') _dog: Dog;
    @Output() messageFromDetail: EventEmitter<any> = new EventEmitter();
```

- The EventEmitter requires an import at the top of the file also:

dog-panel \ dog-detail.component.ts

```
import {Component, Input, Output, EventEmitter}   from '@angular/core';
```

- And the payBill( ) function gets a few more lines to use the EventEmitter to pass an array of values to the parent:

dog-panel \ dog-detail.component.ts

```
    let paidAmount = this.dog.balance;
    this.messageFromDetail.emit(['Paid bill', this.dog, paidAmount]);
    this.dog.balance = 0;
    console.info('Paid bill for', this.dog);
}
```

- Now we've completed the chain of:
  user clicks element in **child** →
  (click) listener in DogDetail **child** element template picks up event →
  event handler in the DogDetail **child** component class is executed →
  event is emitted/published from DogDetail **child** class →
  event subscribed to in DogPanel **parent** element template picks up propagated event →
  and event handler executed in DogPanel **parent** component .

26. Accordion

Copy and update template

- Copy the contents of your **template** directory in VS Code and paste to create copies of all the files. Rename the new **template copy** directory to **accordion**.
- Update the bootstrap module:

/ main.ts

```
import { AppModule } from './accordion/app.module';
```

- Update the component template:

accordion / app.component.html

```
<h1>Accordion</h1>
```

- Save the files and view the page in your browser. Restart the start task if necessary. You should see an Accordion headline.

Add data and extra template

- The accordion app is going to need some data for each section of the accordion. That data could use a structure to be more reusable and manageable.
- Create a new file for the model in a new **model** directory called **accordionSection.ts** and put this class structure in there:

accordion / model / accordionSection.ts

```
export class AccordionSection {
    private _hidden: boolean;
    private _label: string;
    private _content: string;

    constructor(labelIn?: string, contentIn?: string) {
      this._label = labelIn || 'no label';
      this._content = contentIn || 'no content';
            this.hidden = true;
 }
    get label() { return this._label; }
    set label(value: string) { this._label = value; }
    get content() { return this._content; }
    set content(value: string) { this._content = value; }
    get isHidden() {return this._hidden};
    set hidden(value: boolean) {this._hidden = value; }
}
```

The constructor uses optional parameters and can be used without any arguments. The assignment of the arguments to the object properties includes a default operator (the boolean OR) to assign default values if none are passed in.

- The data that can be loaded from an AJAX call or any other source is going to come from a separate file in the model directory also, called **accordionData.ts**. It's just a convenience for testing. Here's the contents:

accordion / model / accordionData.ts

```
export class AccordionData {
    static FOUR: Array<any> = [
        {
            label: 'Definition',
            content: 'Akkordion is German for ‘harmony’.'
        },
        {
            label: 'Invention',
            content: 'An accordion-like instrument called the Handäoline
was invented by Friedrich Buschmann in Berlin, Germany, in 1822.'
        },
        {
            label: 'Official City',
            content: 'The accordion is the designated instrument for the
city of Detroit.'
        },
        {
            label: 'Patent',
            content: 'Cyril Demian patented the first actual accordion
in 1829 in Vienna, Austria under the name Akkordion.'
        }
    ];
}
```

- We access the data and model classes from the **accordion.component.ts** component class first with two imports at the top of the page

accordion / app.component.ts

```
import {Component}        from '@angular/core';
import {AccordionData}  from './model/accordionData';
import {AccordionSection} from './model/accordionSection';
```

- Our dog data was initialized from Dog objects to begin with, but data more often comes in JSON format and is accessed directly or for better functionality, mapped and parsed into objects.
- We will convert the accordion data from the array of anonymous objects into an array of AccordionSection objects in the constructor of the Accordion class. Also, there's a confirmation that the data was loaded after the component is initialized that you will see in the browser's console.

accordion / app.component.ts

```
export class AppComponent{
    sections: AccordionSection[] = [];
    constructor() {
```

```
            AccordionData.FOUR.forEach((element, index, array) => {
                const section =
                new AccordionSection(element.label, element.content);
                this.sections.push(section);
            });
            console.info('constructor()');
        }
        ngOnInit() {
            console.table(this.sections);
        }
}
```

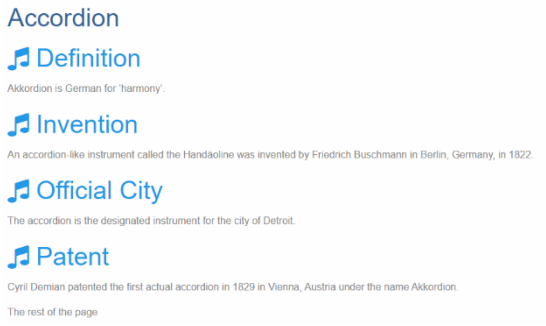
Console.table is a great display function and Chrome knows it exists just fine. IE/Edge knows it hasn't implemented it and throws a warning to use an "alternative mechanism" which to me is a euphemism for Chrome or gives some other confusing message.

- The last part to do is to output the data in a template. The file you need to update is **app.component.html** which will be the view workhorse for the accordion. Here's the code:

accordion / app.component.html

```
<h1>Accordion</h1>
<section *ngFor="let section of sections; let i=index" >
    <h2>
    <a><i class="fa fa-music"></i> {{section.label}}</a>
    </h2>
    <p>
      {{section.content}}
 </p>
</section>
```

- Refresh your browser and you should see the data in a structure that will look like a fully expanded accordion. Check the console out for the raw data.



Hide and view on click

- A click on the accordion section label will not do anything. We want that click to open the closed version of the accordion section.
- Set up the default of each section to have the content closed when first loaded. Add a class to do this in the component stylesheet **app.component.css**:

accordion / app.component.css

```
/* component specific stylesheet */
    section p.hide {
        display: none;
    }
```

- Then we add that class to each one of the sections based on the section's property that's keeping track of the state of its visibility via Angular's special directive in the template:

accordion / app.component.html

```
        <p [ngClass] = "{hide: sections[i].isHidden}">
            {{section.content}}
        </p>
```

- You can check the progress of the code in the browser if you want now. The accordion should be completely collapsed and not respond to clicks.
- To change the state of the visibility of any clicked section, we will pass the id of that section to the component, turn off the visibility of all the others, and then turn on the visibility of the clicked one. Add an event listener for a click on the link element of the section in the label and also, rotate the music icon if the content is visible:

accordion / app.component.html

