

WA3142 Comprehensive Angular 12 Programming

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Introduction to Angular.....	3
Lab 2 - Introduction to TypeScript.....	12
Lab 3 - Introduction to Components.....	19
Lab 4 - Component Template.....	25
Lab 5 - Create a Photo Gallery Component.....	33
Lab 6 - Template Driven Form.....	40
Lab 7 - Create an Edit Form.....	47
Lab 8 - Reactive Form.....	52
Lab 9 - Develop a Service.....	58
Lab 10 - Develop an HTTP Client.....	61
Lab 11 - Use Pipes.....	65
Lab 12 - Basic Single Page Application Using Router.....	68
Lab 13 - Build a Single Page Application (SPA).....	73
Lab 14 - Advanced HTTP Client.....	88
Lab 15 - Using Angular Bootstrap.....	96
Lab 16 - Lazy Module Loading.....	102
Lab 17 - Advanced Routing.....	108
Lab 18 - Unit Testing.....	126
Lab 19 - Debugging Angular Applications.....	137

Lab 1 - Introduction to Angular

In this lab, we will install all the required software, create a very simple Angular application and use it from a browser. We won't get too deep into Angular at this point.

Part 1 - Install Angular CLI

Angular Command Line Interface (CLI) is an indispensable tool for Angular development. It can create a project, generate artifacts like components and build your project. We will now install Angular CLI.

__1. First, run these commands to make sure Node.js is installed correctly. Open a command prompt window and enter:

```
node --version
```

```
npm --version
```

Note: Node.js is only required in the development and build machines. We need it to install required packages and for Angular CLI to work. You don't need Node.js in a production machine.

__2. Run this command to install Angular CLI.

```
npm install -g @angular/cli@12
```

Note, we used the global (-g) option to install the tool in a central location. This will let us run **ng** (the CLI command) from anywhere.

We explicitly set the major version of the package to 10 since by the time you are doing this exercise there may be a newer version available.

__3. You will be asked to share usage data, enter your preference.

__4. It should take a few minutes to install the tool. After installation finishes enter this command to verify everything is OK.

```
ng --version
```

Verify that 12.x.x of Angular CLI is installed. For example:

```
Angular CLI: 12.2.4
Node: 16.0.0 (Unsupported)
```

Part 2 - Create a Project

Angular needs a lot of boiler plate code even for the simplest of projects. Fortunately, Angular CLI can generate most of this code for us.

__1. Create a folder '**C:\LabWork**' on your machine. This is where you will develop code for various labs.

__2. Open a new command prompt. Switch to the **C:\LabWork** folder.

```
cd C:\LabWork
```

__3. Enter this 2 commands to setup a user in git:

```
git config --global user.email "wasadmin@wasadmin.com"  
git config --global user.name "wasadmin"
```

__4. Run this command to create a new project called **hello**.

```
ng new my-test-app --defaults
```

Part 3 - Open Project in Editor

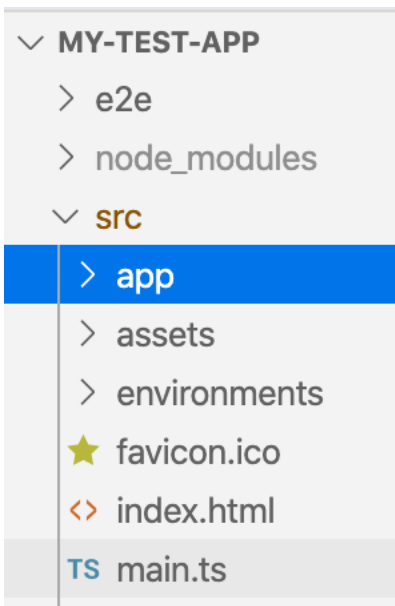
In this lab we will use Visual Studio Code. In real life you can use other editors like Atom, Sublime and WebStorm.

