Building Web Applications with Angular

QUICKLABS GUIDE

CONTENTS

QuickLabs Environment Set-Up

Code Editing

1. Open VSCode (or download and install if not present).
    - Use the *desktop shortcut* to open the VSCode download page:
    - For Windows users download the 64-bit System Installer.
2. Check for *updates* and download and install if necessary:
    - For Windows Users click Help - Check for updates;
    - For MacOS Users click Code - Check for updates.
3. Using File - Open, navigate to the QuickLabs folder and click Open. This will give you access to all of the QuickLab files and solutions needed to complete the QuickLabs.

NodeJS

1. Use the *desktop shortcut* to open the NodeJS download page.
2. Download and install the LTS version for the operating system you are working in:
    - For Windows users, download the Installer file (.msi);
    - For MacOS users, download the Installer file (.pkg).

Do This Before Each QuickLab

Unless specifically directed to do otherwise, the following steps should be taken before starting each QuickLab:

1. Point the terminal/command line at the QuickLab starter folder that contains the package.json.
2. Run the command:

```
npm i
```

3. Compile and output the project by running the command:

```
npm start
```

4. Navigate the browser to:

https://localhost:4200

if it does not open automatically.

Quick Lab 1 - Angular Development Environment

Objectives

- To be able to set up the Angular CLI;
- To be able to create a new Angular application using the CLI;
- To be able to start an Angular application running from the CLI;
- To install and examine Augury;

Overview

In this Quicklab, you will set up and use the Angular CLI. The Angular CLI will be used to create a new application and then start the application running in the browser. Finally, you will install the Augary plug-in for Chrome and examine the application using it.

Activity

Skip the 'Before Each QuickLab' for this Activity.

1. Using CTRL + ' on the keyboard (CTRL + ` on MacOS) or by using click-path View – Terminal (or Terminal – New Terminal on MacOS), open VSCode's integrated terminal or click the terminal icon on the bottom bar. 
2. Install the *Angular CLI* globally on your machine by running the command:

```
npm i -g @angular/cli
```

3. Using the `cd` command, navigate to the QuickLabs/01_IntroToAngular/ folder.
4. Create a *new* Angular application using the command:

```
ng new starter
```

5. Choose `y` for the 'strict typing' option.
6. Choose `N` for the 'add routing option'.
7. Choose `CSS` for the next option.

Wait for the installation to complete.

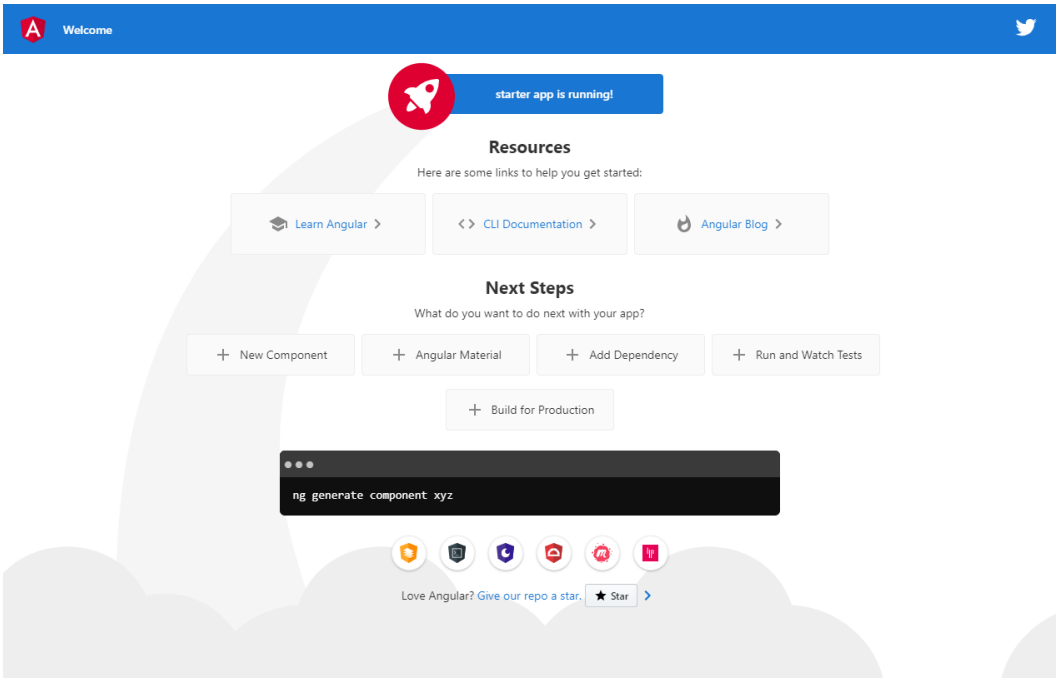
8. Change into the starter folder using the command:

```
cd starter
```

9. Run the application by using the command:

```
ng serve --open
```

Your browser should open at http://localhost:4200 with the following screen:



10. Open another browser tab and search for **Augury**.
11. Go to the **Chrome Store** link and add the **Augury** extension to **Chrome**.
12. Open the **Developer Tools** in the tab running your application - you should see an **Augury** tab - *if you don't close the browser, open it and navigate to the application*.

There are 3 tabs in **Augury**:

- **Component Tree** - shows the hierarchy of Angular components in the app;
- **Router Tree** - shows the hierarchy of routing in the app;
- **NgModules** - shows all of the Angular Modules that are being used by the app.

This is the end of Quick Lab 1

Quick Lab 2 - Angular Architecture

Objectives

- To be able to understand the structure of an Angular project
- Be able to identify key files and where key building blocks are in an Angular Project
- To be able to use the docs keyword from Angular CLI

Overview

In this QuickLab, you will examine the folder structure of an Angular project set up by the CLI. Then you will explore some of the key files within the folder structure, identifying how Angular implements some of the building blocks. Using the built-in documentation functionality, you will look at any APIs, Classes, etc that you feel you need to. Finally, you will fire up the application and change some of the content.

Activity

Complete the section 'Before Each QuickLab' before continuing using the solution folder rather than starter.

1. In VSCode, open the folder QuickLabs/02_AngularArchitecture/solution.
2. Explore the project folder, locate the root module for the application.

   (It's called app.module.ts and can be found in the src folder.)

Note that there are *3 JavaScript module imports* to make Angular's `NgModule` and `BrowserModule` available to the application as well as the `AppComponent`.

The `NgModule` decorator has:

- `declarations` of the `AppComponent` as this primarily belongs to this module;
- `imports` of `BrowserModule`;

  Use the `ng doc BrowserModule` command on the terminal/command line to find out more about this essential module;
- `providers` being an *empty array* as the module/application has no services;
- `bootstrap` of `AppComponent` as this is the *root module* and Angular needs to know which component it should render first.

5. Open the app.component.ts file and look for:

- The *name of the element* that this component will be rendered into;
- The *name of the HTML template* used to create the view for this component;

- The *name of the CSS style sheet* used to style the component;
- Any *class variables* that are declared.

6. Open the **template** for the component and look for:

- Databinding of any class variables.

7. Open **index.html** and locate the element where the app will be rendered.

If you feel confident enough, run `ng serve --open` and change the display of the heading on the page by modifying the value of the appropriate variable.

This is the end of Quick Lab 2

Quick Lab 3 - Jasmine

No Quick Lab for this section.

Quick Lab 4a - Creating a new Angular Component

Objectives

- To be able to use the Angular CLI to create a new component
- To be able to nest components in others

Overview

In this QuickLab, you will create a new Angular Component using the CLI, exploring the files that are created and modified as part of the process. You will then nest this new component in the existing App component.

Activity

Complete the section 'Before Each QuickLab' before continuing.

1. Move the terminal/command line into the **src/app** folder of **04_AngularComponents/starter** and create a new **Component** called `MyFirstComponent` using the Angular CLI command:

```
ng generate component MyFirstComponent
```

Top Tip: the `generate` *command can be shortened to just* `g`

2. Verify that the **Component** has been placed in its own folder and that the **Component** is *declared* by the `AppModule`.
3. Find the name of the **selector** that `MyFirstComponent` should be placed in from the **my-first-component.component.ts** file and make a note of it.
4. Open **app.component.html** and add an HTML tag at the bottom of the supplied code using the name of the selector.

For example, **app.component.ts** is rendered in **app-root** so `<app-root></app-root>` can be found in **index.html**.

5. Save the file and run the application by using the command:

```
ng serve --open
```

The browser window will open, and there should be a screen like the one shown below!

This is the end of Quick Lab 4a

Quick Lab 4b - Editing, Styling and Binding Data to Templates

Objectives

- To be able to add HTML to a template
- To be able to add CSS styling to a template
- To be able to use data-binding to pass data from component to template, vice-versa and bi-directional

Overview

In this QuickLab, you will modify a component's template and CSS styling to change its content and appearance. You will then examine the different types of data-binding, passing some data from the component to the template. Next, the way Angular passes data from the template to the component will be examined and how 2-way binding can be done using the NgModel directive.

Activity

Skip the 'Before Each QuickLab' for this Activity.

1. In VSCode, open the file my-first-component.component.html from the QuickLabs/04_AngularComponents/starter/src/app/my-first-component folder.
2. Add a few lines of HTML - anything you like - include *ids* and *classes* if you wish!
    - We added an `<h2>` and a `<p>`, both with some text
3. Save the file and verify that the output updates in the browser.
4. Open the component's CSS file and add some styling to change the appearance of the HTML you added.
5. Save the file and verify that the output updates in the browser.

Data Binding from Component to Template - Interpolation

1. Open my-first-component.component.ts and add a `readonly public` class variable of `paraText` of type string set to any string of your choosing.
2. Back in the template HTML, add the interpolation of `paraText` in a paragraph.

```
<p>{{paraText}}</p>
```

Activity – Data Binding from Component to Template - Property, Class, Attribute and Style

1. In my-first-component.component.ts, add the following variables to the class:

```
public readonly title=`My Data Binding Experiments`;
public readonly myClasses: string = `red right underlined`;
public readonly imgUrl: string = `../assets/qalogo.svg`;
public selected = false;
public inputtedText = ``;
```

2. Save the file.
3. Copy the CSS from the file my-first-component-styles.txt (in the src/app folder) to the my-first-component.component.css to allow us to work with some CSS.
4. In my-first-component.component.html, add:
    - A *heading tag* that uses the *class* `myClasses` and the `title` *class variable*:

```
<h1 [class]="myClasses">{{title}}</h1>
```

    - An *image* that uses the *class variable* `imgUrl` as its `src` attribute:

```
<img [src]="imgUrl" alt="QA Logo" width="100"/>
```

    - A *paragraph* that has `class.red` set to `true` and some text inside it:

```
<p [class.red]="true">Lorem ipsum…</p>
```

    - A *paragraph* that sets the `style.background-color` `deepskyblue` or `limegreen` dependent on whether `selected` is *true or not*, with some text inside it:

```
<p [style.background-color]="selected ?
'deepskyblue':'limegreen'">
    Lorem ipsum…
</p>
```

Activity – Data Binding Events

1. Add a button that has a click event that flips the status of selected:

```
<button (click)="selected = !selected">Flip selected</button>
```

2. Save the file and check that the button click has the desired effect.

Activity – Two-way Data Binding

1. Open the **app.module.ts** file from the **src** folder for editing.
2. Perform a *JavaScript import* of the `FormsModule` from `@angular/forms`:

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

3. Add the imported `FormsModule` to the **imports** in the array in the `NgModule` decorator:

```
…
imports: [
  BrowserModule,
  FormsModule
],
…
```

4. Save the file and return to **my-first-component.component.html**.
5. Add an *input* of **type** `text` that has a *banana-in-a-box* `ngModel` set to `inputtedText`:

```
<input type="text" [(ngModel)]="inputtedText" />
```

6. Add a *paragraph* beneath this that has `inputtedText` interpolated inside it:

```
<p>{{inputtedText}}</p>
```

7. Save the file and view the browser output. Type in the text box.

You should see text you enter in the box appear in a paragraph beneath it.

This is the end of Quick Lab 4b

Quick Lab 4c - Testing Components

Objectives

- To be able to write tests that verify the component class
- To be able to write tests that verify that the DOM is rendered correctly with the data bindings from the class

Overview

In this QuickLab, you will write some Jasmine tests that ensure that any functions and values in the component class works as expected. Following this, you will create tests that allows the actual DOM element to be examined, checking that the expected data-binding has occurred.

Activity

Skip the 'Before Each QuickLab' for this Activity.

1. Start the testing by opening an additional or new command line pointing somewhere in the project folder and entering the command:

```
ng test
```

You should notice that there are **4 tests** and that all 4 appear to pass. However, examining the console output shows some errors!

This is due to the dependencies of the components on other components and directives. The test setup is unaware of these at the moment.

Activity – Testing AppComponent

The first error to fix is the `AppComponent` 'NG0304 app-my-first-component is not a known element' message. The error is due to the `<app-my-first-component>` tag that is used. There are 2 strategies we could employ here:

- Add `MyFirstComponentComponent` into the declarations of the `TestBed` configuration
- Stub the component

The first would not suffice as `MyFirstComponentComponent` has dependencies on other modules that are not catered for in the `TestBed`. Since we want to test components in isolation, we will create a stub:

1. Open **app.component.spec.ts** for editing.
2. Under the **imports** and *before the first test suite* add a `@Component` decorator that has a `selector` of `'app-my-first-component'` and an *empty string* as the **template** for its meta data
3. This **decorator** should **decorate** a class called `MyFirstComponentStubComponent`.

```
@Component({
  selector: 'app-my-first-component',
  template: ''
})
class MyFirstComponentStubComponent {}
```

4. In the *first test suite* 'AppComponent' add `MyFirstComponentStubComponent` to the declarations.
5. Save the file and the test should rerun, with one remaining error message.

Activity – MyFirstComponentComponent Tests

Angular has provided a my-first-component.component.spec.ts file. However, this test is producing an error - 'NG0303 Can't bind to ngModel since it isn't a known property of 'input''

This test should to fail to create the component. This is because it has a dependency on the `FormsModule` and the `TestBed` is unaware of it.