```
    <section *ngFor="let section of sections; let i=index" >
        <h2><a (click)='accordionClick($event)' id='{{i}}'>
       <i class="fa fa-music {{section.isHidden ?  '' : 'fa-rotate-270' }}
"></i> {{section.label}}</a>
```

Why wouldn't you add the click event handler on the H2 element and then simplify the template by eliminating the link? It's a design issue. What about the design needs to change if you did that? How do you make the user aware that the label is clickable?

- It throws an error in the console because we still need to handle the click. Create the event handler for that in **app.component.ts**

accordion / app.component.ts

```
        console.table(this.sections);
    }
    accordionClick(event: MouseEvent) {
            event.preventDefault();
            const id = event.srcElement['id'];
            console.log('Clicked accordion element', id);
            console.dir(event.srcElement); // to see the properties
            //close all
            this.sections.forEach((section) => {section.hidden = true;})
            const clickedContent = this.sections[id];
            clickedContent.hidden = !clickedContent.isHidden;}
```

- Build the project and view in the browser to test the code.

Trick out with events

- We will add four event bindings and tie them to some new event handler functions. The first will just show us an event in the console when you right-click the mouse on the section. Add the next few code blocks to your **app.component.html** file.

accordion / app.component.html

```
    <section *ngFor="let section of sections; let i=index"
(contextmenu)='showEventInConsole($event)'>
        <h2><a (click)='accordionClick($event)' id='{{i}}'>
```

- The other three will use buttons at the top of the accordion

accordion / app.component.html

```
    <button class="tiny button" (click)='reverseSections()'>
        Click to reverse
    </button>
    <button class="tiny button" (dblclick)='deleteLastSection()'>
        Double-click to delete last section
    </button>
    <button class="tiny button" style = 'box-shadow: 0px 0px 30px hsl(200,
100%, 50%);' (mouseover)='shuffleSections()'>
        Hover to scramble
    </button>
    <section *ngFor="let section of sections; let i=index"
```

- The functions to handle the events go at the bottom of the Accordion component class in the **app.component.ts** file**.**

accordion / app.component.ts

```
            clickedContent.hidden = !clickedContent.isHidden;
        }
        showEventInConsole(event: Event){
            console.info(event);
            event.preventDefault();
        }
        reverseSections() {
            this.sections.reverse();
        }
        static shuffleArray(array : any[]) {
            for (let i = array.length - 1; i > 0; i--) {
                const j = Math.floor(Math.random() * (i + 1));
                const temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
            return array;
        }
        shuffleSections() {
            AppComponent.shuffleArray(this.sections);
        }
        deleteLastSection() {
            this.sections.pop();
        }
```

- You will get a lightbulb message to fix the placement of the static method so that it appears before the other non-static methods. Accept the fix under the lightbulb.
- Now make sure all your work is saved and refresh your browser to test out the four different events.

Forms

27. Form app setup

Copy and update template

- Copy the contents of your **template** directory in VS Code and paste to create copies of all the files. Rename the new **template copy** directory to **form**.
- Update the bootstrap module:

/ main.ts

```
import { AppModule } from './form/app.module';
```

- Update the component template:

form / app.component.html

```
<h1>Form</h1>
```

- Save the files and view the page in your browser. Restart the start task if necessary. You should see a Form headline.

Add child component and template

- We will add a component that will handle the name and address of a person and use several different templates to investigate the ways that it can be managed.
- Update your template with the element selector first:

form / app.component.html

```
<h1>Form</h1>
<name-address-form></name-address-form>
```

- Update your module file to bring in the future component and declare it to the module:

form / app.module.ts

```
import { AppComponent }  from './app.component';
import {NameAddressForm} from './name-address.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, NameAddressForm ],
```

- Then, we create the child component class. Here's the file

form / name-address.component.ts

```
import { Component } from '@angular/core';
@Component({
        selector: 'name-address-form',
    styleUrls: ['name-address.component.css'],
    templateUrl : `name-address.component.0.html`
})
```

```
export class NameAddressForm  {
}
```

- Create an empty file for the **form/name-address.component.css** for the future and so it will not fail in the build.
- Create an empty file for the template also. Copy the contents from the site for the project code. It has input status indicators, error message placeholders, buttons, and input fields on it. The code looks like:

form/name-address.component.0.html

```
<form>
<div>
    <b>Preview for model data</b>: <br>
    <b>Preview for form data</b>: <br>
</div>

<h3>0 Base form</h3>
<button class='tiny button'>Show form</button>
<fieldset class="fieldset">
    <legend>Name and address form
        <span class ='label secondary' >Unchanged</span>
        <span class ='label warning' >In progress</span>
        <span class ='label success'>Ready to submit</span>
    </legend>

<div name="name" class="row">
    <div class="large-6 columns">
        <label for="first">First Name:
            <span class ='label alert' >err</span>
        </label>
        <input id='first' type="text" >

    </div>
    <div class="large-6 columns">
        <label for="last">Last Name:
            <span class ='label alert' >err</span>
        </label>
        <input id='last' type="text" >
    </div>
</div>

<div>
    <div name="street" class="row">
        <div class="large-8 columns">
            <label for="street">Street:
                <span class ='label alert' >err</span>
            </label>
            <input id='street' type="text" >
```

```
            </div>
        </div>

        <div name="city-state-zip" class="row">
            <div class="large-4 columns">
                <label for="city">City:
                    <span class ='label alert'>err</span>
                </label>
                <input id='city' type="text">
            </div>
            <div class="large-1 columns">
                <label for="state">St:<span
                    class ='label alert' >err</span>
                </label>
                <input id='state' type="text">
            </div>
            <div class="large-3 columns">
                <label for="zip">Zip:
                    <span class ='label alert' >err</span>
                </label>
                <input id='zip' type="text">
            </div>
        </div>
    </div>

    </fieldset>
    <button type="submit" class="button" >Submit when valid</button>
    </form>
```

- The label's for attribute and the input's id must match to be HTML5 compliant. The local variables prefixed with # do not have to be named the same but it helps give order to all the names being used.

- Build the project and view in your browser. You should have a useless form that will not submit any data but will refresh. And it looks good.

28. Submitting with local variables

Copy and submit form

- We'll keep a copy of this basic html form and use it later. Make a copy of **form/name-address.component.0.html** and call it **form/name-address.component.1.html.**
- Update your templateUrl to use the new file

form / name-address.component.ts