__1. Launch VS Code.

__2. From the menu select **File > Open Folder**.

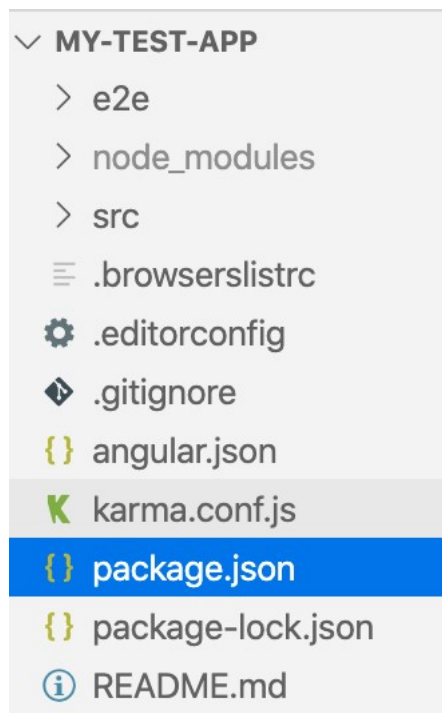
__3. Select the **C:\LabWork\my-test-app** folder and open it.

__4. The project's directory structure should look like this at this point.



Part 4 - The Anatomy of an Angular Project

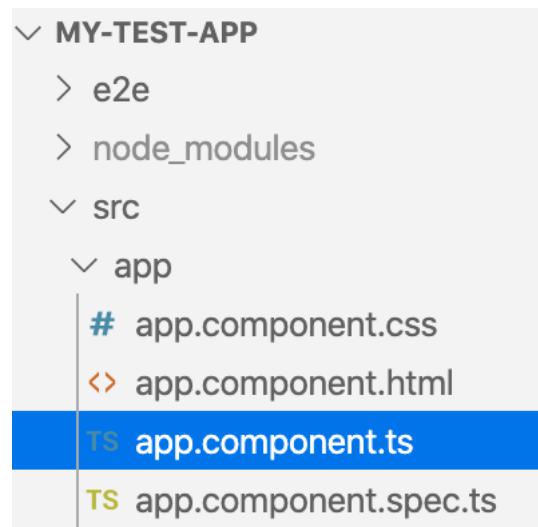
__1. In VS Code, click **package.json** to open it.



The Angular framework is released as several separate Node.js packages. For example, common, core, http, router etc. In addition, Angular depends on third party projects such as rxjs and zone.js. The **package.json** file declares the packages we have dependency on. Specifically, the **dependencies** property lists packages that we need at compile and runtime. The **devDependencies** section lists packages that we need for certain development tasks. You will see **typescript** listed there. We need this to be able to compile our TS code into JS.

These packages are physically located inside the **node_modules** folder. This folder should not be added to a version control system like git. Since anyone can easily run **npm install** to install these packages.

Next, we will look at the source code of the generated project. We have not covered much of TypeScript or Angular yet, so do not worry too much about the details. There will be plenty of more labs to learn the details.



2. Open the root component class **src/app/app.component.ts**.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'hello';
}
```

The **import** statement tells the compiler that the name "Component" is available from the **@angular/core** module. Which means we can now start using the "Component" name for its intended purpose, which may be anything from a class name to a variable name. In this particular case "Component" is a decorator. By using the decorator with the **AppComponent** class, we are saying that **AppComponent** is an Angular component.

We are also adding a few meta data elements to the decorator. The **selector** property declares the HTML tag for the component. The **templateUrl** property points to the HTML template of the component.

What is a Component?

A component is a TypeScript class that is responsible for displaying data to the user and collecting input from the user.

Every Angular application has one main component. This is displayed first to the user when the application launches. Our AppComponent class here is such a main component. We will soon see how we display it to the user by bootstrapping it.

__3. Briefly look at the **app.component.html** file. This is the template code of AppComponent. We will revisit this file later.

__4. Next open the application module file **src/app/app.module.ts**.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

An Angular module is a collection of components, services, directives etc. You can loosely think of it like a Java JAR file or a C++ DLL. Every application must have a root level module called application module. Our AppModule class is such an application module.

Every component that is a member of a module must be listed in the **declarations** array.

The main component of the application must also be listed in the **bootstrap** array.

__5. Next open **src/main.ts**. This file is the entry point of your application. It bootstraps or attaches the application module to the browser.

__6. Finally, open **src/index.html**.

__7. Note how the root component is inserted there.