1. Add `FormsModule` to the imports in the configuration of the `TestBed` in the `waitForAsync` `beforeEach` call:

```
…
  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      declarations: [MyFirstComponentComponent],
      imports: [FormsModule] // ensure it is also imported at
  })                         // the top of the file from
…                            // @angular/forms
```

2. Save the file and observe that all tests now pass without warnings.

Activity – Testing the DOM

In this section, you will set up another suite of tests to test the rendering of the DOM.

1. Open my-first-component.component.spec.ts for editing.
2. Under the existing `describe` block add *another* with the title 'DOM Testing'.
3. The function should set 3 variables as follows:
   - `fixture` of type

`ComponentFixture<MyFirstComponentComponent>` (imports from @angular/core/testing);

- `myFirstComponentDe` of type `DebugElement` (imports from @angular/core);
- `myFirstComponentEl` of type `HTMLElement`.

4. Add a `beforeEach` function that is `waitForAsync` - the body of which should:
    - Call `TestBed.configureTestingModule` with an *object* that has `MyFirstComponentComponent` as a `declaration` and `FormsModule` as `imports`;
    - Then calls `compileComponents()`.

```
  beforeEach(waitForAsync() => {
    TestBed.configureTestingModule({
      declarations: [MyFirstComponentComponent],
      imports: [FormsModule]
    })
      .compileComponents();
  });
```

5. Add a `beforeEach` function that has:
    - `fixture` set to a call to `TestBed`'s `createComponent` passing in `MyFirstComponentComponent`;

```
fixture = TestBed.createComponent(MyFirstComponentComponent);
```

    - `myFirstComponentDe` set to `fixture.debugElement`;
    - `myFirstComponentEl` set to `myFirstElementDe.nativeElement`.

A first nested suite will test that *6 paragraphs* are rendered. The solution shows several other tests that could be performed on paragraph rendering.

6. Nest a `describe` suite called 'Testing Paragraphs' in the existing suite.
7. Inside the 'Testing Paragraphs' suite, create an `it` spec called 'it should render 6 paragraphs'.
8. Inside the *spec*, add a `const` called `paragraphs` and set this to be a call to `querySelectorAll` on `myFirstComponentEl`, looking for `'p'`.

```
…
  const paragraphs = myFirstComponentEl.querySelectorAll('p');
…
```

9. `expect` the `length` of this *returned array* to be `6`.

```
…
  expect(paragraphs.length).toBe(6);
…
```

10. Save the file and then check the testing output.
11. For peace of mind, change the expected value to 5 and check that it fails.
12. Return the test to a passing status.

Other tests that could be run here are:

- Checking for a paragraph with an `id` of `normal` and containing some expected text;
- Checking there is a paragraph with a CSS `class` of `red`;
- Checking the last paragraph initially has no text (as it is controlled by the user);
- Checking there is initially a paragraph that has a `background-color` of `limegreen`;
- Checking there is a paragraph with a `background-color` of `deepskyblue` when the component property `selected` is `true`.

Tests in the solution also include that the `src` for the *image* is correct and that the `alt` text is as expected.

Testing the button click changes the status of selected in the class can be done as follows:

13. Write *another nested suite* called 'Button click testing'.
14. Add a *spec* to the suite called 'clicking the button should change the state of selected'.
15. Inside the *spec*, declare a `const button` and set this to a `query` call on `myFirstComponentDe`, passing in `By.css('button')`.
16.

```
      const button =
myFirstComponentDe.query(By.css('button'));
```

17. Call `fixture.detectChanges();`
18. Set an *expectation* that the value of `fixture.componentInstance.selected` will be `falsey`.
19. Simulate a click on the button using the `tiggerEventHandler` call on the button, passing in `'click'` and `null`.

```
      button.triggerEventHandler('click', null);
```

20. Call `fixture.detectChanges()` to update the component.
21. Set an *expectation* that the value of `fixture.componentInstance.selected` will be `truthy`.
22. Simulate another click on the button and update the component.
23. Set an expectation that the value of `fixture.componentInstance.selected` will be `falsy`.
24. Save the file and run the tests.

All tests should pass.

This is the end of Quick Lab 4c

Quick Lab 5a - Structural and Attribute Directives

Objectives

- To use Attribute and Structural directives

Overview

In this QuickLab, you will start by looking at how attribute directives can be used to affect the template of a component. Following this, you will look at how structural directives can be used to add multiple elements to the template as well as conditionally selecting if particular mark-up is displayed.

Activity – ngClass Attribute Directive

Complete the section 'Before Each QuickLab' before continuing.

1. In VSCode, open the file my-first-component.component.html from the QuickLabs/05_AngularDirectives/5a/starter/src/app/my-first-component folder.
2. Underneath the current code, add a `<section>` that has the following attributes:
    - `[ngClass]` set to `currentClasses`;
    - `(mouseover)` set to `currentClasses.selected=true`;
    - `(mouseout)` set to `currentClasses.selected=false`.
3. In the section add a *paragraph* with some text - we used *50 lorem ipsum words*.
4. In my-first-component.component.ts add another *class variable* called `currentClasses`, setting it to:
    - Be `public` and `readonly`;
    - Be of type **object**;
    - Contain a **key** of `selected` with a **value** of `false`.
5. Save both files and view the page in the browser.

You should see that the selected class styling is only applied when the mouse is over the section boundaries.

Activity – ngStyle Attribute Directive

1. In my-first-component.component.html, add another `<section>` under the previous one with attributes:
    - `[ngStyle]` set to `currentStyles`;
    - `(mouseover)` set to `selected=true; setCurrentStyles()`;

- `(mouseout)` set to `selected=false; setCurrentStyles()`.

2. In the *section* add a *paragraph* with some text - we used *50 lorem ipsum words*.
3. In **my-first-component.component.ts** add another *class variable* called `currentStyles`, setting it to be:
    - `public`;
    - Of type **object**;
    - Initialised as an *empty object*.
4. In the class, add a method called `setCurrentStyles()` that has *no return type*.
5. Make the method body set `currentStyles` to have key/value pairs:
    - `backgroundColor` conditionally set to `lightpink` or `lightgreen` dependent on `selected`
    - `border` conditionally set to `5px solid red` or `5px solid green` dependent on `selected`
6. Save both files and view the page in the browser.

You should see that, initially, the section is not styled at all. When you hover the mouse over this section, it changes its style (as well as the paragraph further up the page whose style depends on `selected` also!) When you remove the mouse from the section, it changes to the style defined for `selected`'s *false* state. Adding a call to `setCurrentStyles()` to the constructor changes this behaviour so that the false state is initially set.

Activity – *ngIf

1. In the **my-first-component.component.ts** file, add a *public class variable* called `viewPara`, initialised as `false`.
2. In the **template**, add a `<button>` that has a *click event handler* that changes the *status* of `viewPara` to its opposite (i.e. `!viewPara`).
3. Add a *paragraph* that displays the current status of `viewPara`.
4. Add a *paragraph* to the **template** with an `*ngIf` property set to `viewPara`.
5. Populate the *paragraph* with some text.
6. Save the file and see what happens as you click the button that changes the status of `viewPara`.

Activity – *ngFor

1. In the **my-first-component.component.ts** file, `import` `users` from the file **users.ts** in the **src** folder.

2. Add a *public class variable* of `users` of type `{username: string, upvotes: number, downvotes: number} []` set to be the `users` imported.
3. In the **template**, add `<h3>` to title a **'List of Users'**.
4. Add a `<ul>` with a `<li>` that has a `*ngFor` structural directive set to *loop* each `user` in the `users` array.
5. Set the content of the `<li>` to be `user.username`.
6. Save the file and view the output.

Activity – *ngSwitch

1. In the **my-first-component.component.ts** file, add a *public class variable* `selectedUser` to be an **object** that has a **key** `username` of **type string**.
2. In the **template**, add a *click handler* to the `<li>` that sets the `selectedUser` to be the *current* `user`.
3. Under the list, add a `<div>` that has an `[ngSwitch]` attribute set to an *optional* `selectedUser`'s `username`:

```
<div [ngSwitch]="selectedUser?.username">…</div>
```

4. For each of the `usernames` in the `users` array, create a *paragraph* that is an `*ngSwitchCase` for the `name` as a **string** and populate the *paragraph* with some text, so for example, Chris's paragraph would be:

```
<p *ngSwitchCase="'Chris'">Chris enjoys cycling</p>
```

5. Add a final *paragraph* that is `*ngSwitchDefault` inviting users to **'Select an instructor'**.
6. Save the file and click on each instructor. The paragraph should change dependent on who you have selected.

This is the end of Quick Lab 5a

Quick Lab 5b - Input and Output Properties

Objectives

- To understand how to use Input and Output

Overview

In this QuickLab, you will add an Input property to a component so that it can receive data from its parent when rendering. Secondly, you will see how a component can emit data that can be caught by its parent in the form of an event.

Activity

Skip the 'Before Each QuickLab' for this Activity.

1. Point the terminal/command line at the **starter/src/app** folder for **QuickLab 5a** and create a new **component** called **user**. Using the command:

```
ng g component user
```

2. In the **user.component.ts** file, add an `@Input` decorator binding a variable called `user` of **type object** to it.
3. Add an `@Output` property called `vote` that is a `new EventEmitter` with **type number**:

```
@Output()
vote = new EventEmitter<number>();
```

4. Add `Input`, `Output` and `EventEmitter` to the **imports** in this file from **@angular/core**.
5. Add a *method* called `upvote` that takes no arguments and has a body that calls `emit` on `vote` with a *value* of `1`.

```
upvote() { this.vote.emit(1); }
```

6. Add a *method* called `downvote` that takes no arguments and has a body that calls `emit` on `vote` with a *value* of `-1`.
7. In the **user**'s **template**, add HTML that:
    - Creates an `ng-container` that *only displays if* a `user` exists;

```
<ng-container *ngIf="user">
```

    - Has a *paragraph* that shows which `user` has been passed to the component as an *input*;
    - A *button* that has an `id` of `upvote`, a *click handler* that calls `upvote`, content of a *thumbs up* (👍) and displays the `user upvotes` property after it;

- A *button* that has an `id` of `downvote`, a *click handler* that calls `downvote`, content of a *thumbs down* (👎) and displays the `user downvotes` property after it;
- A *paragraph* that displays the *total votes cast* (`user.upvotes + user.downvotes`).

```
<ng-container *ngIf="user">
  <p>
    {{user.username}} has been passed as the current
    instructor
  </p>
  <button id="upvote" (click)="upvote()">👍</button>
  {{user.upvotes}}
  <button id="downvote" (click)="downvote()">👎</button>
  {{user.downvotes}}
  <p>Total votes: {{user.upvotes + user.downvotes}}</p>
</ng-container>
```

8. Open my-first-component.component.ts and add a *method* to the class called `handleVote`. It should:
    - Take `event` as an argument;
    - Have a *return type* of `void`;
    - *Log out* that `` `a vote was made` `` and the *event object received*;
    - Set a `const votedUsers` as the *class'* `users` array;
    - Loop through the `votedUsers` array taking each `user` and:
        - Checking to see if the `user`'s `username` is equal to the `selectedUser`'s `username`
            - If it is, see if the `event` is *equal* to `1` and **add** the *value* to the `user`'s `upvotes` property, otherwise **subtracting** the *value* from the `user`'s `downvotes` property:

```
for (const user of votedUsers) {
    if (user.username === this.selectedUser.username) {
        (event === 1) ? user.upvotes += event :
           user.downvotes -= event;
    }
}
```

    - Set the class' `users` to the `votedUsers` array.
9. In my-first-component's **template**, add an `<app-user>` element to insert the `User` component giving *attributes*:
    - `[user]="selectedUser"`
    - `(vote)="handleVote($event)"`

10. Ensure that all files are saved before returning to the browser and ensuring that all of the upvote and downvote buttons work.

Notice that the votes for each person are held in the application. Will they still be there if you refresh the application?

This is the end of Quick Lab 5b

Quick Lab 5c - More Component Testing

Objectives

- To understand how to test a component that has a nested component
- To be able to test a component with Inputs and Outputs

Overview

In this QuickLab, you will stub nested components to allow the parent to be tested as a unit. After this, you will write tests to check that the Input and Output functionality passes data as expected.

Activity

Skip the 'Before Each QuickLab' for this Activity.

1. Make sure the command line is pointing at the project and then run the tests using:

```
ng test
```

Note that there are now test errors due to the additional `User` component. The `TestBed` for the `MyFirstComponentComponent` does not know about the `UserComponent` and therefore throws an error.

To resolve this, a 'Stub' component will be created to ensure that `MyFirstComponentComponent` can compile and be tested.

2. Open my-first-component.component.spec.ts for editing.
3. Between the *imports* and the *first test suite* add code to create a `Component` class within this file called `UserStubComponent`. The decorator should have:
    - `selector` set as `app-user`;
    - `template` set as an *empty string*.

```
…
@Component({        // Make sure Component is imported
  selector: 'app-user',
  template: ''
})

…
The actual component has an input called user, so this will
also be simulated in the stub.
```

4. In the `UserStubComponent` class add an `@Input()` with `user` of type object.

```
class UserStubComponent {
  @Input() user: object;      // Make sure Input is imported
}
```

5. Make the all test suites aware of the `UserStubComponent` by adding it to the list of `declarations` in all `TestBed.configureTestingModule` calls.
6. Save the file and look at the test output. It should now have passing tests again.
7. Stop the currently running test and retest showing the coverage of the code by the tests:

```
ng test --code-coverage
```

Once the tests have run, you should see that an additional folder called coverage has been added to the root of the project.

8. Open the coverage report by opening the file coverage/index.html in the browser.

This report will show that although some of `MyFirstComponentComponent` has been tested, there is still about *a third of its code not exercised by tests*.

9. Click through the links until you end up at the code for the Component.

You will find that the functions `setCurrentStyles` and `handleVote` added earlier are the code not tested.

Activity – Testing setCurrentStyles

As this is just a function that receives no inputs and sets class values, it can be tested in the class-testing style.