```
    styleUrls: ['name-address.component.css'],
    templateUrl : `name-address.component.1.html`
```

- Update the h3 element in the new template file to this:

form / name-address.component.1.html

```
<h3>1 Scrape and send data</h3>
<button class="tiny button">Show form</button>
```

- We first add some local variables to the fields in this template file so they are easier to access

form / name-address.component.1.html

```
        <input id='first' type="text" #nameFirst >
    </div>
    <div class="large-6 columns">
        <label for="last">Last Name:
            <span class ='label alert' >err</span>
        </label>
        <input id='last' type="text" #nameLast >
```

- Then we'll add an event listener in the form for a submit button that will scrape the data from JavaScript properties and call the component function submittingForm1. Also, the button text doesn't make sense so we'll update it a little.

form / name-address.component.1.html

```
<button type="submit" class="button" (click)="
        data = {
            nameFirst: nameFirst.value,
            nameLast: nameLast.value,
            nameFirstId: nameFirst.id,
            nameFirstType: nameFirst.type,
            nameFirstOffsetHeight: nameFirst.offsetHeight,
            nameFirstParentNode: nameFirst.parentNode.nodeName
        };
        submittingForm1($event, data);
        "> Submit when valid </button>
```

- The click event is usually assigned one statement but you can use a multi-line script. It is in context of the component not the window, so jQuery and document context cannot be used here. Also the parser does not allow var to be used. The variable however is not global since it is in Angular's scope.

- The different properties being shipped off to the component and more based on the DOM can be found by looking up HTML DOM element object properties.

- Clicking the button now will just get you an error which tells you that you don't have a function you need.
- Add some logic to your component file. The **submittedDataText** field is just an example to show how to store the state of the form and can be used for what you like. It won't be used in this exercise.

form / name-address.component.ts

```
export class NameAddressForm {
submittedDataText: string;
data: any;

    submittingForm1($event, data): void {
        $event.preventDefault();
        console.log('Data 1 was submitted.');
        this.submittedDataText = data.nameFirst + ' ' + data.nameLast;
    }
    showEachPropertyOf(data): void {
        for (var eachKey in data) {
            console.info(eachKey, '=', data[eachKey]);
        }
    }
    showStringified(data): void {
        console.info('Stringified data:', JSON.stringify(data, null,
2));
    }
}
```

- Now run the form again after entering some data, submit it, and check the console. You should have no errors and a confirmation that the data was submitted.
- Add a few more lines of code to call the remaining methods to see how they work. The stringified method works for checking all the data the best. Also, we'll add a line to show how to clear a form field and put the cursor back into the text field where you want it. This simulates a round-trip of entering data and starting again.

form / name-address.component.1.html

```
            nameFirstParentNode: nameFirst.parentNode.nodeName
        };
        submittingForm1($event, data);
        showEachPropertyOf(data);
        showStringified(data);
        nameFirst.value = '';
        nameFirst.focus();
      "> Submit when valid</button>
```

- Run the form again after entering some data (Chrome may by this time will allow you to auto-enter data with a double-click.)
- You'll see some good console output including a JSON object for passing to the server, you'll initialize the first form field and put the cursor there for the user to start over again.
- For now, it's not a good idea to try to deal with the error messages until we have more control over the inputs.

29. Binding to ngModel – template driven

We'll just adjust the templates on the previous exercise for changing this to automatic bindings and then add some real functionality to the component.

- Make a copy of **form/name-address.component.1.html** and call it **form/name-address.component.2.html.**
- Update your templateUrl in to use the new file.

form /name-address-component.ts

```
styleUrls: ['name-address.component.css'],
templateUrl : `name-address.component.2.html`
```

- Update the h3 element in the template file to this:

form / name-address.component.2.html

```
<h3>2 Bound data - simple</h3>
```

Setting up the model

- The project file server where you've been getting your files has some data to grab (or use the pdf if you have it but check for extra LF/CRs. Copy and paste the contents into this form project directory in the same directory and file. It looks like:

form / model / formSchemas.ts

```
// form schemas
// ----------------------------------------------------------------
export class Address {
    private _street: string;
    private _city: string;
    private _state: string;
    private _zip: string;
    constructor(
        street?: string,
        city?: string,
        state?: string,
        zip?: string ){
        this._street = street || '';
        this._city = city || '';
        this._state = state || '';
        this._zip = zip || '';
    }
    reset() : void {
        this.street = '';
        this.city = '';
        this.state = '';
```

```
            this.zip = '';
        }
        get street() {
            return this._street;
        }
        set street(arg: string) {
            this._street = arg;
        }
        get city() {
            return this._city;
        }
        set city(arg: string) {
            this._city = arg;
        }
        get state() {
            return this._state;
        }
        set state(arg: string) {
            this._state = arg;
        }
        get zip() {
            return this._zip;
        }
        set zip(arg: string) {
            this._zip = arg;
        }
        get oneLine() {
            return this.street + ", " + this.city + ", " + this.state + ' '
+ this.zip;
        }
    }
    // ---------------------------------------------------------------------
    export class Name {
        private _first: string;
        private _last: string;
        constructor(first?: string, last?: string) {
            this._first = first || 'John';
            this._last = last || 'Smith';
        }
        reset() : void {
            this.first = '';
            this.last = '';
        }
        get first() {
            return this._first;
        }
        set first(nameIn: string) {
            this._first = nameIn;
```

```
        }
        get last() {
            return this._last;
        }
        set last(nameIn: string) {
            this._last = nameIn;
        }
        get oneLine() {
            return this.first + ' ' + this.last;
        }
    }
    // ---------------------------------------------------------------------
    export class Phone {
    }
```

- We will use the properties of Name and Address objects in the NameAddressForm component to structure our data. Update your NameAddressForm class by starting with the import to get the class in scope.

form \ name-address.component.ts

```
import {Component}   from '@angular/core';
import {Name, Address} from './model/formSchemas';
```

- Then add fields referencing the model classes and a constructor to the component:

form \ name-address.component.ts

```
export class NameAddressForm {
    submittedDataText: string;
    data: any;
    private _name: Name;
    private _address: Address;

    constructor() {
        this._name = new Name();
        this._address = new Address();
    }
```

- Also add the getters and setters / properties to complete the access to the fields :

form \ name-address.component.ts

```
    get name(): Name {
        return this._name;
    }
    set name(arg: Name) {
        this._name = arg;
    }
    get address(): Address {
        return this._address;
    }
```

```
        set address(arg: Address) {
            this._address = arg;
        }

        submittingForm1($event,data): void {
```

Using the model

- We'll delete much of the submit function and change it so that the elements themselves are being sent to the component.

form / name-address.component.2.html

```
   <button type="submit" class="button" (click)="
          submittingForm2([nameFirst, nameLast]);
            ">Submit when valid</button>
```

- Finally, add the function to handle the submitted data as an array of the name fields

form / name-address.component.ts

```
        showStringified(data): void {
            console.info('Stringified data:', JSON.stringify(data, null,
2));
        }
        submittingForm2($event, data): void {
                $event.preventDefault();
                console.info('Form 2 was submitted.');
                console.log('Submitted data', data);
                console.info('Model is now: ', this.name, this.address);
        }
```

- When you add data to the form and submit it you'll see a similar console output as before with new lines about the submitted data and the default model state.
    - Look at the submitted elements.
    - The form does not clear and the default model has not been updated without binding to the form.
- This sets up the model but does not bind them to the inputs on the form. Add in the model bindings.

form / name-address.component.2.html

```
            <input id='first' type="text" [(ngModel)]="name.first"
#nameFirst >

        </div>
        <div class="large-6 columns">
            <label for="last">Last Name:
                <span class ='label alert' >err</span>
            </label>
            <input id='last' type="text" [(ngModel)]="name.last" #nameLast >
```

- Using ngModel requires us to update our module configuration with another two lines to import it from the Forms module like so:

form / app.module.ts

```
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import {NameAddressForm} from './name-address.component';

@NgModule({
  imports:      [ BrowserModule, FormsModule],
```

- Another recent change since RC5 is the requirement to add name attributes to the input element. We'll use a different naming convention so as not to confuse what is going where.

form / name-address.component.2.html

```
            <input id='first' type="text" [(ngModel)]="name.first"
#nameFirst name='name-first'>
        </div>
        <div class="large-4 columns">
            <label for="last">Last Name:
        <span class ='label alert' >err</span>
    </label>
            <input id='last' type="text" [(ngModel)]="name.last" #nameLast
name='name-last'>
```

- Then show the changes to the model as the inputs change them through the oneLine property in the model classes

form / name-address.component.2.html

```
        <b>Preview for model data</b>: {{name.oneLine}} lives at
{{address.oneLine}}<br>
```

- Make changes to the input fields and watch the immediate change to the output above them.
- The form will have your default data on it. Change it as you like. As you change the data, notice the binding that changes the preview area for the model as you type.
- Submit the data with the new model data. Look at the submitted elements and the validity states that have been added.

Reset the form

- We should erase the data when you submit the old data so you have a clean form to look at. Add this code to the submit function and use any data you like.

form / name-address.component.ts

```
        submittingForm2($event, data) {
            $event.preventDefault();
```

```
            console.info('Form 2 was submitted.');
            console.log('Submitted data', data);
            console.info('Model is now: ', this.name, this.address);
            this.name.reset();
        }
```

- Now when you test the code, you will see the name fields clear after you submit and the model and model preview will clear also.
- An alternative way to clear the fields in the data array is to set the values to an empty string like so:

```
          data.forEach(field => { field.value = ''; })
```

- Try that and refresh the browser again to check out that the form does clear the fields on submit. This does not affect the model so it will not clear the model preview.

Finish binding the model to the form

- We should finish adding the bindings for the form data to the Address model object. Add these bindings to the appropriate inputs in the template.

form / name-address.component.2.html

```
[(ngModel)]= 'address.street' #addressStreet name='address-street'
[(ngModel)]= 'address.city' #addressCity name='address-city'
[(ngModel)]= 'address.state' #addressState name='address-state'
[(ngModel)]= 'address.zip' #addressZip name='address-zip'
```

- There's a reset function in the Address class also to use after the name reset().

form / name-address.component.ts

```
    this.name.reset();
    this.address.reset();
```

- Now when you enter data into the address fields, it will simultaneously update the preview. The component also knows about the changes so the submit function can access the fields. Try it out.

30. Binding to ngForm – template driven

We'll just adjust the templates on the previous exercise for changing this to automatic bindings and then add some real functionality to the component.

- Make a copy of **form/name-address.component.2.html** and call it **form/name-address.component.3.html.**
- Update your templateUrl in your component file to use the new template file.

form / name-address.component.ts

```
        styleUrls: ['name-address.component.css'],
        templateUrl : `name-address.component.3.html`
```

- Update the h3 element in the **form/name-address.component.3.html** template file to this:

form /name-address.component.3.html

```
      <h3>3 Bound data with ngForm</h3>
```

Control the form with ngForm

- We'll remove the submit function from the click event on the button and move it up to the submit event on the form element. The button will be a normal button now like this:

form / name-address.component.3.html

```
    <button type="submit" class="button">Submit when valid </button>
```

- And the form element will listen for the submit event and send a copy of itself to a submit function. We'll also add the focus just to show that more code can be added even though it's better to use the HTML5 autofocus attribute on the element.

form / name-address.component.3.html

```
   <form (ngSubmit)="submittingForm3($event, nameAddressForm);
nameFirst.focus();" #nameAddressForm="ngForm">
```

- Make a stub with a few lines from the previous submit method for this submit function so it will not error out when you test

form / name-address.component.ts

```
        submittingForm3(formData) {
             $event.preventDefault();
             console.info('Form 3 was submitted.');
             console.info('Model is now: ', this.name, this.address);
        }
```

- Build and view the form and the console output should look like it did previously.
- Let's see what the form object has inside of it for us by outputting it to the console.

form / name-address.component.ts

```
    submittingForm3(formData) {
       $event.preventDefault();
       console.info('Form 3 was submitted.');
       console.info('Model is now: ', this.name, this.address);
       console.info('ngControl form data:', JSON.stringify(formData.value));
    }
```

- Add data to the form and submit it. The form input values from the **name** attributes are registered with the form and become the local variable #nameAddressForm.

Add style

- We'll add some CSS styles to react to the changes in the data states by putting these styles in our component's CSS.

form / name-address.component.css