1. In the *first test suite*, add a spec with the string: `` `should set styles based on the value of selected when setStyles() is called` ``
2. The body of the arrow function should:
    - Set a variable `expectedCurrentStylesFalse` as an *object* with:
        - a key of `backgroundColor` set to value `lightgreen`;
        - a key of `border` set to value `5px solid green`.
    - Set a variable `expectedCurrentStylesTrue` as an *object* with:
        - a key of `backgroundColor` set to *value* `lightpink`;
        - a key of `border` set to *value* `5px solid red`.
    - `expect` the value of `component.selected` to be `falsy`;
    - Call the `setCurrentStyles` function on component;
    - `expect` the value of `component.currentStyles` *to equal* `expectedCurrentStylesFalse`;
    - Set the value of `component.selected` to `true`;
    - Call the `setCurrentStyles` function on `component`;

- `expect` the value of `component.currentStyles` *to equal* `expectedCurrentStylesTrue`;

3. Save the file and observe the test output.

The new test should pass and inspection of the code coverage should reveal the component is now just over 70% tested.

Activity – Testing handleVote(event)

1. Add another spec under the last one with a title of: `should add the supplied vote to the supplied user's vote count when handleVote() is called`.
2. The arrow function body should:
    - Set `component.users` to an *array* containing a *single object* with keys of:
        - `username` set to `TestUser`,
        - `upvotes` set to `0`;
        - `downvotes` set to `0`.
    - Set `component.selectedUser` to an *object* with a key of `username` and a value of `TestUser`;
    - Call `handleVote` on `component` with a value of `1`;

```
component.handleVote(1);
```

- `expect` `component.users[0]` *to equal* a call to `jasmine.objectContaining` with an *object* of `upvotes:1`

```
expect(component.users[0]).
  toEqual(jasmine.objectContaining({upvotes: 1}));
```

- Call `handleVote` on `component` with a value of `-1`;

```
component.handleVote(-1);
```

- `expect` `component.users[0]` *to equal* a call to `jasmine.objectContaining` with an *object* of `downvotes:1`:

```
expect(component.users[0]).
  toEqual(jasmine.objectContaining({downvotes: 1}));
```

3. Save the file and observe both the test output and the code coverage.

You should find that all tests pass and the code coverage is now 100% for `MyFirstComponentComponent`.

You will notice that the `UserComponent` is only 80% tested. The final 20% of the code can be tested as below.

Activity – Testing UserComponent's upvote() and downvote()

1. Open user.component.spec.ts for editing.
2. In the `beforeEach` call, add a `const` of `expectedUser` with a value of `{ username: `TestUser`}` and set the `component.user` to be the `expectedUser` - this should go immediately BEFORE the call to `fixture.detectChanges()`.
3. Add a spec with a title of `` `should emit a value of 1 when upvote is clicked` ``.
4. Inside the arrow function:
    - Declare a variable called `voteValue` of type number;
    - Declare a `const` called `upVoteButton` of type DebugElement and set it to `fixture.debugElement.query(By.css('button#upvote'))`;

Note: DebugElement should be imported from @angular/core.

Note: By should be imported from @angular/platform-browser.

- Subscribe to the Observable created by `component.vote`, passing in `value` and setting `voteValue` to `value`:

```
…
    component.vote.subscribe(value => voteValue = value);
…
```

- Call `triggerEventHandler` on `upVoteButton` passing in `'click'` and `null`.

```
    upVoteButton.triggerEventHandler('click', null);
```

- `expect` `voteValue` *to be* `1`.

The setups for these tests are almost exactly the same - for downvote, simply copy the test and replace upvote with downvote and 1 for -1 in the expected value.

This is the end of Quick Lab 5c

Quick Lab 5d - Custom Directives

Objectives

- To create a custom directive and apply it to an already existing application.

Overview

In this QuickLab, you will create a custom directive that is applied to a set of images. As the mouse is moved over an image, it will change from being in grayscale to being in full colour.

Activity

Complete the section 'Before Each QuickLab' before continuing.

In this exercise you are going to build a custom *Attribute Directive* which we can use to apply a *grayscale->full-colour* effect triggered by a user hovering over any element we place it on.

1. In VSCode, point the terminal/command line to QuickLabs/05_AngularDirectives/5d/starter/src folder.
2. Create a new MODULE called `shared` - this will be where we put the directive to be created.

```
ng g module shared
```

3. Change into the folder created for the module and create a directive called grayscale:

```
ng g directive directives/grayscale
```

4. In the constructor for the class of the *Grayscale directive*, inject a `private ElementRef` called `el`:

```
…
    constructor(private el: ElementRef) {}
…
```

5. Make the directive implement `OnInit` by adding it to the class declaration and add `OnInit` to the imports from @angular/core:

```
import { … , OnInit } from '@angular/core';
…
class GrayscaleDirective implements OnInit {
…
}
```

6. Within the class, under the constructor add an `ngOnInit` method that:

- Sets `el`'s `style.filter` property to `grayscale(100%)`;
- Sets `el`'s `style.transition` property to `filter 1s`:

```
…
    ngOnInit() {
        this.el.nativeElement.style.filter =
`grayscale(100%)`;
        this.el.nativeElement.style.transition = `filter 1s`;
    }
…
```

7. Add the `GrayscaleDirective` to the **exports** from the `SharedModule`.
8. **Import** the `SharedModule` into the `InstructorsModule`.
9. Apply the **directive** to the **instructors-gallery** component by adding `appGrayscale` to the attributes of the **img** tag for the thumbnails.
10. Save all files and launch the application.

You should find that the images at the bottom of the page are grayed out.

Activity – React to User Activity

We are going to colourise the image when a user hovers the mouse pointer over it. To do this, we are going to detect when a user has mouse entered and/or mouse left the image.

1. Declare a *method* called `onMouseEnter`, decorating it with `@HostListener` that has a parameter of `mouseenter` and a method body that:
    - Sets `el`'s `style.filter` property to `grayscale(0%)`.

```
…
    @HostListener('mouseenter') onMouseEnter() {
        this.el.nativeElement.style.filter = `grayscale(0%)`;
    }
…
```

2. Declare a *similar method* called `onMouseLeave`, decorating it with `@HostListener` that has a parameter of `mouseleave` and a method body that:
    - Sets `el`'s `style.filter` property to `grayscale(100%)`.
3. Save all files and check the application.

Images should now transition to full colour when they are hovered.

Activity – Taking Input from Consumers

An important aspect of Directives is their ability to take input from the consumers of them. In this instance, it would be good to be able to specify how much grayscale filter should be used.

1. Open grayscale.directive.ts for editing.
2. Create an `@Input` property in the *directive class* called `appGrayscale` and set its value to `100%` (for default):

```
…
    @Input() appGrayscale = `100%`;
…
```

This means the consumers can simply use the directive attribute itself to set the value rather than adding another attribute to the element in addition to the directive selector.

3. In all places within the *directive class* where the `grayscale` value has been set to `100%`, *replace the value* with `this.appGrayscale`:

```
…
    this.el.nativeElement.style.filter =
    `grayscale(${this.appGrayscale)`;
…
```

4. Update the instructors-gallery component so that we pass `50%` to `appGrayscale` within the `<img>` tag.

```
<img appGrayscale="50%" [style.border]="instructor…>
```

5. Save all files and check the application.

You should find that the images start more colourful now (as less grayscale is being applied) and go full-colour when hovered.

This is the end of Quick Lab 5d

Quick Lab 5e - Testing Custom Directives

Objectives

- To be able to test a custom directive.

Overview

In this QuickLab, you will write Jasmine tests to ensure that the grayscale custom directive behaves as expected.

Activity

Skip the 'Before Each QuickLab' for this Activity.

You are going to test the Custom Directive.

1. Open grayscale.directive.spec.ts for editing from the 5d-e/starter/src/app/shared/directives folder.

Running the tests now would result in an error as the constructor receives an `ElementRef`.

2. In the existing suite's `it` spec, declare a `const` `element` of type ElementRef (imported from @angular/core) and *set* it to `null`.
3. Add `element` as an argument to the instantiation of the `new GrayscaleDirective` in the `declaration` of `directive`.

```
const directive = new GrayscaleDirective(element);
```

4. Run the test and check that all pass.

The Directive will need a *'wrapping component'* so that it can be applied to it.

5. In between the *imports* and the *first test suite*, add a `Component` class called `TestGrayscaleComponent` with a decorator template of:

```
@Component({
  template: `
  <img src="../../assets/images/adrian.png" appGrayscale="50%"
/>`
})
class TestGrayscaleComponent { }
```

6. Create a *new test suite* with a title of 'Testing Directive: Grayscale' and an arrow function body that:
    - Declares a variable `component` of type TestGrayscaleComponent;
    - Declares a variable `fixture` of type ComponentFixture<TestGrayscaleComponent>;

*Note: **ComponentFixture** is imported from **@angular/core/testing**.*

- Declares a variable `imageEl` of type DebugElement;

  *Note: **DebugElement** is imported from **@angular/core**.*

- Has a `beforeEach` function that has an arrow function body that:
  - Calls `TestBed.configureTestingModule` with:
    - declarations of `GrayscaleDirective` and `TestGrayscaleComponent`.
  - Sets `fixture` to `TestBed.createComponent(TestGrayscaleComponent)`;
  - Sets `component` to `fixture.componentInstance`;
  - Sets `imageEl` to `fixture.debugElement.query(By.css('img'))`;

    *Note: **By** is imported from **@angular/platform-browser**.*

  - Calls `fixture.detectChanges()`;

7. Write a *spec* that has a title `` `should become 0% grayscale on mouseenter and 50% on mouseleave` `` and an arrow function body that:
   - *expects* `imageEl.nativeElement.style.filter` *to equal* `` `grayscale(50%)` ``;
   - Calls `triggerEventHandler('mouseenter', null)` on `imageEl`;
   - Calls `fixture.detectChanges()`;
   - *expects* `imageEl.nativeElement.style.filter` *to equal* `` `grayscale(0%)` ``;
   - Calls `triggerEventHandler('mouseleave', null)` on `imageEl`;
   - Calls `fixture.detectChanges()`;
   - *expects* `imageEl.nativeElement.style.filter` *to equal* `` `grayscale(50%)` ``;
8. Save the file and observe the test output - all tests should still pass.

This is the end of Quick Lab 5e

Quick Lab 6a - Observables and RxJS

Objectives

- To investigate Observables provided by RxJS.

Overview

In this QuickLab, you will create some observables using the built-in creation operators of and from Event. In the second part, you will filter data from the outputs of some observables. You will then examine the use of combination operators subscribing to two or more observables. Finally, you will use some of the transform operators to change the data provided by the observables.

Activity

Complete the section 'Before Each QuickLab' before continuing.

Initially, the browser window will open, showing some mark-up that does not do anything!

- Open the file index.ts from the 06_Observables/6a/starter/src folder for editing.
- Import `{ Observable }` from rxjs.

Activity – Part 1 – Creation Operators

'of'

An observable operator that simply creates an observable of a list of given values - in this case, sequential numeric values.

1. Create an Observable called `ofOperatorObservable` that uses the `of` operator to *emit the values* 0, 1, 2, 3, 4, 5.
2. Subscribe to the observable, passing the *emitted value* into the *callback* and *logging it out*, prefixing `` `of emitted: ` `` to the *outputted text*.
3. Save the file and check the output of the observable on the console.

Ensure that `of` has been added to the list of imports from rxjs.

'fromEvent'

An observable operator that creates an observable of a given event - in this case, detect a button was clicked,

1. Create a variable called `fromEventButton` and set it to *select the button with the id of fromEvent*.
2. Create an Observable called `fromEventOperatorObservable` that uses the `fromEvent` operator to *detect a click event* on the `fromEventButton`.
3. Ensure that `fromEvent` has been imported from rxjs.

4. Subscribe to the observable, passing the *event in to the callback* and *logging out*:

```
    `The ${(<HTMLInputElement>event.target).id} button raised
a ${event.type} event`
```

5. Save the file and check the output of the observable when the 'Click Me!' button is clicked.

Note: Use `npm start` from the terminal pointing at the starter folder to run this application.

Activity – Part 2 – Filter Operators

'filter'

An observable operator that can intercept the emitted values and filter out those that are not wanted - in this case only show even numbers.

1. Create an Observable called `isEven` that uses the `of` operator to *emit the values* 0-5.
2. Create a subscription to the Observable:
    - `pipe` the `filter` function so that *only EVEN values* are passed through;

```
…
    isEven.pipe(filter((value: number) => value % 2 === 0))
…
```

    - Make the subscribe function *return the values logged out to the console*.
3. Save the file and view the output on the console.

Ensure that `filter` is imported from rxjs/operators.

'take'

An observable operator that restricts the number of values that are emitted to the subscribe function - in this case the first 5 values only.

1. Create an Observable called `takeFive` that *emits values* 1-10.
2. Create a subscription to the Observable:
    - `pipe` the `take` function so that *only the first 5 values* are passed through;
    - Make the subscribe function *return the values logged out to the console*.
3. Save the file and check the output on the console.

Activity – Part 3 – Combination Operators

'concat'

An observable operator that completes the first observable's emissions before continuing the output with the next observable's values - in this case the first five from the first at 1 second intervals and the first 10 from the second at 0.5 second intervals. This function should be imported directly from rxjs.

1. Create an Observable called `firstObservable` that *emits a value at 1 second intervals*, taking *only the first five* emissions.

```
…
let firstObservable = interval(1000).pipe(take(5));
…
```

2. Create a second Observable called `secondObservable` that *emits a value at 0.5 second intervals*, taking the *first ten* emissions.
3. Make a subscription to `firstObservable`, *piping* in `concat` and passing in `secondObservable`.
4. Make a `subscribe` call to `concat`, passing in `firstObservable` and `secondObservable` as arguments and *log out the values*.

```
…
concat(firstObservable, secondObservable).subscribe((value:
number => console.log(value);
…
```

5. Save the file and check the output on the console.

'merge'

Output values from one or more merged observables, emitted the values as they are delivered - in this case 5 emissions from the initial observable at 1s intervals, merging in 10 values from the next observable emitted every 0.5s. This function should be imported directly from rxjs.

1. Create an Observable called `thirdObservable` that *emits a value at 1 second intervals*, taking *only the first five* emissions.
2. Create another Observable called `fourthObservable` that *emits a value at 0.5 second intervals*, taking *the first ten* emissions.
3. Make a `subscribe` call to `merge`, passing in `thirdObservable` and `fourthObservable` as arguments and *log out the values*.
4. Save the file and check the output on the console.

Activity – Part 4 – Transformation Operators

'map'

An observable operator to change the values of each emission before returning the new value - in this case adding a # to the text supplied in an input when a connected button is clicked.