```
    input.ng-valid[required] {
        border-left: 5px solid #42A948;
    }
    input.invalid, input.ng-invalid {
        border-left: 5px solid #a94442;
        background-color: hsl(0,100%,97%);
    }
```

- This won't change anything immediately. It will require us setting a rule on the fields to require input. You can add as many as you want but put at least one on the first name like this:

form / name-address.component.3.html

```
    <input id='first' type="text" [(ngModel)]="name.first" #nameFirst
name='name-first' required>
```

- Delete the data in the first name field where the green left border is. You should see the invalid state rules invoked with a red left border in the field and a light red form background. Then put some data back in and the valid state rules will change the view again.
- Add an element to show the style that is used on first name input element like this:

form / name-address.component.3.html

```
            <label for="first">First Name:
                <span class ='label alert' >err</span>
                   <span>{{nameFirst.className}}</span>
            </label>
```

- Now when you refresh the page, you will not get what you expect. You should be getting the class names to appear but Angular is mixed up about which ones to show since it evaluates the page several times over the lifecycle of the component. The new way to manage it is to stop it in the debugger. Click the blue play button to have it continue.
- We can ask it to check just when a change is made by us, or when we push data to it. Here's where you make those changes in the component:

form / name-address.component.ts

```
import {Component, ChangeDetectionStrategy} from '@angular/core';
import {Name, Address} from './model/formSchemas';

@Component({
    moduleId: module.id,
    selector: 'name-address-form',
    styleUrls: ['name-address.component.css'],
    templateUrl: 'name-address.component.3.html',
    changeDetection: ChangeDetectionStrategy.OnPush
```

- That will get rid of the debugger stepping in the lifecycle. Refresh your browser to make sure.
- Now click in the first name field and out again into a few fields and the class names will show up eventually. Add the same kind of spans on other input elements to see their validity states by CSS classes if you want.
- Test the different kinds of states by clicking in the text box (touched), changing the data (dirty), and removing the data from the required text box (invalid).

Using validity states

- We can show the states of the entire form by checking the validity like this

form / name-address.component.3.html

```
          <span class ='label secondary'
[class.hide]='!nameAddressForm.pristine'>Unchanged</span>
          <span class ='label warning'
[class.hide]='nameAddressForm.pristine'>In progress</span>
          <span class ='label success'
[class.hide]='!nameAddressForm.valid'>Ready to submit</span>
```

- Try changing the fields by causing invalid states and then back to valid to see that the labels appear when they should.
- We can also use the valid state on the submit button to make sure it doesn't appear when it shouldn't be used.

form / name-address.component.3.html

```
   <button type="submit" class="button"
[disabled]='!nameAddressForm.valid'>Submit when valid</button>
```

- Test out the button by removing one of the required fields' data.
- You can add messages wherever you like with the form.valid style of checking like this:

form / name-address.component.3.html

```
   <button type="submit" class="button"
[disabled]='!nameAddressForm.valid'>{{nameAddressForm.valid ? 'Send data' :
'Fix problems before sending'}}</button>
```

- Or we could update those err labels now with

form / name-address.component.3.html

```
   <span class ='label alert' [class.hide] =
"!nameFirst.className.includes('ng-invalid')" >err</span>
```

- But there's a better way still coming. For now, try out the CSS class name check method of validation on a few more input elements to clean up the form for practice.

31. Binding to formGroup – reactive forms

- Make a copy of **form/name-address.component.3.html** and call it **form/name-address.component.4.html.**
- Update your templateUrl in **components/forms/name-address.ts** to use the new file.

form / name-address.component.4.html

```
    styleUrls: ['name-address.component.css'],
    templateUrl : `name-address.component.4.html`
```

- Update the h3 element in the **form/name-address.component.4.html** template file to this:

form / name-address.component.4.html

```
    <h3>4 Bound data with formGroup </h3>
```

- Also, we'll need to add the Reactive Forms module in to our declarations in the **app.module.ts** file like this

form / app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

import { AppComponent}  from './app.component';
import {NameAddressForm} from './name-address.component';

@NgModule({
  imports:      [ BrowserModule, FormsModule, ReactiveFormsModule],
```

- You could remove the FormsModule declaration since we won't be using that style of form in this exercise for minimum code.

Setting up the FormBuilder

- We are going to use the FormGroup and FormBuilder in the next few steps without using an import and then do that later so you will see some red squiggly lines in Code for a minute.
- The FormGroup that will be bound to our ngFormModel will be created in the component. We'll keep a property for it so we can reuse it if necessary. Add that to the component file first along with the FormBuilder:

form / name-address.component.ts

```
    private _name: Name;
    private _address: Address;
    private nameAddressFormGroup: FormGroup;
    private builder: FormBuilder;
```

- Then we'll initialize the FormBuilder in the constructor where it is injected as an argument to the constructor.

form / name-address.component.ts

```
        constructor( builder: FormBuilder) {
            this.builder = builder;
            this._name = new Name();
            this._address = new Address();
        }
```

- You can run the form to see what happens but the Angular team has hidden much of the messy errors of the past so you don't get much now. On that note, they are providing much better explanations of how to fix problems that used to be vague.
- Both FormBuilder and FormGroup are from a different package so we must import them with this line in the component file:

form / name-address.component.ts

```
    import { Component, ChangeDetectionStrategy } from '@angular/core';
    import { FormBuilder, FormGroup, FormControl} from '@angular/forms';
    import {Name, Address} from './model/formSchemas';
```

- Building the form now will not show errors which used to come from a lack of a provider which validates the form.

Creating the FormGroup

- Let's create a new submit function under the last submittingForm3 function in the component file to handle the differences in this form as we change it. Make a stub with a few lines from the previous submit method for this submit function so it will not error out when you test.

form / name-address.component.ts

```
    submittingForm4(formData) {
                console.info('Form 4 was submitted.');
    }
```

- We'll update the ngSubmit event handler in the template to use submittingForm4 instead of submittingForm3. And we'll switch over to using the FormGroup and bind it to the model which we'll create in code next.

form / name-address.component.4.html

```
    <form (ngSubmit)="submittingForm4(nameAddressFormGroup);
nameFirst.focus();" [formGroup]='nameAddressFormGroup'>
```

- The form reference has changed so we must update the old reference in several places:

form / name-address.component.4.html

```
            <span class ='label secondary'
[class.hide]='!nameAddressFormGroup.pristine'>Unchanged</span>
```

```
            <span class ='label warning'
[class.hide]='nameAddressFormGroup.pristine'>In progress</span>
            <span class ='label success'
[class.hide]='!nameAddressFormGroup.valid'>Ready to submit</span>
```

- and

form / name-address.component.4.html

```
   <button type="submit" class="button"
[disabled]='!nameAddressFormGroup.valid'>
        {{nameAddressFormGroup.valid ? 'Send data' : 'Fix problems before
sending'}}
   </button>
```

- Test out the form but you will still have an error. Angular wants a validator because that's what it does with a form group when it's submitted. The object has to be created so we'll create a method to build the group and its controls called buildFormGroup( ) and call it in the constructor. Here's where we call the method we have to write:

form / name-address.component.ts

```
        constructor(builder: FormBuilder) {
            this.builder = builder;
            this._name = new Name();
            this._address = new Address();
            this.buildFormGroup();
        }
```

- And here's where we have a method to make the FormGroup and bind all the FormControls on our form to it.

form / name-address.component.ts

```
        private buildFormGroup(): void {
            this.nameAddressFormGroup = this.builder.group({
                'nameFirst': [this.name.first],
                'nameLast': [this.name.last],
                'addressStreet': [this.address.street],
                'addressCity': [this.address.city],
                'addressState': [this.address.state],
                'addressZip': [this.address.zip]
            });
        }
```

- The last step is to make sure the form fields have their name declared to bind to the code. The names are attributes with the key of formControlname. Here are the input elements after we've rewritten them for reactive forms. Replace the elements in your template with these.

form / name-address.component.4.html

```
   <input id='first' type="text" formControlName="nameFirst" #nameFirst
required>
```

```
   <input id='last' type="text" formControlName='nameLast' #nameLast>
   <input id='street' type="text" formControlName='addressStreet'
#addressStreet>
   <input id='city' type="text" formControlName='addressCity' #addressCity>
   <input id='state' type="text" formControlName='addressState'
#addressState>
   <input id='zip' type="text" formControlName='addressZip' #addressZip>
```

- Now there should be no errors when you submit your data. The model data will not update. But we have more options to build in validators and reference the data.

Retrieving the data

- Let's make sure we're getting the data from the form and put in a few checks when we submit. The form data being submitted is the nameAddressFormGroup so the output will be exactly the same.

form / name-address.component.ts

```
   submittingForm4(formData) {
      console.info('Form 4 was submitted.');
      console.info('form data:', JSON.stringify(formData.value));
      console.info('nameAddressFormGroup data:',
          JSON.stringify(this.nameAddressFormGroup.value));
   }
```

- Let's access the data that's in the form through the FormGroup. We'll write a function to provide the formatting for us and output it to the form. The function, which can be appended to the NameAddressForm class in the component file is this:

form / name-address.component.ts

```
       showGroupDataInOneLine(formData : FormGroup) {
           return formData.controls['nameFirst'].value + ' ' +
                  formData.controls['nameLast'].value + ', ' +
                  formData.controls['addressStreet'].value + ', ' +
                  formData.controls['addressCity'].value + ', ' +
                  formData.controls['addressState'].value + ' ' +
                  formData.controls['addressZip'].value;
       }
```

- Then we'll add the call on the form at the top for a preview as we type

form / name-address.component.4.html

```
       <b>Preview for model data</b>: {{name.oneLine}} lives at
{{address.oneLine}}<br>
       <b>Preview for form data</b>:
{{showGroupDataInOneLine(nameAddressFormGroup)}}<br>
```

- This provides a second way to interactively see the values on the form as we type or as data comes in from the server.

- Here's one more way similar to what we just did that will output the data to the console again when we submit the form. It is appended to the submittingForm4( ) function:

form / name-address.component.ts

```
      console.info('nameAddressFormGroup data:',
          JSON.stringify(this.nameAddressFormGroup.value));
      let keysAndValues: string = '';
      for (const key in formData.controls) {
          keysAndValues += key + ' = ' + formData.controls[key].value +
'\n';
      }
      console.info(keysAndValues);
   }
```

Validators and error messages

- Now we'll set up the validators. We need to import this class as well at the top of the class in the component file.

form / name-address.component.ts

```
   import { FormBuilder, FormGroup, FormControl, Validators} from
'@angular/forms';
```

- Then we'll add the different types of validators to the FormGroup that the FormBuilder is creating for us in our buildFormGroup function.

form / name-address.component.ts

```
   private buildFormGroup(): void {
       this.nameAddressFormGroup = this.builder.group({
           'nameFirst': [this.name.first, Validators.required],
           'nameLast': [this.name.last, [Validators.required,
Validators.minLength(3)]],
           'addressStreet': [this.address.street, Validators.required],
           'addressCity': [this.address.city],
           'addressState': [this.address.state, Validators.maxLength(2)],
           'addressZip': [this.address.zip, [Validators.required,
Validators.pattern('[0-9]{5}(-[0-9]{4})?')]]
       });
   }
```

- Check on the form that the CSS shows the invalid state when you
    - don't have a first name
    - have a last name with fewer than three characters
    - don't have an address
    - have more than two characters in the state
    - don't have a zip or use something other than a digit except when you use a zip+4 format (12345-1234).

- Now we'll adjust the error messages to show some feedback as well. I've always found that it takes a little coding to do a good job on error messages so I've done a little more than just enough. We'll be adding two fields to the class and two functions to manage them
    - an array of error messages which can be externalized or initialized through a database
    - an object to manage when an error message should appear
- Here's the fields for NameAddressForm class

form / name-address.component.ts

```
        private builder: FormBuilder;
        errorMessages: any;
        whenValidStateFor: any;
```

- Here's the call to run the functions to be written that executes from the NameAddressForm constructor:

form / name-address.component.ts

```
            this.buildFormGroup();
            this.setErrorMessages();
            this.updateValidState();
    }
```

- And here's the functions appended to the end of the NameAddressForm class which you can customize however you want.

form / name-address.component.ts

```
    updateValidState(): void {
        var groupControls = this.nameAddressFormGroup.controls;
        this.whenValidStateFor = {
            nameFirst: groupControls['nameFirst'].valid ||
groupControls['nameFirst'].pristine,
            nameLast: groupControls['nameLast'].valid ||
groupControls['nameLast'].pristine,
            addressStreet: groupControls['addressStreet'].valid ||
groupControls['addressStreet'].pristine,
            addressCity: true,
            addressState: groupControls['addressState'].valid ||
groupControls['addressState'].pristine,
            addressZip: groupControls['addressZip'].valid ||
groupControls['addressZip'].pristine
        };
    }
    private setErrorMessages(): void {
        this.errorMessages = {
            nameFirst: 'required',
            nameLast: 'Use at least 3 characters',
            addressStreet: 'required',
            addressState: 'x2',
            addressZip: '5 or 5-4 digits'
```

```
        };
    }
```

- Run the form one more time just to check that you haven't developed any problems.