1. Create a variable called `hashtagifyButton` and *set it to select the button with the id of* ***hashtagify***.
2. Create an **Observable** called `hashtagify` that *emits on a click event on* ***hashtagifyButton***.
3. **Subscribe** to the `hashtagify` observable:
    - `pipe` the map function - it should:
        - Not take any values in the callback
        - Set a local variable called `toHashTag` to the **HTMLInputElement** with the `id` of `tohashtag`;

```
…
    let toHashTag =
<HTMLInputElement>document.querySelector('#tohashtag');
…
```

        - *Return the value* with `#` prepended to it.

```
…
    return `#${toHashTag.value}`;
…
```

4. Save the file and return to the browser.
5. Input in the box under the **Map** title and click the **'Apply Hashtag'** button.
6. Observe the output on the console.

This is the end of Quick Lab 6a

Quick Lab 6b - Observables and Angular

Objectives

- To use the in-built EventEmitter in Angular.

Overview

In this QuickLab, you will review how the Voter component previously created extends Observables via Angular's EventEmitter.

Activity

If you did not complete QuickLab 5c, run an npm install on the solution folder found in the 06_Observables/6b/

Activity – The voter component class

1. Open the file src/app/voter/voter.component.ts from your 5c work or the 06_Observables/6b/solution folder in VSCode.
2. Find the *2 Inputs* to the component:
    - One for `upvotes` which is of type number;
    - One for `downvotes` which is of type number.
3. Find the `Output` called `vote` that is set as a `new EventEmitter` of type number.

```
…
  @Output() vote = new EventEmitter<number>();
…
```

4. Find the *method* called `upvote` that calls `emit` with a *value* of `1` on `vote`.
5. Find the *method* called `downvote` that calls `emit` with a *value* of `-1` on `vote`.
6. Use the `ng doc` command to find out more about the `EventEmitter`

Activity – The voter component template

1. Open the file src/app/voter/voter.component.html in VSCode.
2. Find the following mark-up inside an ng-container tag:
    - A *button* with:
        - A *click handler* of `upvote()`;
        - *Text* of Upvote.

- Binding of the `upvotes` *input* (as text after the button).

```
<button (click)="upvote()">Upvote</button>{{upvotes}}
```

- A *button* with:
  - A *click handler* of `downvote()`;
  - *Text* of Downvote.
- Binding of the `downvotes` *input* (as text after the button).

Activity – The app component class

1. Open the file src/app/app.component.ts in VSCode.
2. Find the class variables `upvotes` and `downvotes` both set to `0`.
3. Find the method `handleVote` that:
   - Takes `vote` as an argument;
   - Uses a *ternary that checks* if `vote` is `1`:
     - Adds `vote` to `upvotes` if it is;
     - Subtracts `vote` from `downvotes` if not.

Activity – The app component template

1. Open the file src/app/app.component.html in VSCode.
2. Find the `app-voter` tag that has:
   - A `vote` *event handler* that calls `handleVote` passing in `$event`;
   - An attribute `[upvotes]` set to `upvotes`;
   - An attribute `[downvotes]` set to `downvotes`.

```
<app-voter
  (vote)="handleVote($event)"
  [upvotes]="upvotes"
  [downvotes]="downvotes"
>
</app-voter>
```

Click the Upvote and Downvote buttons and check that the numbers increase as expected. Asynchronous data is provided through the Voter component, via its EventEmitter emit function. This is just one example of how Angular extends Observables within it.

This is the end of Quick Lab 6b

Quick Lab 7 - Template-Driven Forms

Objectives

- To be able to create template-driven forms.

Activity

Complete the section 'Before Each QuickLab' before continuing.

Overview – Parts 1 to 3

In this part of the QuickLab, you will import the Modules that are needed work with template driven forms. You will add a form to the template and bind the data to the component class using the ngModel directive.

Part 1 – The FormsModule

1. Open 07_TemplateDrivenForms/starter/src/app/app.module.ts for editing.
2. Add `FormsModule` to the imports in the `NgModule` decorator, ensuring that it is also imported from @angular/forms at the top of the file.

Part 2 – The Template

A template form has been provided in the app.component.html. Inspect this file and note that the *Gender* field has been constructed using an `*ngFor` and an array called `genders` stored in the component class.

Part 3 – Data Binding with ngModel

1. Add `[(ngModel)]` to each of the *inputs* using the same *value* as the *input's name*.
2. Under the *form* add a `<h1>` with the content Outputs.
3. Add a `<p>` that has *text* for the *input name* and uses *data binding to show the value*.

Overview – Parts 4 and 5

In these parts of the QuickLab, you will examine how Angular can help to track the changes on the form. This provides an indication of whether a field's value is valid and allows CSS to provide users with immediate visual feedback.

Part 4 – Tracking state changes and validity

1. In app.component.html, locate the *input* for `firstname` add a *template reference variable* called `ngclasses`.

```
    <input type="text" id="firstname" name="firstname"
      [(ngModel)]="firstname" #ngclasses>
```

2. Under the *input* tag, add a *div* that contains *paragraph* that has the *text* `Classes:  {{ngclasses.className}}`.

```
…
    <div>
      <p>Classes: {{ngclasses.className}}</p>
    </div>
…
```

3. Save the file and view the output in the browser.

Examine what happens to the values as you click in and out of the input box and then change its value.

You should note that the field is always valid. That is because there is no HTML5 validation applied to the element. You should also note that the touched property only toggles when the element loses focus.

4. Add `required` to the attributes of `firstname`'s *input* tag.

Examine the effect on the valid property.

5. Add `minlength="2"` to the attributes of `firstname`'s *input* tag.

Examine the effect on the valid property.

Part 5 – Provide Visual Feedback on validity

1. Open the file app.component.css for editing.
2. Add a *CSS rule* for *input* elements with *classes* `ng-dirty` and `ng-invalid` AND *input* elements with *classes* `ng-touched` and `ng-invalid`:
    - Set the `border` property to be a `solid red` line that is `2px` thick.
3. Add a *CSS rule* for *input* elements with *classes* `ng-dirty` and `ng-valid`:
    - Set the `border` property to be a `solid green` line that is `2px` thick.
4. Save the file and return to the browser.

Check the output responds to what is expected.

Overview – Parts 6 to 8

In this part of the QuickLab, you will use template reference variables, ngclasses and the hidden attribute to display messages relating to the validity of fields on a form. Secondly, you will examine how submission of the form is managed by Angular, disabling the 'submit' button until the form is valid. Finally, you will use the form values to set data within the component class during submission, creating a new User object within the class.

Part 6 – Display validation errors

1. In the *input* tag for `firstname`, add another *template reference variable* called `propVals` and set to `ngModel`.

```
…
    <input type="text" id="firstname" name="firstname"
      [(ngModel)]="firstname"
      #ngclasses required minlength="2"
      #propVals="ngModel"
    >
…
```

2. Under the *paragraph* displaying ngclasses, add *6 more paragraphs* to output the value of the following `propVals` properties:
    - `valid`
    - `invalid`
    - `dirty`
    - `pristine`
    - `touched`
    - `untouched`
3. Save the file and view the output in the browser.

Examine what happens to the values as you click in and out of the input box and then change its value.

4. Insert a `<div>` between in the *input* and the *div* containing the *paragraphs*.
5. Add a `[hidden]` attribute that *examines* if the `valid` AND `dirty` properties are `true` OR the `pristine` property is `true`.
6. Add a *CSS class* of `validationMessage` to the `<div>`.
7. Populate the `<div>` with the content 'Name is invalid'.
8. Open src/app/app.component.css for editing.
9. Add a *CSS rule* for a *div* with the class of `validationMessage` that sets its `border` to `solid red` with a thickness of `2px`, the *background colour* to `lightpink`, a `margin-top` of `3px` and the *text colour* to `red`.
10. Save the file and return to the browser output.
11. Check that the validation message appears when it should.

Part 7 – Submit the form

1. In the *form* tag:
    - Add a *form reference variable* called `myForm` set to `ngForm`;

- Add an `ngSubmit` event handler that calls `onSubmit`.

2. In the *button* add a `disabled` attribute that evaluates the *valid state* of the form `myForm`.

```
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
  …
  <button type="submit" [disabled]="!myForm.form.valid">
    Submit
  </button>
</form>
```

3. In app.component.ts, add a class variable called `submitted` and set this to `false`.
4. Create a *method* called `onSubmit` that changes the *value* of `submitted` to `true`.

Part 8 – Data binding to a model in the component class

1. Create a class called `user` using the CLI, ensuring the terminal/command line is pointing to the src/app folder:

```
ng g class user
```

2. Inject `firstname`, `surname`, `age` and `gender` (with the correct type assignments: string, string, number, string) to the constructor of the class with *defaults* of an *empty string* for string fields and *null* for number fields.
3. Save the file and go back to the app.component.ts file.
4. Add a `user` set to a `new User()` (ensuring it is imported) to the `AppComponent` class.
5. For each form field, update the `[(ngModel)]` to prepend `user.` to the control name.
   So:

```
… [(ngModel)]="firstname" …
```

   would become:

```
… [(ngModel)]="user.firstname" …
```

   etc.

6. Add a line in the `onSubmit` method that *logs out* the `user`.

Note - this is where an AJAX call could be made to submit the data or where a call to a Service could be made to handle the data.

7. Save the file and check that the form works as expected.

This is the end of Quick Lab 7

Quick Lab 8 - Reactive Forms

Objectives

- To be able to create reactive forms.

Activity

Complete the section 'Before Each QuickLab' before continuing.

Overview – Parts 1 – 4

In these parts of this QuickLab, you will add the required Modules for working with Reactive forms. You will then create new FormControls for the fields required on the form in the component class. Finally, you will bind the control to the template, using the formControl directive and display the form data on another part of the template.

Part 1 – The ReactiveFormsModule

1. Open 08_ReactiveForms/starter/src/app/app.module.ts for editing.
2. Add `ReactiveFormsModule` to the imports in the `NgModule` decorator, ensuring that it is also imported from @angular/forms at the top of the file.

Part 2 – Generating and Importing a new form control

1. Open src/app/app.component.ts for editing.
2. import the class `FormControl` from @angular/forms.
3. Create a `new FormControl` for *each input element*:
    - `firstname`, supply an *empty string*;
    - `surname`, supply an *empty string*;
    - `age`, supply *null*;
    - `gender`, supply an *empty string*.

```
// as an example
firstname = new FormControl(``);
```

4. Save the file.

Part 3 – Registering the control in the template

1. Open src/app/app.component.html for editing.
2. Add `[formControl]` attributes to each *input* (and the *select*) set to the *name given for it* in the component.

```
// as an example
<input type="text" id="firstname" name="firstname"
```

```
  [formControl]="firstname" >
```

Part 4 – Displaying a form control value

1. In the *section* under the *form* components, add ***data binding*** to *display the value* of each *control*.

```
// as an example
<p>Firstname: {{firstname.value}}</p>
```

2. Save the file and examine the output in the browser.
3. Ensure that each value displayed changes as you change the input for it.

Overview – Parts 5 – 8

In these parts of the QuickLab, you will add a FormGroup to the class and associate this group with the template using the formGroup directive. You will additionally nest a form group inside another in the class and add this group to the form on the template.

Part 5 – Grouping of form controls

1. Open src/app/app.component.ts for editing.
2. Add an import for `FormGroup` to the imports from @angular/forms.
3. In the component class, surround the *four* `formControl` declarations with a `FormGroup` declaration called `userForm`:
    - Make sure that you change the assignment (`=`) to a colon (`:`) for each `FormControl` and the semi-colons (`;`) to commas (`,`):

```
…
  userForm = new FormGroup({
    firstname: new FormControl(''),
    …
  });
…
```

4. Save the file.

Part 6 – Associating the FormGroup model and view

1. Open src/app/app.component.html for editing.
2. Surround the form elements with `<form></form>` (i.e. between the `<hr>` and `<section>`).
3. Add a `[formGroup]` attribute to the form tag with a value of `userForm`.

```
<form [formGroup]="userForm">
```

4. Change the [formControl] attribute to a formControlName for each of the occurrences in the form input elements.

```
…
  <input type="text" id="firstname" name="firstname"
    formControlName="firstname"
  >
…
```

5. Change the *bound values* in the *section* element so that they follow the same pattern as this example for `firstname`:

```
 <p>Firstname: {{userForm.value.firstname}}</p>
```

6. Save the file and return to the browser view.
7. Check that the form receives and displays the values for each form field as expected.

Part 7 – Nesting a form group

1. Open src/app/app.component.ts for editing.
2. In the current form group, after the `surname` declaration, insert a `FormGroup` called `address`.
3. Populate this *form group* with *form controls* for:
    - `houseNo`, set to an *empty string*;
    - `street`, set to an *empty string*;
    - `city`, set to an *empty string*;
    - `postcode`, set to an *empty string*.

```
…
  surname: new FormControl(''),
  address: new FormGroup({
     houseNo: new FormControl(''),
     street: new FormControl(''),
     city: new FormControl(''),
     postcode: new FormControl('')
  }),
…
```

4. Save the file.

Part 8 – Grouping the nested form in the template

1. Open src/app/app.component.html for editing.
2. Insert a new *fieldset* after the *fieldset* for surname, giving it a `formGroupName` attribute of `address`.
3. Add *labels* and *text input elements*, including the `formControlName` attribute for the *four fields* declared in the *address form group*.
4. In the *section* below the *form*, add a *paragraph* to display the *four new form control values*, obtaining their values using the pattern:

```
<p>House Number: {{userForm.value.address.houseNo}}</p>
```

5. Save the file and check that the form accepts and updates values as expected.

Overview – Part 9

In this part of the QuickLab, you will use FormBuilder as an alternative to Controls and Groups.

Part 9 – Using FormBuilder

1. Open src/app/app.component.ts for editing and add an import for `FormBuilder` to the imports from @angular/forms.
2. Add a constructor to the component class that has a `private` property `fb` of type `FormBuilder` as an argument:

```
constructor(private fb: FormBuilder) {}
```

3. edefine the declaration for the `userForm`, swapping the `new FormGroup` declaration for a `FormBuilder` group method:

```
userForm = this.fb.group({
  …
})
```

4. Change each of the `FormControl` declarations for *an array with a single element* that's *value* is the *same* as the *current* `FormControl` argument:

```
  firstname: [''],
```

*NOTE: For the **address** group, replace the **new FormGroup** with another **FormBuilder** group method.*

5. Save the file and check that the form still works as before.

This is the end of Quick Lab 8

Quick Lab 9a - Validating Forms

Objectives

- To be able to apply built-in and custom validators to forms.