Wire up the form to show stuff

- That little button that says Show Form has been wanting some attention. Here's a little function to show the form data, the event and stop the form from submitting. Add this to your NameAddressForm class at the bottom also.

form / name-address.component.ts

```
    showForm(form, event): void {
        console.info(form, event);
        this.updateValidState();
        console.info(this.whenValidStateFor);
        event.preventDefault();
    }
```

- Hook that up to the button with this code on the template:

form / name-address.component.4.html

```
    <button class='tiny button'
    (click)='showForm(nameAddressFormGroup, $event);'>Show form</button>
```

- Test out the button at various times to check the status of the form properties via the FormGroup. The event isn't that interesting but it shows how to get access to it.
- Now we can adjust the error messages to be all that they should be. Anytime there's a piece of code like:

form / name-address.component.4.html

```
    <span class ='label alert'>err</span>
```

- We will replace it with another piece of code. Here are the replacements that you can use. They should be easy to figure out where to put them.

form / name-address.component.4.html

```
    <span class ='label alert' [class.hide]= 'whenValidStateFor.nameFirst'
>{{errorMessages.nameFirst }}</span>

    <span class ='label alert' [class.hide]= 'whenValidStateFor.nameLast'
>{{errorMessages.nameLast }}</span>

    <span class ='label alert' [class.hide]=
'whenValidStateFor.addressStreet' >{{errorMessages.addressStreet }}</span>

    <span class ='label alert' [class.hide]= 'whenValidStateFor.addressCity'
>{{errorMessages.addressCity }}</span>
```

```
   <span class ='label alert' [class.hide]=
'whenValidStateFor.addressState' >{{errorMessages.addressState }}</span>

   <span class ='label alert' [class.hide]= 'whenValidStateFor.addressZip'
>{{errorMessages.addressZip }}</span>
```

- If you want to clean up your form you can remove these or any like these class name status messages:

form / name-address.component.4.html

```
      <span>{{nameFirst.className}}</span>
      <span>{{nameLast.className}}</span>
```

Detect changes

- Check the form out and you'll notice that nothing is happening. The easy way I fix this is first to understand that the validity states are not being updated. Then we put a call into our update validity function anywhere on the template in moustaches (an interpolation) so that every time it evaluates, it gets run.

form / name-address.component.4.html

```
   <form (ngSubmit)="submittingForm4(nameAddressFormGroup);
nameFirst.focus();" [ngFormModel]='nameAddressFormGroup'>
   {{updateValidState()}}
```

- Check the form out and try the invalid fields again to see the error messages appear. Some messages won't appear until you enter in something in an invalid state then erase it again to get rid of the pristine state. Try changing some of the whenValidStateFor states to see how that works.
- So that was a cheap observable essentially. We can do better.
- First, get rid of the {{updateValidState()}}.
- We'll add the OnInit interface and implement it with the right observable pattern. Put the declaration for the interface in the first line of the component:

form / name-address.component.ts

```
   import { Component, ChangeDetectionStrategy, OnInit } from
'@angular/core';
   import { FormBuilder, FormGroup, FormControl, Validators} from
'@angular/forms';
```

- Add the declaration to use the interface on the class:

form / name-address.component.ts

```
   export class NameAddressForm  implements OnInit {
```

- Then implement the required function to listen for the changes in any value on the form group with a callback to update our valid states.

form / name-address.component.ts

```
  ngOnInit() {
      this.nameAddressFormGroup.valueChanges.subscribe(value => {
        this.updateValidState();
      });
    }
```

32. Use a US state drop down

- Here's an implementation of using a real list of US states for the form as a drop down. You should have the **form/model/states.ts** in the project files directory online. We'll use that to provide a good set of data for the drop down. Create a new file in your project under the **form** directory in the model directory called **model/states.ts**, copy the contents from online, and save it.
- First, you must import it into the component. Add this to the top of your component class

form / name-address.component.ts

```
import {States} from './model/states';
```

- Add it as a field to your class

form / name-address.component.ts

```
whenValidStateFor: any;
states = States.withAbbreviations;
```

- Then in the **name-address.component.4.html** file, swap out the old addressState input text box for this:

form / name-address.component.4.html

```
<select id='state' formControlName='addressState'>
    <option *ngFor="let state of states" [value]="state.abbreviation">
{{state.name}}</option>
</select>
```

- Take a look to see if it's working OK. You might want to change your column formatting a little to see the state name fully and shorten the zip code like this:

form / name-address.component.4.html

```
<div class="large-2 columns"> <label for="state">St:...

<div class="large-2 columns "> <label for="zip ">Zip:...
```

- So if the validation is still working (it is) and the state that appears is longer than two characters, why aren't we seeing an error message? Can you change something to test whether you are right or not?
- Here's a test object to add to your states to get you going

form / model / states.ts

```
        {
            "name": "State of No Return",
            "abbreviation": "SoNR"
        },
```

33. Custom validators

Adding a custom validator is just writing a service that follows an interface. Follow the rule on the class, add the service, and use the validator.

Write the validation and messages for the city

- The city was left alone so we can add a custom validator to it. In our last exercise in the **form / name-address.component.ts** component, we made the guts for our validation. Let's add the validation to be written where the FormBuilder initializes it in buildFormGroup( ) function

form / name-address.component.ts

```
 'street': [this.address.street, Validators.required],
  'addressCity': [this.address.city,
   ValidateCity.notWichita_notStartWithK_min3],
  'state': [this.address.state, Validators.maxLength(2)],
```

- Then we'll update the validity state that shows a message based on that validator we added in the updateValidState( ) function. We will not show a message when the city passes validation or when the text field has not been modified.

form / name-address.component.ts

```
addressStreet: groupControls['addressStreet'].valid ||
   groupControls['addressStreet'].pristine,
addressCity: groupControls['addressCity'].valid ||
   groupControls['addressCity'].pristine,
addressState: groupControls['addressState'].valid ||
   groupControls['addressState'].pristine,
```

- And finally, we'll write an error message that is appropriate to the validation in the setErrorMessages( ) function.

form / name-address.component.ts

```
    addressStreet: 'required',
    addressCity: 'did not pass custom city check',
    addressState: 'x2',
```

- Of course, without a validator to use, our code will produce and error when we run the form, but check it out anyway. You should get this in the console with other error text.

```
ValidateCity is not defined
```

Write and use the validator

- A custom validator is considered a service so we'll add a **form.validator.ts** file to our form directory and start the class out like this:

form / form.validator.ts

```
import {FormControl} from '@angular/forms';

interface ValidationResult {
    [key: string]: any;
}

export class ValidateCity {
    public static notWichita_notStartWithK_min3(control: FormControl):
ValidationResult {
        var errors = {};
        if (!control.value) {
            errors['no city'] = true;
        }
        if (control.value.match(/Wichita/i)) {
            errors["I don't like Wichita"] = true;
        }
        if (control.value.match(/^K.*/i)) {
            errors["Don't start with a K"] = true;
        }
        if (control.value.length < 3) {
            errors["I need more letters"] = true;
        }
        for (var anyError in errors) {
            if (errors.hasOwnProperty(anyError)) {
                return errors;
            }
        }
        return null;
    }
}
```

- The validator gets handed our control that we add the validator to and we then have to send back a result based on the interface that is described above. You don't need an interface but it makes things explicit and checkable.
- We're going to check for no value, the value of a case-insensitive Wichita, any value that starts with 'K' or 'k', and any value and is less than three characters long.
- Let's import that validator into our **form/name-address.component.ts** file and set it up. Here's the import

form / name-address.component.ts

```
import {States} from './model/states';
import {ValidateCity} from './form.validator';
```

- The validator can be attached to the Control at several points. We put it in when the FormBuilder was making our ControlGroup and Groups in the buildFormGroup() function
- Here's an alternate way to do that:

form / name-address.component.ts

```
    'zip': [this.address.zip, [Validators.required, Validators.pattern('[0-
9]{5}(-[0-9]{4})?')]]
            });
            this.nameAddressFormGroup.controls['addressCity'].validator =
ValidateCity.notWichita_notStartWithK_min3;
    }
```

- Now run the form and start typing in the city field. These things should not work and provide the error message that we set:
    - two characters or less after typing
    - Wichita
    - Kansas City, Kanopolis, Kanorado or any other city starting with K or k
- The customCity function also provides errorMessages which are available in the city.errors property. We'll add a class to the **form.validator.ts** file to help provide a nice format to them:

form / form.validator.ts

```
export class ValidationUtilities {
    public static getErrorString(errors: ValidationResult): String {
        var separator = '';
        var errorString = '';
        for (let key in errors) {
            if (errors[key]) {
                // only use error message when true
                errorString += separator + key;
                separator = ', ';
            }
        }
        return errorString;
    }
}
```

- Add another import to use this new class in our component

form / name-address.component.ts

```
import {ValidateCity, ValidationUtilities } from './form.validator';
```

- Then we'll swap out our static error message for a formatted version of the errors that the validator is providing us with in the setErrorMessages( ) function in the component file.

form / name-address.component.ts

```
        addressStreet: 'required',
        addressCity:
ValidationUtilities.getErrorString(this.nameAddressFormGroup.controls
['addressCity'].errors),
```

- The error message doesn't show on the city field yet, so you will want to call the setErrorMessages function again in the updateValidState() function.

form / name-address.component.ts

```
        addressZip: groupControls['addressZip'].valid ||
            groupControls['addressZip'].pristine
      };
      this.setErrorMessages();
  }
```

- Now try the form. You will get custom messages and multiple messages will appear depending on the rules that fail.

Turn the validator into a directive

- We'll take our model-driven validator and turn it into a template-driven validator with a directive that wraps it. It's simple to do.
- Add this code to the very bottom of the **form.validator.ts** file:

form / form.validator.ts

```
@Directive({
  selector: '[grumpy-city]',
  providers: [
      {
        provide : NG_VALIDATORS,
        useValue: ValidateCity.notWichita_notStartWithK_min3,
        multi: true
  }]
})
export class GrumpyCityValidator {}
```

- The directive wraps the method and provides a selector to use it. For special types of forms, you may see that there is an additional selector required like '[grumpy-city][ngModel]' or '[grumpy-city][formControlName]' to restrict it to just one type of form.
- You'll need a few imports to stop any dependency errors so add these to the top of the file as well:

form / form.validator.ts

```
import { FormControl} from '@angular/forms';
import {Directive} from '@angular/core';
import { NG_VALIDATORS } from '@angular/forms';
```

- The module file **app.module.ts** needs a few small updates as well to declare the use of it throughout the app

form / app.module.ts

```
import { GrumpyCityValidator} from './form.validator';

@NgModule({
  imports:      [ BrowserModule, FormsModule, ReactiveFormsModule],
  declarations: [ AppComponent, NameAddressForm, GrumpyCityValidator ],
```

- It's now hooked in to the component so we just have to disable the validator in the **form/name-address.component.ts** file in the buildFormGroup() function with either

form / name-address.component.ts

```
      'addressCity': [this.address.city]
    // ,ValidateCity.notWichita_notStartWithK_min3
```

- or if you used the alternate method, just comment it out

form / name-address.component.ts

```
    // this.nameAddressFormGroup.controls['addressCity'].validator =
ValidateCity.notWichita_notStartWithK_min3;
```

- The attribute to get the directive to validate the city goes on the template. In your **form/name-address.component.4.html** add the attribute

form / name-address.component.4.html

```
    <input id='city' type="text" formControlName='addressCity' grumpy-city>
```

- Make sure you are saving all the files as you go. Then run the form and the validation should occur as previous but this time it's template-driven and designers can use them to customize the elements of the template without using JavaScript.