Activity

Complete the section 'Before Each QuickLab' before continuing.

Overview – Part 1

In this part of the QuickLab, you will add built-in validators to the form.

Part 1 – Built-in Validators

1. Open 09_FormValidation/starter/src/app/app.component.ts for editing.
2. Add `Validators` to the list of imports from @angular/forms.
3. Add the `Validators` to the controls as prescribed below:

CONTROL	VALIDATORS
FIRSTNAME	Required, minLength of 2
SURNAME	Required, minLength of 2
AGE	Min of 18 and Max of 68

Remember, Validators can be added as a single value:

```
…
    fmCtrl1: ['', Validators.required],
…
```

Or, multiple Validators can be added as an array:

```
…
    fmCtrl2: [null, [Validators.required, Validators.min(2)]],
…
```

4. Save the file and view the form in the browser.

Note that the button is disabled until appropriate values have been included in the validated fields on the form. Try out some combinations of valid and invalid values to satisfy yourself that the validation is being applied.

Overview – Part 2

In this part of the QuickLab, you will create a custom validator that accepts 2 numbers and checks that the supplied number is between the 2 values specified.

Part 2 – Custom Validators

1. In 09_FormValidation/starter/src/app, create a new file called age-range-validator.directive.ts.
2. import `AbstractControl` and `ValidatorFn` from @angular/forms.
3. export a *function* called `ageRangeValidator` that:
    - Takes *2 arguments*, both of type number called `min` and `max`;
    - *Returns* a `ValidatorFn`;

```
export function ageRangeValidator(min: number, max: number):
ValidatorFn {}
```

    - Has a function body that:
        - *Returns* an *arrow function* that:
            - Takes `control` of type AbstractControl as an argument and ValidationErrors (imported from @angular/forms) as a return type;

```
…
   (control: AbstractControl): ValidationErrors =>
   {}
…
```

            - Has a function body that sets forbidden to be the output of the following conditions:
                1. `control.value` does NOT have a value OR `control.value` is *Not a Number*
                2. OR `control.value` is *less than* `min`
                3. OR `control.value` is *greater than* `max`.

```
…
        const forbidden  = ((isNaN(control.value) ||
           !control.value) || control.value < min ||
           control.value > max);
…
```

                4. Returning `forbidden` as an *object* with a key of `ageRange` and a value of an *object*

with key of `value` and a value of `control.value` when true of `null` when false.

```
…
        return forbidden ? { 'ageRange' : {value:
control.value} }
           : null;
…
```

4. Save the file and open app.component.ts for editing.
5. Import the custom validator function.

```
import { ageRangeValidator } from './age-range-
validator.directive';
```

6. Replace the validator for age with the new custom validator, passing arguments that mean the age must be between 18 and 68.

```
…
    age: [null, ageRangeValidator(18, 68)],
…
```

7. Save the file.

Return to the browser and check that the button is disabled when the validated fields (including the new age range validator) are invalid.

If you have time:

Add a validation message when the age does not meet the validation. You may want to:

1. Add a div the uses a condition to only display when the field's `touched` and `invalid` properties are `true`;
    - Hint: use `userForm.get('age')` to access the field's properties;
2. Add a paragraph that only displays if the `ageRange` error exists;
3. Add a CSS rule to style a paragraph that is a descendent of an element that has a class of `ng-invalid` applied to it so that the text is coloured `red`.

Optionally, add the status of the validity of the age to the display below the form.

This is the end of Quick Lab 9a

Quick Lab 9b - Testing Reactive Form Validation

Objectives

- To be able to test field validation and form validity.
- To be able to write tests for a custom validator.

Overview

In this QuickLab, you will test the validation on a Reactive Form. This will be done by firstly setting up a test environment for a Reactive Form. The action of the built-in validators used on the form will then be checked to ensure that they produce the expected behaviour. After each field check, the conditions for overall form validity will be tested. The final part of the QuickLab looks at how the custom validator can be tested.

Activity

Skip the 'Before Each QuickLab' for this Activity.

Default tests

1. From the command line, run the test suite using:

```
ng test --code-coverage
```

Notice that there are **3 failures**.

These errors are due to the standard tests that the CLI writes when creating `AppComponent`.

2. Open **app.component.spec.ts** for editing from **09_FormValidation/starter/src**.
3. Add `ReactiveFormsModule` to the **imports** in the `TestBed.configureTestingModule` object.

```
beforeEach(waitForAsync(() => {
        TestBed.configureTestingModule({
            declarations: [
                AppComponent
            ],
            imports: [ReactiveFormsModule] // add me!
        }).compileComponents();
    }));
```

4. *Delete* the other 2 `it` specs (for title and render h1).
5. Save the file and observe the test output again – 1 test, 1 pass!

Set Up for All tests

1. Immediately under the `describe` function declaration, add *2 variables* to the body of the arrow function, before the `waitForAsync beforeEach`:
    - `fixture` of type ComponentFixture<AppComponent>;

        imported from ***@angular/core/testing***.
    - `app` of type AppComponent;
2. Under the existing `waitForAsync beforeEach` function, add a *synchronous* `beforeEach` that:
    - Sets `fixture` to be a *call* to `TestBed.createComponent(AppComponent)`;
    - Sets `app` to be `fixture.debugElement.componentInstance`;
    - Calls `fixture.detectChanges()`;

```
beforeEach(() => {
        fixture = TestBed.createComponent(AppComponent);
        app = fixture.debugElement.componentInstance;
        fixture.detectChanges();
    });
```

3. *Remove* the *declarations* for `fixture` and `app` in the current `it` spec.

Testing the field validators

1. Under the *existing spec*, add a test suite called `` `Test the validity of the required fields` ``.

```
describe(`Test the validity of the required fields`, () => {

});
```

2. *Nest* a *suite* inside this called `` `firstname field:` ``.

```
describe(`Test the validity of the required fields`, () => {
  describe(`firstname field:`, () => {

  });
});
```

3. Add a *spec* to this suite called `should be initially invalid and have required error`.

```
describe(`Test the validity of the required fields`, () => {
  describe(`firstname field:`, () => {
    it(`should be initially invalid and have required error`,
() => {

    });
  });
});
```

4. Inside the arrow function for this spec:
    - Set a `const` called `firstname` to `app.userform.get('firstname')`;
    - `expect` `firstname.valid` *to be* `false`;
    - `expect` `firstname.errors['required']` *to be* `true`.

```
describe(`Test the validity of the required fields`, () => {
  describe(`firstname field:`, () => {
    it(`should be initially invalid and have required error`,
() => {
      const firstname = app.userForm.get('firstname');
      expect(firstname.valid).toBe(false);
      expect(firstname.errors['required']).toBe(true);
    });
  });
});
```

5. Add a second *spec* inside this *suite* called `should be valid if value has 2 or more characters`.
6. Inside the arrow function for this spec:
    - Set a `const` called `firstname` to `app.userform.get('firstname')`;
    - Calls `setValue` on `firstname` with the value `` `A` ``;
    - `expect` `firstname.valid` *to be* `false`;
    - `expect` `firstname.hasError('minlength')` *to be* `true`;
    - Calls `setValue` on `firstname` with the value `` `Ab` ``;
    - `expect` `firstname.valid` *to be* `true`.
7. Save file and observe the test output – do all your tests pass? If a test fails can you identify why and fix it?

These tests could be repeated for the surname field.

Testing form validity

The *form* should report itself as valid when *all required fields* have a valid state.

1. Under the *suite* that Tests the validity of the required fields, add a further *suite* with the name `Test form validity`.
2. Add a *spec* inside the suite's arrow function called `*should be valid when required fields are valid*`.
3. Inside its arrow function:
    - `expect app.userForm.valid` *to be* `false`;
    - Call `setValue` on `app.userForm.get('firstname')` with `` `TestFirstName` ``;
    - Call `setValue` on `app.userForm.get('surname')` with `` `TestSurname` ``;
    - Call `setValue` on `app.userForm.get('age')` with `30`.
    - `expect app.userForm.valid` *to be* `true`.
4. Save the file and check that the new specs pass.

Testing the Custom Validator function

Inspection of the code-coverage report shows that the age-range-validator directive is apparently tested. However, we are relying on the custom validator working as we intended and that it reports as valid and invalid as we intended it to. There are no tests for the function. This can be done in straight Jasmine code - no need for any Angular TestBed or components, etc.

1. Create a file called age-range-validator.directive.spec.ts in the same folder as the actual directive.
2. Import the `ageRangeValidator` function from the *file that contains it*.
3. Import `AbstractControl` from @angular/forms.
4. Create a *test suite* called `` `min and max age range validator tests` ``.
5. Inside the arrow function:
    - Add a `const` called `ageRangeValidatorFunction` and set this to a *call* to `ageRangeValidator` with `10` and `20` as the arguments;
    - Add a `const` called `control` of type `{ value: any }` *assigned* to `{ value: '' }`;
    - Declare a variable called `result` set to type ValidationErrors imported from @angular/forms.
6. Add a *spec* to the *suite* with the title `` `should return an object if an empty string or alphanumeric string is passed` ``.

7. Inside its arrow function:
    - Set `result` to be a *call* to `ageRangeValidatorFunction` with an argument of `control as AbstractControl`;

```
...
    result = ageRangeValidatorFunction(control as
      AbstractControl);
...
```

    - `expect result` *to equal* an *object* with a **key** of `'ageRange'` and a **value** of an *object* with a **key** of `value` and a **value** of `control.value`;
    - Set `control`'s `value` property to `` `Arbitrary String 1234` ``;
    - Assign `result` to a *call* to `ageRangeValidatorFunction` with an argument of `control as AbstractControl`;
    - `expect result` *to equal* an *object* with a **key** of `'ageRange'` and a **value** of an *object* with a **key** of `value` and a **value** of `control.value`;
8. Save the file and check that the new spec passes.
9. Add another *spec* to the *suite* called `` `should return an object if value of 9 or less OR 21 or higher is supplied (as a string or number)` ``.
10. Inside its arrow function:
    - Set `control`'s `value` property to `'9'`;
    - Set `result` to be a *call* to `ageRangeValidatorFunction` with `control as AbstractControl`;
    - `expect result` *to equal* an *object* with a **key** of `'ageRange'` and a **value** of an *object* with a **key** of `value` and a **value** of `control.value`;
    - Assign `control`'s `value` property to be `21`;
    - Assign `result` to be a *call* to `ageRangeValidatorFunction` with an argument of `control as AbstractControl`;
    - `expect result` *to equal* an *object* with a **key** of `'ageRange'` and a **value** of an *object* with a **key** of `value` and a **value** of `control.value`;
11. Save the file and check that the new spec works.
12. Add a final *spec* called `` `should return null if value of 10 or more AND less than 21 is supplied (as string or number)` ``.
13. Inside its arrow function:
    - Set `control`'s `value` property to be `'10'`;

- Set `result` to be a *call* to `ageRangeValidatorFunction` with an argument of `control as AbstractControl`;
- `expect result` *to be* `null`;
- Assign `control`'s `value` property to be `20`;
- Assign `result` to be a *call* to `ageRangeValidatorFunction` with an argument of `control as AbstractControl`;
- `expect result` *to be* `null`;

14. Save the file and check that this final spec passes. Note that the code coverage report is now at 100%.

This is the end of Quick Lab 9b

Quick Lab 10a - Creating, Testing and Consuming Services

Objectives

- To be able to create a Service in Angular using BDD
- To be able to retrieve data from a Service
- To be able to submit data to a Service

Activity

Complete the section 'Before Each QuickLab' before continuing.

Before continuing with this QuickLab, take a few minutes to observe the application in the browser and the code for the 2 components that have been provided (list-users and add-user). Also notice that the users folder has a .model file to provide details for a User class and a .data file that will provide an initial list of users for the application.

Once you are happy with the way the application is set up, proceed with the instructions below.

Overview – Parts 1 – 5

In these parts of the QuickLab, you will write tests for and then write code for a service. Initially, you will create a service stub using the Angular CLI. Following this, using BDD/TDD principals, you will write tests to check that required service functionality works. Once the tests are written, you will go about adding the code. You will write code to deal with 'users' being able to get all of the users and add a new user.

Part 1 – Creating the Service

The Service you are about to create will provide an *array of users* and also *allow a user to be added* to this array.

1. Open the terminal window and navigate to the users folder i.e. 10_Services/10a/starter/src/app/users.
2. Use the Angular CLI to create a Service called user

```
ng g service user
```

Inspect the file user.service.ts and satisfy yourself that it will be provided by the root injector.

The service is to return an array with object of type User in it and take a User that is passed to the service and add it to the stored array. Write the tests to ensure that this happens before writing the actual code to perform these actions!

Part 2 – getUsers() tests

1. Open user.service.spec.ts for editing and import the `User` class from user.model and the `users as expectedUsers` from user.data.
2. In the *suite*, add a variable of `service` of type UserService.
3. Add a `const` of `newUser` of type `User` with keys `firstname` (of type string), `surname` (string), `age` (number) and `gender` (string) with *any values you choose*.
4. Modify the `beforeEach` so that it *sets* `service` to be `TestBed.inject(UserService)`.
5. Remove the declaration of `service` from the *first spec*.
6. Write a *spec* with a title `` `getUsers() should return an array of the 4 expected users` ``.
7. Set *expectations* that `service.getUsers()` *equals* `expectedUsers` and that the `length` of the *returned array to be* `4`.
8. Save the file and run the tests with `ng test --code-coverage`, the spec should produce a fail.

Part 3 – Writing the getUsers() code

1. In the user.service.ts file, import the `User` class from user.model and `users` (aliasing it as `externalUserData`) from users.data:

```
…
import { User } from './user.model';
import { users as externalUserData } from './users.data';
…
```

2. In the class itself, declare a *property* of `users` that is an array of User and initialise it to be `externalUserData`.
3. Add a *method* called `getUsers` to the class that *should return* an array of User - and add the return statement to `return this.users`.
4. Save the file and check that the spec now passes.

Part 4 – Writing the addUser() tests

1. Open user.service.spec.ts for editing.
2. Add a *second spec* to the *suite* with the title `` `addUser() should add the newUser to the users array` ``.
3. Inside the arrow function's body:
    - Call `service.addUser(newUser)`;

- Set *expectations* that the *call* to `getUsers` will *return* `expectedUsers` and that the `length` of the *returned array to be* `5`.

4. Save the file and check that this new spec fails.

Part 5 – Writing the addUser() code

1. Open user.service.ts for editing.
2. Add a *second method* called `addUser` that takes a `user` of type User as an argument and has *no return value*.
3. Make the body of the method:
    - Declare a `const` `users` that is *set to* the *current value* of `users` from the class;

```
const users = this.users;
```

    - *Pushes* the new user to the `users` array;
    - Sets the *class value* of `users` to the *value* of `users` from this function.

```
this.users = users;
```

4. Save the file and check that the test specs all pass.
5. Save the file and make sure that the last failing spec now passes.

Overview – Parts 6 – 9

In these parts of the QuickLab, you will use the Service within the application to retrieve and submit data in components. Firstly, you will test and implement the use of the getUsers function from the service in the ListUsersComponent. Finally, you will test and implement the use of the addUser function from the service in the AddUserComponent, taking data from the form and submitting it.

Part 6 – Testing getUsers() in ListUsersComponent

1. Open list-users.component.spec.ts for editing.
2. After the imports add a class called `MockUserService`.
3. Add a *function* called `getUsers` that *returns an array of User objects* - set `firstname`, `surname`, `age` and `gender` to *any values* of your choosing.
4. Add a *declaration* of `userService` of type UserService to the suite (under fixture).

```
describe('ListUsersComponent', () => {
  let component: ListUsersComponent;
  let fixture: ComponentFixture<ListUsersComponent>;
```

```
let userService: UserService; // add me!

beforeEach(waitForAsync(() => {…
```

5. Modify the `TestBed.configureTestingModule` object so that it has
    - An **additional key** of **providers** set to *an array* that contains
        - An *object* with a **key** of `provide` set to `UserService` and a **key** `useClass` set to `MockUserService`.

```
TestBed.configureTestingModule({
  declarations: [ListUsersComponent],
  providers: [
    { provide: UserService, useClass: MockUserService }
  ]
})
```

6. In the *synchronous* `beforeEach`, before `fixure.detectChanges()` set `userService` to a *call* to `TestBed.inject(UserService)`.
7. Add an `it` *spec* to the *suite* with the title **`should make a call to the service's getUsers method`**.
8. The arrow function's body should:
    - Set `usersService.getUsers` to be `jasmine.createSpy(`getUsers spy`)`;
    - Calls `component.ngOnInit()`;
    - `expect` `userService.getUsers` *to have been called*.
9. Add a second `it` *spec* called **`should set users in the component to the return of the service`**.
10. The arrow function's body should:
    - Set a `const` `expectedUsers` to be the *same array* as you *declared* for the *return* of `getUsers` in the mock service;
    - Call `component.ngOnInit()`;
    - `expect` `component.users` *to equal* `expectedUsers`.
11. Save the file and all the new specs should fail.

Part 7 – Using getUsers() in ListUsersComponent

1. Open **list-users.component.ts** for editing.
2. Inject the `UserService` into the constructor of this component (ensuring that it is imported at the top of the file):

```
…
    constructor(private userService: UserService) {}
…
```

3. Edit the *body* of the `ngOnInit` function by making a *call* to the `userService`'s `getUsers` method, setting `this.users` to be the *value returned*.

```
  ngOnInit() {
    this.users = this.userService.getUsers();
  }
```

4. Save the file and observe the output in the browser.

You should see that the table of users is now populated with the values supplied by the Service. Inspection of the tests should reveal that all of the specs now pass.

Part 8 – Testing addUsers calls to the service from AddUserComponent

1. Open **add-user.component.spec.ts** for editing.
2. Add a *class* called `MockUserService` that has a *method* called `addUser` that takes an *argument* of `user` and simply *returns* `null`.

This method does not need to do anything - we will simply test that it is called.

3. Add a **declaration** of `userService` of type **UserService** to the *suite*.
4. Modify the `TestBed.configureTestingModule` *object* so that it has:
    - An additional **key** of `providers` set to an **array** that contains:
        - `AddUserComponent`
        - An object with a **key** of `provide` set to `UserService` and a **key** `useClass` set to `MockUserService`.
5. In the *synchronous* `beforeEach`, **before** `fixure.detectChanges()` set `userService` to a *call* to `TestBed.inject(UserService)`.
6. Nest a *suite* called **`Form submission tests`** and declare:
    - A `const` `firstname` set to a **string** of your choice;
    - A `const` `surname` set to a **string** of your choice;
    - A `const` `age` set to a **number** of your choice;
    - A `const` `gender` set to a **string** of your choice;
    - A `const` `expectedUser` as an *object* with **key-value** pairs of the **constants** above.

```
…
    const firstname = `Test2`;
    const surname = `Test2`;
    const age = 30;
    const gender = `male`;
    const expectedUser = { firstname, surname, age, gender };
…
```

7. Create a *synchronous* `beforeEach` function that:
    - Sets *each of the 4 form values* using the *pattern*:

```
component.userForm.get('fieldName').setValue(fieldName);
```

    - Calls `fixture.detectChanges()`;
8. Write an `it` *spec* with a title of `` `should have a form value of the supplied user` ``.
9. The arrow function body should:
    - `expect` `component.userForm.value` *to equal* `expectedUser`.
10. Write another `it` *spec* with a title of `` `should call addUser with the expectedUser` ``.
11. The arrow function body should:
    - Set `userService.addUser` to `jasmine.createSpy(`addUser spy`)`;
    - Call `component.onSubmit()`;
    - `expect` `userService.addUser` *to have been called* `1` *times*;
    - `expect` `userService.addUser` *to have been called with* `expectedUser`.
12. Save the file and check that the specs fail.

Part 9 – Using the Service to add data from the form

1. Open add-user.component.ts.
2. *Inject* the `UserService` into the constructor of this component (***ensuring that it is imported at the top of the file***):

```
…
    constructor(private userService: UserService) {}
…
```

3. Edit the `onSubmit()` function so that it:

- Has a *return type* of void;
- Sets a `const` of `user` (type User) that is *set to the value of the form group*:

```
…
    const user: User = this.userForm.value;
…
```

- Make a call to the `addUser` method in the `userService`, passing in the `user`.

4. Save the file.
5. Observe the browser and ***add valid data*** to the form.
6. Click Submit and see that the user is added to the list of users displayed above the form.
7. Check that the specs have also passed.
8. Refresh the application in the browser, note that the data entered disappears…

This is the end of Quick Lab 10a

Quick Lab 10b - Using Services with HTTP

Objectives

- To be able to make HTTP requests to a REST API with Services.

Activity

Complete the section 'Before Each QuickLab' before continuing.

You are going to change the methods provided in the Service to use HTTP requests to a REST API provided by JSON-server.

Part 1 – Install and run JSON Server

1. Open a terminal/command line, *additional to that running the Angular application*, and enter:

```
npm i -g json-server
```

This installs JSON server globally on your computer - you don't need to install it each time you want to use it in a project.

The data file for this QuickLab has been provided as data.json in the root folder for 10_Services/10b/starter.

2. Still on the command prompt, start JSON server, pointing at this file by navigating to the folder described above and entering the command:

```
json-server --watch data.json
```

3. Check that the server is running by navigating to the URL below and checking that adding /2 to the end of the URL shows the data for the user with id of 2:

http://localhost:3000/users

Overview – Parts 2 – 4

In these parts of the QuickLab, you will set up the service so that it can use HTTP capability to retrieve data from a RESTful end point. This requires the use of the HttpClientModule and an instance of HttpClient injected into the service. The service will then be modified to obtain the data from the HTTP call and return an Observable as the data will be asynchronous. Finally, you will modify the component that calls the getUsers service to subscribe to the Observable and use the data.

Part 2 – Import HTTP capability into the Application and inject into the Service

1. Open app.module.ts.
2. Add `HttpClientModule` to the list of imports in the metadata - *ensure*

that it is also imported from @angular/common/http.

3. Save the file.
4. Open /src/app/users/user.service.ts.
5. *Inject* an instance of `HttpClient`, (*ensuring that it is imported from @angular/common/http*) into the constructor of the service:

```
…
  constructor(private http: HttpClient) {}
…
```

6. Save the file.

Part 3 – Modify the Service to return the Users array with data from the REST API

1. In user.service.ts, add a `readonly` property to the class called `USERSURL` and set it to the string `` `http://localhost:3000/users` ``.
2. Modify `getUsers()` so it should return an `Observable` of type User[].
3. Make `getUsers()` *return* the *result* of a GET (typed as <User[]>) request to `this.USERSURL`:

```
  getUsers(): Observable<User[]> {
    return this.http.get<User[]>(this.USERSURL);
  }
```

4. Add an import for `Observable` from rxjs at the top of the file.
5. Save the file.

Part 4 – Modify the list-users component to subscribe to the returned Observable

1. Open list-user.component.ts.
2. Modify `ngOnInit` so it subscribes to the Observable returned by the call to `getUsers`:
    - The data returned should *update the value* of the `users` array in the class.

```
  ngOnInit() {
    this.userService.getUsers().subscribe(data => {
      this.users = data;
    });
  }
```

3. Save the file.
4. Observe the browser and ensure that the data is displayed in the table when the app is refreshed.

Overview – Parts 5 – 6

In these parts of the QuickLab, you will send data to the REST API using HTTP calls. This involves making POST requests from the service after receiving data from a call in the component. The component will need to subscribe to the Observable attached to the request to initiate it.

Part 5 – Modify the Service to send data to the REST API when a new user is submitted

1. Open user.service.ts.
2. Change `addUser()` so that it:
    - Returns a POST request to `this.USERSURL`, supplying the `user` passed in as the body of the request.

```
…
  addUser(user): Observable<User> {
    return this.http.post<User>(this.USERSURL, user);
  }
…
```

3. Save the file.

Part 6 – Modify the add-user component to subscribe to the Observable

1. Open add-user.component.ts.

2. Modify `onSubmit` so that it subscribes to the `addUser` observable rather than making a direct call to it:

```
…
  onSubmit() {
    const user:User = this.userForm.value;
    this.userService.addUser(user).subscribe();
  }
…
```

4. Save the file.
5. Return to the browser and try adding a user. Refresh the app and you should see that the new data is added, updating the table with the new

value.

6. Check the data.json file - you should see that the new user is permanently in this file too.

Overview – Checking for Errors - Parts 7 – 8

In these parts of the QuickLab, you will ensure that the application handles errors in the Observable gracefully. You will use the HttpErrorResponse object to supply the error data. This will then be used within the component to display a message to the user via the template.

Part 7 – Checking for errors when getting user data

1. Open list-users.component.ts.
2. Declare `public` class variables of:
   - `error` set to be `false`;
   - `errorText` of type string.
3. Modify the subscription to the `getUsers` observable by adding an *error callback* that:
   - Takes `err` of type HttpErrorResponse as an argument;
   - Sets `error` to `true`;
   - Checks to see if `err` is an `instanceof Error` and set `errorText` to `` `An error occurred in the application.` `` as its body;
   - Else it should set `errorText` to `` `A server error occurred.` ``;
4. Save the file.

Part 8 – Make the template display the table or an error

1. Open list-user.component.html.
2. Add `*ngIf` to the *table* tag that is set to `!error`.
3. Under the *table*'s closing tag add a *paragraph* with an `*ngIf` set to `error` and content of `errorText`.
4. Save the file.
5. Stop JSON Server running by opening its terminal and pressing CTRL+C.
6. Refresh the page and observe `errorText` being displayed where the *table* was.

Bonus Challenge - can you make the add-user component display an error message if the POST request fails?

This is the end of Quick Lab 10b

Quick Lab 10c - Testing Services with HTTP

Objectives

- To be able to test services with HTTP requests to a REST API with Services.

Activity

Skip the 'Before Each QuickLab' for this Activity.

Overview – Part 1

In this part of the QuickLab, you will test that the service makes the correct HTTP calls and returns the correct data. As part of this, you will set the tests up correctly and then check that the service methods are called as expected and that it returns the correct data. Following this, you will test that the service responds correctly when the REST endpoint is not available.

Part 1 – Testing the service makes the correct calls and returns the correct data

Test Set Up

1. Open src/app/users/user.service.spec.ts for editing.
2. Remove the existing *suite*.
3. Add another *suite* with the title `Testing the service makes correct calls and gets correct data`.
4. Add a variable called `httpClientSpy` of type *object* with keys `get` and `post` and values `jasmine.Spy`.

```
let httpClientSpy: { get: jasmine.Spy, post: jasmine.Spy }
```

5. Add a second variable called `userService` of type `UserService`.
6. Create a `beforeEach` function that:
    - Sets `httpClientSpy` to be a *call* to `jasmine.createSpyObj` with arguments of `'HttpClient'` and `['get', 'post]`;
    - Sets `userService` to me a `new UserService`, passing in `httpClientSpy` with an `<any>` assertion.

```
beforeEach(() => {
    httpClientSpy =
      jasmine.createSpyObj('HttpClient', ['get', 'post']);
    userService = new UserService(<any>httpClientSpy);
  });
```

Testing the service methods get called correctly

Testing getUsers()

1. Add an **import** of `{ asyncData }` from '../../testing/async-observable-helpers' at the top of the file.
2. Add an **import** of `{ users as expectedUsers }` from './users.data'.
3. Create an `it` **spec** with the title `` `should return the expected data when get is called (once and only once)` ``, with an arrow function body that:
    - Makes the `httpClientSpy` *call* `get` and *return* the value `expectedData` that has been *passed into* the `asyncData` function.

```
httpClientSpy.get.and.returnValue(asyncData(expectedUsers));
```

    - Call `getUsers` on `userService` **subscribing** to its observable:
        - `values` passed into the *'next' arrow function* should *return* an *expectation* that the *values equal* the `expectedUsers`;
        - `fail` should be the second argument to subscribe:

```
…
    userService.getUsers().subscribe(
      values => expect(values).toEqual(expectedUsers,
`expected users`),
      fail
    );
…
```

    - `expect` `httpClientSpy.get` *to have been called* `1` *times*:

```
    expect(httpClientSpy.get).toHaveBeenCalledTimes(1);
```

4. Save all of the files and run the tests using ng test.

You should find that all of the tests pass, proving the service makes a single call to the `getUsers` method and that the data returned is as expected.

Testing addUsers()

1. Add another `it` *spec* in this suite called `` `should return an object when addUser is called, calling with the supplied object - once and only once` ``.
2. The arrow function body should:
    - Make the `httpClientSpy` call `post` and *return the value* `expectedData` that has been *passed into* the `asyncData` function.

```
httpClientSpy.post.and.returnValue(asyncData(newUser));
```

- Call `addUser` on `userService`, *passing in* `newUser` and subscribing to its observable:
  - `value` passed into the *'next' arrow function* should *return* an *expectation* that the `value` *equal* `newUser`;
  - `fail` should be the second argument to subscribe;
- expect `httpClientSpy.post` to have been called one time.

3. Save all files and check that the new test passes.

Test Errors

1. Add an `it` *spec* with the title: `` `should return an error when the server returns a 404` ``.
2. The arrow function body should:
   - Declare a `const` called `errorResponse`, setting it to be a `new HttpErrorResponse` and *passing in* an *object* with:
     - `error` set to `` `test 404 error` ``;
     - `status` set to `404`;
     - `statusText` set to `` `Not Found` ``.
   - Make the `httpClientSpy` *call* `get` and *return the value* `errorResponse` that has been passed into the `asyncError` function.

```
httpClientSpy.get.and.returnValue(asyncError(errorResponse));
```

   - Call `getUsers` on `userService` subscribing to its observable:
     - `values` passed into the *'next' arrow function* `fail` with an argument of `` `expected an error, not data` ``;
     - `error` should be passed into the *'error' arrow function* and `expect` `error.error` *to contain* `` `test 404 error` ``.
   - Make the `httpClientSpy` *call* `post` and *return the value* `errorResponse` that has been passed into the `asyncError` function.

```
httpClientSpy.post.and.returnValue(asyncError(errorResponse));
```

   - Call `addUser` on `userService`, passing in `newUser` and subscribing to its observable:
     - `values` passed into the *'next' arrow function* `fail` with an argument of `` `expected an error, not data` ``;

- `error` should be passed into the *'error' arrow function* and `expect` `error.error` to contain `` `test 404 error` ``.

3. Save and check that the new tests pass.

These tests have verified that the service returns the correct data when its functions are called and that they can handle an error. However, we need to check the actual calls made to the service by mocking the HTTP requests and checking that the correct URLs are being used.

Overview – Part 2

In this part of the QuickLab, you will use a mocking strategy to test that the service uses the correct location in HTTP requests. This requires mocking the HttpClient using the HttpClientTestingModule to be able to access the actual location that the requests are made to.

Part 2 – Mocking the HttpClient calls for testing

1. Under the *previous test suite*, create a new one with the title `` `Mocking the HttpClient calls - ensuring the correct locations are activated` ``.
2. Declare 3 variables as follows:
    - `httpClient` of type HttpClient;
    - `httpTestingController` of type HttpTestingController (*imported from @angular/common/http/testing*);
    - `userService` of type UserService.
3. Add a `beforeEach` function that:
    - Sets the `TestBed.configureTestingModule` with:
        - `imports` of `[HttpClientTestingModule]`, (*imported from @angular/common/http/testing*);
        - `providers` of `[UserService]`;
    - Sets `httpClient` to a *call* to `get` on `TestBed` with `HttpClient` as an argument;
    - Sets `httpTestingController` to a *call* to `get` on `TestBed` with `HttpTestingController` as an argument;
    - Sets `userService` to a *call* to `get` on `TestBed` with `UserService` as an argument.
4. Add an `afterEach` function that calls `verify()` on `httpTestingController`.
5. Create an `it` *spec* titled `` ` should return an error when a bad url is requested - 404 error` `` with an arrow function body that:
    - Declares a `const` `errmsg` set to `` `generated 404 error` ``;

- Declares a `const` `testUrl` set to `` `getMeA404` ``;
- Makes a `get<any[]>` call on `httpClient`, passing in `testUrl` and subscribing to its observable:
    - `data` should be passed into the *'next' arrow function*, calling `fail` with `` `fail with 404 error` `` as an argument;
    - `error` of type HttpErrorResponse should be passed into the 'error' arrow function, executing:
        - `expect` `error.status` *to equal* `404`;
        - `expect` `error.error` *to equal* `errmsg`.
- Declares a `const` `req` and sets it to a *call* to `expectOne` on `httpTestingController`, passing in `testUrl`.
- `expect` `req.request.url` *not to be* `userService.USERSURL`.
- Call `flush` on `req`, passing in `errmsg` and an *object* with keys of `status` and `statusText` and *values* of `404` and `` `Not Found` `` respectively.

```
it(`should return an error when a bad url is requested - 404
error`, () => {

  const errmsg = `generated 404 error`;
  const testUrl = `getMeA404`;

  httpClient.get<any[]>(testUrl).subscribe(
    data => fail(`fail with 404 error`),
    (error: HttpErrorResponse) => {
      expect(error.status).toEqual(404, `status`);
      expect(error.error).toEqual(errmsg, `message`);
    }
  );

  const req = httpTestingController.expectOne(testUrl);
  req.flush(errmsg, { status: 404, statusText: `Not Found` });
});
```

6. Save the file and check that the new tests pass.

If you have time...

Write a test that returns expectedUsers when a call to get on httpClient is made with a testUrl set to `http://localhost:3000/users` and verifies that the `req.request.url` is the same as the `USERSURL` set in the UserService. Try to check the request method is correct.

Write a test for `getUsers` and `addUser`, expecting data to be returned from the observable as expected and that a request is made to the correct URL once for each method.

This is the end of Quick Lab 10c

Quick Lab 11a - Setting up an application with routing

Objectives

- To be able to use the Angular CLI to set up an application with routing enabled as default.

Overview

In this QuickLab, you will use the CLI and Routing options to set up a project that has a root router module predefined and integrated into the project. You will then examine the effect this has on the provided component.

Activity

There is no need to complete the section 'Before Each QuickLab' before continuing.

1. Open VSCode's in-built console by selecting View - Integrated Terminal (or use the shortcut key or icon on the bottom bar).
2. Ensure that the terminal is pointing to the QuickLabs/11_Routing/starters folder and create a new application using the command:

```
ng new QL11a
```

3. Choose `y` for the 'strict typing' option.
4. At the prompt below, type `y` and press `Enter`:

```
? Would you like to add Angular routing? (y/N)
```

5. Choose `CSS` as the styling and run the installation.
6. Once the installation is complete, check the app folder for the file app-routing.module.ts and that it is imported into app.module.ts. Also check that `<router-outlet></router-outlet>` has been added to app.component.html.

This is the end of Quick Lab 11a

Quick Lab 11b - Adding and Testing the Routes array

Objectives

- To be able to create a routes array using various options

Overview

In this QuickLab, you will generate some components to use for display on different routes. You will then define the Route for each component in the routing module. Finally, you will go through the process of testing the routing works correctly, rendering the correct component.

Activity

There is no need to complete the section 'Before Each QuickLab' before continuing.

1. Point the integrated terminal to QuickLabs/11_Routing/starters/QL11a.
2. Use the CLI to generate components with the following names:
    - `page1`;
    - `page2`;
    - `page-not-found`.
3. Open page2.component.ts for editing and add a property `title` to the class, initialised as `'default'` and save the file.
4. Open page2.component.html for editing and replace the *text* with *data-binding* to show title and save the file.
5. Open app-routing.module.ts for editing and add export to the declaration of the routes array

```
export const routes: Routes = [];
```

6. Add an *object* to the `routes` array that has:
    - A key `path` set to `'page1'`;
    - A key `component` set to `Page1Component`.
7. Add a second *object* to the array that has:
    - A key `path` set to `'page2'`;
    - A key `component` set to `Page2Component`;
    - A key `data` set to an *object* that has a key `title` and a value of `'Page 2'`.
8. Add a third *object* to the array that has:

- A key `path` set to `''`;
- A key `redirectTo` set to `'/page1'`;
- A key `pathMatch` set to `'full'`.

9. Add a fourth *object* to the array that has:
    - A key `path` set to `'**'`;
    - A key `component` set to `PageNotFoundComponent`.
10. Ensure that the 3 components have been added to the list of items imported into this file.
11. Save the file and serve the application.

page1 works show be displayed at the bottom of the page (as it is the / route!). Change the end of the address in the browser address bar to /page1, /page2 and /notaroute and see the effect working as intended.
Note: The title data supplied will not be displayed as part of the component view yet!

Testing the Routes

1. Open app.component.spec.ts for editing.
2. Add 3 declarations to the existing *suite* for:
    - `location` of type Location, *imported from @angular/common*;
    - `router` of type Router, *imported from @angular/router*;
    - `fixture` of type ComponentFixture<AppComponent>, *imported from @angular/core/testing*.
3. In the `TestBed` config *object*, add `.withRoutes(routes)` to the import of `RouterTestingModule`, importing `routes` from './app-routing.module'.
4. Add `Page1Component`, `Page2Component` and `PageNotFoundComponent` to the list of declarations.
5. Under the configuration code, create a synchronous `beforeEach` and set:
    - `router` to be a *call* to `inject` on `TestBed` with `Router` as an argument;
    - `location` to be a *call* to `inject` on `TestBed` with `Location` as an argument;
    - `fixture` to be a *call* to `createComponent` on `TestBed` with `AppComponent` as an argument.
6. Under the current `it` *specs*, add another *spec* with a title of `navigate to "" redirects you to /page1`.
7. The arrow function for this `it` spec should be wrapped in a `fakeAsync` call, *imported from @angular/core/testing*.

```
it(`navigate to "" redirects you to /page1`, fakeAsync(() => {
}));
```

8. The arrow function body should:
    - *Call* `navigate` on `router` with an argument of `['']`;
    - `tick()`;
    - `expect` `location.path()` *to be* `'/page1'`.
9. Save the file and check that the test passes. If you have time, write further tests to check that the routes for '/page1' and '/page2' work correctly.

This is the end of Quick Lab 11b

Quick Lab 11c - Using ActivatedRoute

Objectives

- To be able to use ActivatedRoute to obtain the data object from a Route

Overview

In this QuickLab, you will use the Activated Route object in a component to be able to access data passed to it from the router module. Data is available as an Observable, so this will be subscribed to before using the data in the component. In this instance, you will provide a title for a component via the route object's data key.

Activity

Continue working in the folder QuickLabs/11_Routing/starters/QL11a.

1. Open page2.component.ts for editing and add a *class property* `sub` of type any.
2. *Inject* a `private` `ActivatedRoute` called `route` into the constructor:

```
…
    constructor(private route: ActivatedRoute) {}
…
```

   - Ensure that `ActivatedRoute` is added to the imports at the top of the file. It should come from @angular/router.

3. In the ngOnInit method body:
   - Set `sub` to be the result of the observable route chaining:
     - `data`
     - a *call* to `subscribe` that takes a `value` and sets `title` to `value.title`.

```
    ngOnInit() {
        this.sub = this.route
            .data
            .subscribe(value => this.title = value.title);
    }
```

4. Save the file and return to the browser.

Activate the route /page2 and observe that the title now matches the data supplied in the route. i.e. it is now Page 2 rather than default. Comment out the ngOnInit function to check if it's not obvious!

It is worth mentioning that only static data should be supplied in this way.

This is the end of Quick Lab 11c

Quick Lab 11d - RouterLinks

Objectives

- To be able to add links to activate routes in the template

Overview

In this QuickLab, you will add some routes to the template to act as a navigation bar. This will use the routerLink directive along with routerLinkActive for setting styles on active links.

Activity

Continue working in the folder QuickLabs/11_Routing/starters/QL11a.

1. Open app.component.html for editing and *remove all* of the *markup* apart from the `<router-outlet></router-outlet>`.
2. Above the `<router-outlet>` add:
    - `<nav>` containing a `<ul>` with `<li>`s containing:
        - `<a>` with attributes `routerLink` set to `"/"`, `routerLinkActive` set to `active` and *text* of `Home`;
        - `<a>` with attributes `routerLink` set to `"/page1"`, `routerLinkActive` set to `active bordered` and *text* of `Page 1`;
        - `<a>` with attributes `routerLink` set to `"/page2"`, `routerLinkActive` *bound to* `activeClasses` and *text* of `Page 2`;

```
<a routerLink="/page2" [routerLinkActive]= "activeClasses">
  Page 2
</a>
```

        - `<a>` with attributes `routerLink` set to `"/not-a-route"`, `routerLinkActive` set to `active` and *text* of `404 route`.
3. Save the file and open app.component.css for editing.
4. Add a class of `active` that sets the `background-color` and `color` to *colours of your choice*.
5. Add a class of `bordered` that sets the `border` to a style of your choice (e.g. `solid red 2px`).
6. Save the file and open app.component.ts for editing.
7. Add a *class property* of `activeClasses` that is a string array containing `active` and `bordered`.
8. Save the file and observe the browser.

You should notice that the styling is now applied to the link that you clicked on. You will also notice that the Home link always seems to be active...time to use `[routerLinkActiveOptions]`! Setting this attribute with an **object** that has a key of **exact** and a value of **true** makes sure that the CSS classes are only applied when the route is an exact match to the route being visited.

9. Open **app.component.html** for editing.

10. Modify the Home link so it is:

```
<li>
  <a
    routerLink="/"
    routerLinkActive="active"
    [routerLinkActiveOptions]="{exact: true}"
  >
    Home
  </a>
</li>
```

11. Save the file and return to the browser.

You should observe that the styling around the home link has now disappeared as the active class is only being applied to this link if the URL matches the route's path -and as we redirect this path to page1, this will never occur!

This is the end of Quick Lab 11d

Quick Lab 11e - Child Routes

Objectives

- To be able to add routing for modules
- To be able to create child routes

Overview

In this QuickLab, you will modify an existing application and leverage the power of child routes. You will use a decoupled routing module for a new feature of the application and define child routes within this.

Activity

Complete the section 'Before Each QuickLab' before continuing, working in the QuickLabs/11_Routing/starters/QL11e-f folder.

View the application. It should show 5 links, the final one being to Todo. Presently, this link will activate the *wildcard route* as nothing has been set up for the path '/todo'.

1. Open src/app/todo-feature/todo-feature-routing.module.ts for editing.
2. Remove the import for `CommonModule` and replace it with *imports* for `Routes` and `RouterModule` from @angular/router.
3. Remove `CommonModule` from the list of `imports` in the `NgModule` decorator.
4. Before the decorator, add a `const` called `todoRoutes` of type Routes setting it to an array with the following object:
    - Key `path` set to `'todo'`;
    - Key `component` set to `TodoComponent`;
    - Key `children` set to an *array of objects* defined as:
        - Key `path` set to `''`;
        - Key `component` set to `TodoListComponent`.
        - Key children set to an array of objects defined as:
            - Key `path` set to `'tododetails'`, Key `component` set to `TodoDetailComponent`;
            - Key `path` set to `''`, Key `component` set to `TodoHomeComponent`.
5. In the `imports` section of the `NgModule` decorator, add:

```
imports [RouterModule.forChild(todoRoutes)],
```

6. Add a key of `exports` to the decorator and set its value to

`[RouterModule]`.

7. Save the file.
8. Open **todo-feature.module.ts** for editing.
9. Add `TodoFeatureRoutingModule` to the list of `imports` in the decorator (ensuring that it is also added to the list of **imports** from libraries/files).
10. Save the file.
11. Open **todo.component.html** for editing and add a `<router-outlet></router-outlet>` under the current *markup*.
12. Save the file.
13. Open **app.module.ts** for editing.
14. In the list of `imports` in the decorator, add `TodoFeatureModule` ***BEFORE*** the import for `AppRoutingModule`

(You can experiment with what difference this makes when you have completed the set of instructions!)

15. Save the file and observe the browser.

Clicking on the **Todo** link should reveal:

- A *heading* of **Todos** (in a `<h1>` supplied by the `TodoComponent`)
- An *ordered list* of **todos** - supplied by the `TodoListComponent`, rendered because:
  - the path **/todo** has a ***child route*** that specifies to display it;
  - the `TodoComponent` has a <router-outlet> to display it.
- A paragraph asking you to click a **todo** to see details supplied by the `TodoHomeComponent`, rendered because:
  - the path **/todo** on the `TodoListComponent` has a ***child route*** that specifies to display it;
  - the `TodoListComponent` has a `<router-outlet>` to display it.

16. Make the path **/todo/tododetails**.

This should change the paragraph under the **todo** list. This is rendered because the `TodoListComponent` has a path that specifies displaying the `TodoDetailComponent` in its `<router-outlet>` when this path is hit.

This is the end of Quick Lab 11e

Quick Lab 11f - Parameterised Routes

Objectives

- To be able to work with Parameterised Routes

Overview

In this QuickLab, you will access the ActivatedRoute object in the component again to be able to decipher the parameters in the URL and use these to help populate the component. You will also create links dynamically in the template in conjunction with the *ngFor structural directive.

Activity

Continue working in the folder QuickLabs/11_Routing/starters/QL11e-f.

1. Open src/app/todo-feature/todo-detail/todo-detail.component.ts for editing.
2. Add a *class property* of `todo$` of type Observable<Todo>.
3. Import `Observable` from rxjs at the top of the file.
4. *Inject* the `todoService` of type TodoService and `route` of type ActivatedRoute into the constructor of the Component. Remember to make the necessary imports to the file.
5. The body of the `ngOnInit` function should:
    - Set `this.todo$` to the `paramMap` from `route` that is *piped* into `switchMap`;
    - `switchMap` should take `params` of type ParamMap as a *callback* argument and the body of the *callback* should:
        - Make a *call* to the `getTodo` method in the `todoService`, passing in the result of `params.get('id')`:

            `ParamMap` should be imported from @angular/router

            switchMap should be imported from rxjs/operators

```
…
    ngOnInit() {
        this.todo$ = this.route.paramMap.pipe(
            switchMap((params: ParamMap) => (
                this.todoService.getTodo(params.get('id'))
            )
        );
    }
…
```

6. Inject `router` of type Router into the component constructor.
    - `Router` should be imported from @angular/router
7. Add a *method* called `goToToDoHome` that:
    - *Calls* `this.router.navigate` with an argument of `['/todo']`.
8. Save the file.
9. Open todo-detail.component.html for editing.
10. Under the `<h2>`, add:
    - A `<div>` that uses `*ngIf` checking `todo$` *piped to* `async as todo`;
    - The content, it should *output* the `id` of the `todo` and `todoDetail`.
11. Under the `<div>` add a `<button>` that has a `click` *event* that *calls* `goToToDoHome()` and *text* of `Todo Home`.
12. Save the file.
13. Open todo-list.component.ts for editing.
14. Add a variable called `todos$` of type `Observable` that expects an array of `Todo` instances.
15. Declare `selectedId` as a `number`.
16. Inject the `todoService` and `route` (as `ActivatedRoute`) into the constructor.
17. Add an `ngOnInit` function that sets `todos$` to be:
    - A call to `pipe` on `route`'s `paramMap` that itself calls `switchMap` passing the `params` and:
        - Sets the `selectedId` to a call to `get` on `params` using the `id` property
          Hint: use `+params.get('id')` to return a number
        - Returns a call to `getAll()` on the `todoService`.
18. Save the file.
19. Open todo-list.component.html for editing.
20. In the <li>, surround the *bound data* with an **a** tag that has the `routerLink` attribute *evaluating* an **array** containing:
    - The **string** `'tododetails'`;
    - The `todo` object property `id`.
21. Add a `routerLinkActive` property set to `active`.
22. Open todo-feature-routing.module.ts for editing.

23. Add `/:id` to the end of the **route** that displays the `TodoDetailComponent`.
24. Save the file and observe the browser.
25. Click around the application ensuring that the links for the Todos work as expected and that the button navigates to the correct place.

This is the end of Quick Lab 11f

Quick Lab 11g - Route Guards

Objectives

- To explore Angular's Route Guards.

Activity

Complete the section 'Before Each QuickLab' before continuing, working in the QuickLabs/11_Routing/starters/QL11g folder.

Observe the application in the browser and note that navigation is allowed to/from both of the links supplied.

Overview – Parts 1 – 2

In these parts of the QuickLab you will implement the CanActivate and CanActivateChild route guards. You will create a service that returns the logic for the relevant guard. For CanActivate, you will make the service implement the CanActivate interface with a method that returns true or false dependent on a confirmation window. For CanActivateChild, you will make the service implement the CanActivateChild interface with a method that returns a Boolean dependent on a confirmation window. You will add the guard information to the routes array.

Part 1 – CanActivate

1. On the command line/terminal, navigate to src/app/route-guards.
2. Create a service called `activation` using the CLI command:

```
ng g service activation
```

3. Open the file activation.service.ts for editing and make the *service's class implement* the `CanActivate` *interface* (importing this from @angular/router).
4. Add a *method* `canActivate` to the class who's *body* uses `window.confirm` to ask the user: `` `Do you want to return true for CanActivate?` ``
5. Save the file.
6. Open route-guards-routing.module.ts for editing and in the *object* for the *child route* `canactivate`, add a key of `canActivate` with its *value* as a *one element array* containing the `ActivationService`.

```
  {
    path: 'canActivate',
    component: ActivatedComponent,
    data: {title: `CanActivate`},
    canActivate: [ActivationService]
```

7. Save the file and check that the prompt is shown when the CanActivate link is clicked in the browser.

Part 2 – CanActivateChild

1. Open route-guards-routing.module.ts for editing.
2. In the *object* for the *root path* `routeguards`, add a key of `canActivateChild` with its value as a *one element array* containing the `ActivationService`.
3. Save the file.
4. Open activation.service.ts for editing.
5. Make the *service's class* additionally implement the `CanActivateChild` interface (importing this from @angular/router).
6. Add a *method* `canActivateChild` to the class who's body uses `window.confirm` to ask the user: `` `Do you want to return true for CanActivateChild?` ``
7. Save the file.
8. Point the browser at *http://localhost:4200/routeguards*:
    - Note that there are no prompts here as the `CanActivateChild` route-guard is only activated when a child route is hit.
9. Point the browser at *http://localhost:4200/routeguards/canactivate*
    - The first prompt is the `CanActivateChild` as this is propagating from the parent route;
    - The second prompt is the `CanActivate` as this is propagating from the child route.
10. Click on the `CanDeactivate` link
    - Note that the only prompt here is for the `CanActivateChild` route-guard.

Clicking cancel at any point here makes the return of the method false and therefore the navigation does not occur.

Overview – Part 3

In this part of this QuickLab, you will construct an interface to specify the requirement of a canDeactivate method that needs to ultimately return a Boolean value. The CanDeactivate interface will be implemented by the service and be generically of the interface type created. The canDeactivate method will check to see if the passed component has a canDeactivate method itself that returns a Boolean, and then returns its own Boolean based on that. The component will be modified as will the route in the Routes array.

Part 3 – CanDeactivate

1. Open activation.service.ts for editing.
2. *Under the list of imports* but *before the @Injectable decorator*, `export` an `interface` `CanComponentDeactivate` that *defines a method* called `canDeactivate` that *should return* an `Observable` of type boolean or a `Promise` of type boolean or a `boolean`:

```
…
export interface CanComponentDeactivate {

  canDeactivate: () => Observable<boolean> | Promise<boolean>
| boolean;

}

@Injectable({…
```

Note that Observable will need to be imported from rxjs.

3. Make the `ActivationService` class additionally implement `CanDeactivate` with a *generic type* of `CanComponentDeactivate`:

```
…
export class ActivationService implements
CanDeactivate<CanComponentDeactivate>, … {

…
```

Note `CanDeactivate` will need to be imported from @angular/router.

4. Add a `canDeactivate` *method* to the *class* that:
    - Takes a value `component` of type CanComponentDeactivate as an argument;
    - *Returns* an `Observable` of type boolean, `Promise` of type boolean or a `boolean`;
    - Has a body that:
        - Checks `!component` *returning* `true`;
        - *Returns* a *ternary* from `component.canDeactivate` calling the method `component.canDeactivate()` or returning `true`:

```
canDeactivate(component: CanComponentDeactivate):
Observable<boolean> | Promise<boolean> | boolean {

    if (!component) { return true; }

    return component.canDeactivate ? component.canDeactivate()
:

      true;
```

```
}
```

5. Save the file.
6. Open route-guards/activated/activated.component.ts for editing.
7. Make the `ActivatedComponent` class implement `CanComponentDeactivate`, importing it from ActivationService.
8. Add a *class method* called `canDeactivate` that should return a `boolean`.
9. The body should *return* the *result* of a `window.confirm` asking the user: `Do you want to navigate away from this route? i.e return true for Can Deactivate`.
10. Save the file
11. Open route-guards/route-guards-routing.module.ts for editing.
    - Add a key `canDeactivate` to the path for `'candeactivate'` set to an array containing `ActivationService`
12. Save the file and return to the browser.

You should notice that if you are displaying the CanDeactivate-titled component, clicking away from the route brings the prompt up. Confirming allows navigation to continue, cancelling means the navigation does not occur.

This is the end of Quick Lab 11g

Quick Lab 12 - Pipes

Objectives

- To be able to use built-in Angular Pipes.
- To be able to create and use a custom pipe

Activity

Complete the section 'Before Each QuickLab' before continuing, working in the QuickLabs/12_Pipes/starter folder.

Overview

In this QuickLab, you will investigate the built-in pipes supplied by Angular, then create and test a custom pipe. The built-in date and async pipe will be used to demonstrate how in-built pipes can be used. A custom pipe to separate words will be created (and tested, if you have time) as the second and final part of the QuickLab.

Part 1 – Built-In Pipes

1. Open **app.component.ts** and observe that there are 2 class properties:
2. `today` is a **Date** - generated when the component is constructed;
3. `words$` is an **Observable** created from the *array supplied* to the `of` operator.
4. Open **app.component.html** for editing.
5. Add a *paragraph* that *displays* the `today` in the *date format* of **MMMM dd yyyy** using the in-built `date` pipe with parameters.

```
<p>Today's date is {{today | date: "MMMM dd yyyy"}}</p>
```

6. Save the file and observe the output.
7. Add an *unordered list* that lists *each word* in the `words$` observable by *piping it* into `async`:

```
<ul>
  <li *ngFor="let word of words$ | async>
    {{word}}
  </li>
</ul>
```

8. Save the file and observe the output.
9. *Pipe* each `word` into the *built-in* `uppercase` pipe.

```
<li *ngFor="let word of words$ | async>
```

```
  {{word | uppercase}}>
</li>
```

10. Save the file and check the output.

Part 2 – Creating and using a Custom Pipe

1. Point the command-line/ terminal at QuickLabs/12_Pipes/starter.
2. Generate a new pipe called `word-separator` using the CLI:

```
ng g pipe word-separator
```

Note that this adds the pipe as a declaration to the `AppModule` and supplies a name of `wordSeparator` to be used when using the pipe in the HTML

3. Open word-separator.pipe.ts for editing.
4. Edit the *signature* of the `transform` function so that it takes `word` as a string for its parameters and returns a string.
5. Make the body of the function return the supplied word with a space after it, unless the word is Observable when it should add an exclamation mark.

```
transform(word: string): string {
    if (word === `Observables`) {
      return `${word}!`;
    }
    return `${word} `;
}
```

6. Save the file.
7. Open app.component.html for editing.
8. Add a *paragraph* with a *span* inside it that uses an `*ngFor` to loop through each `word` in the `words$` observable (using the `async` pipe):
    - Display the `word` in the *span* ***piping*** it into the custom `wordSeparator` pipe.

```
<p>
  <span *ngFor="let word of words$ | async>
    {{word | wordSeparator}}
  </span>
</p>
```

9. Save the file and view the output.

If you have time…

- Write a test spec that tests the output of the transform function of the WordSeparator pipe.
- Write a test spec that tests that the AppComponent correctly renders the output of the WordSeparator pipe in the DOM. Remember that the data

that is fed into this pipe is asynchronous.

The solutions for these tests can be found in the relevant spec files.

This is the end of Quick Lab 12