```
<body>  
  <app-root></app-root>  
</body>
```

When the AppModule is bootstrapped the DOM tree generated by the template of AppComponent will be inserted where <app-root></app-root> appears.

Notice that no JavaScript files are currently imported into the index.html. How does our code get loaded into the browser? All such dependencies will be added into the index.html file by the build process.

Part 5 - Run the Development Server.

In production all you need is a static web server like Apache and IIS to serve an Angular application. But, during development you should access your Angular application site using the Angular CLI provided web server. It has many benefits. Among which are:

- It builds the application in debug mode.
- Rapidly rebuilds the application when you modify any file.
- Automatically refresh the browser after build finishes.

__1. In VS Code open a new terminal window by selecting **Terminal > New Terminal** from the menu bar. This will open a terminal already switched the root folder of our project **C:\LabWork\my-test-app**.

__2. Run this command to build the project and start the web server.

```
npm start
```

__3. If you are asked to share the information, enter your preference to continue.

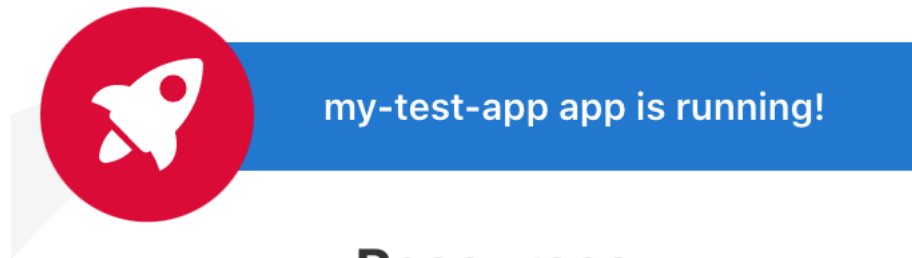
This will compile the TypeScript files and start the development server.

Once this process is running, it will recompile TypeScript, rebundle the result and repost the bundles to the browser every time file changes are saved.

__4. Try loading the application by entering the following URL into the address bar of the Chrome browser.

```
http://localhost:4200/
```


You should see the following:



Resources

Here are some links to help you get started:

__5. Leave the browser and the server running for the next section.

__6. View the source code of the page. Notice that a bunch of `<script>` tags have been added at the end of the `<body></body>` element. These scripts include your own application code, Angular library code and anything else your application may depend on.

Part 6 - Change the Component Code

We will make a small change. You will see how the 'start' command will automatically recompile the code.

__1. Open `src/app/app.component.ts` in an editor.

__2. Replace the 'title' property with a 'name' property.

```
export class AppComponent {  
  name = 'Goofy Goober'  
}
```

__3. Save changes.

__4. Open `app.component.html`.

__5. Delete the entire content of the file. Then enter this line:

```
<h1>I 'm {{name}}</h1>
```

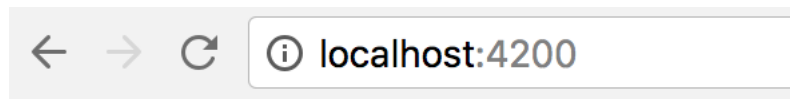
__6. Save the file.

The component has some state in the form of a property called 'name'. It is rendering the value of this "name" variable in the HTML template using the `{{name}}` syntax.

Back in the server command prompt, you should see messages about the code being recompiled and bundles being rendered.

Note: The 'npm start' command executes the 'ng serve' command which invokes the Angular build and deployment system. The system listens for file changes and makes sure that any TypeScript files that are changed are recompiled to JavaScript and included in one of the *.js bundles. These bundles are then made available through the development server. This all happens automatically so that any changes you make to the code will appear very quickly in the browser. If the code you added resulted in compile errors you will see those in the command prompt.

__7. Go to the browser window and verify that the changes you made to the app appear. If everything went well you will see the following in your browser:



I'm a Goofy Goober

__8. Let's clean up. First, close out the server by entering Ctrl-C in the Terminal window and press Y.

__9. Close out the browser tab where you were viewing the application.

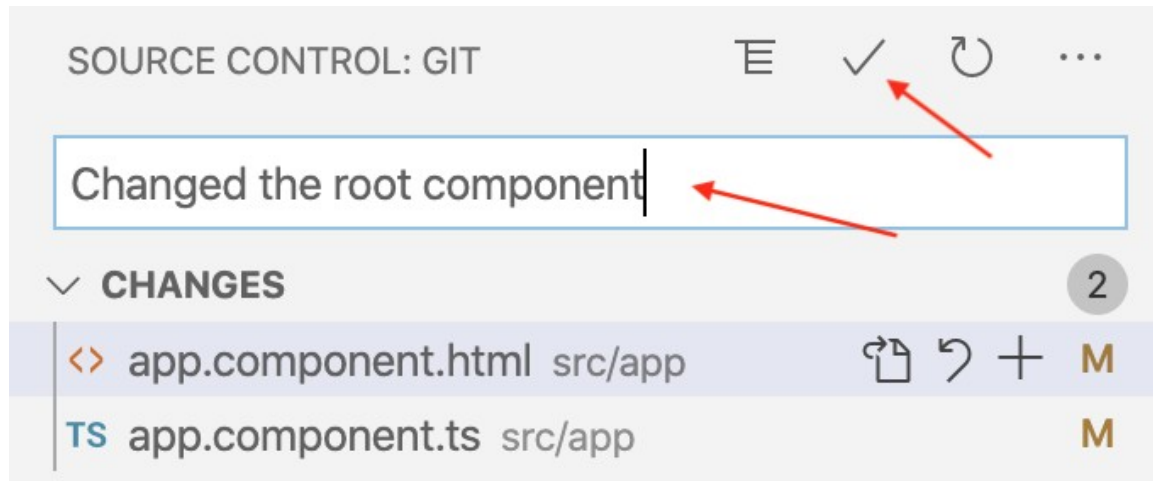
__10. Close any open files and command prompt windows.

Part 7 - Version Management

In these labs we will commit our changes to Git from time to time. This will help us recover from any errors. In VS Code left side bar click the source control icon.



__1. Enter a commit message like below.



__2. Click the commit button (with a check mark) to commit changes. Click **Yes** when you're asked to confirm.

Part 8 - Review

In this lab we learned:

- How to create an Angular application
- What are some of the key files in a project
- How to build and serve the application in development mode
- How to commit changes to Git using VS Code.

Lab 2 - Introduction to TypeScript

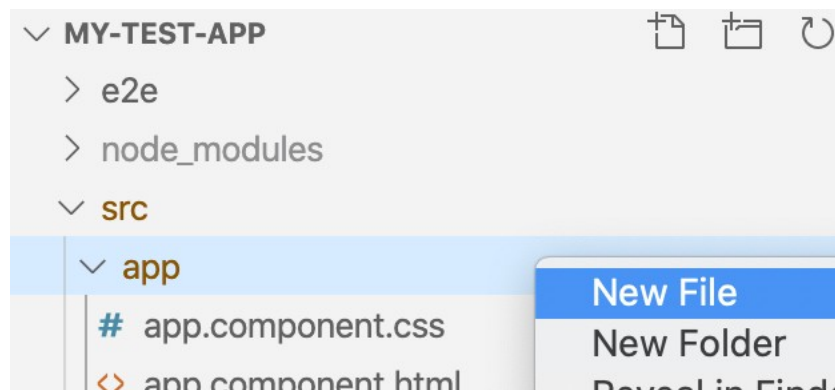
TypeScript is a superset of JavaScript ES6. Its main contribution is to make ES6 strongly typed. The compiler can catch many type related problems. Strong typing also allows an editor to offer code completion, refactoring and code navigation.

In this lab, we will get introduced to the type system of TypeScript.

Part 1 - Get Started

We will now create a file called experiments.ts where we will write all the code for this lab.

__ 1. Open **C:\LabWork\my-test-app** in VS Code unless it is already open.



__ 2. Right click the **app** folder and select **New File**.

__ 3. Type **experiments.ts** as the file name and hit the Enter key.

Quickest way to execute our code will be to simply import experiments.ts from the AppComponent class file.

__ 4. Open **app-component.ts**.

__ 5. Add this import statement at the top. Note: Do not add the ".ts" extension.

```
import './experiments'
```

__ 6. Save the file.

Now we can write code in experiments.ts. Angular CLI will include it in the build process.

Part 2 - Variable Types

__1. Open **experiments.ts**.

__2. In this file, add these lines to define and print a numerical variable.

```
var age: number = 20  
  
console.log("Age is", age)
```

The ":number" bit comes from TypeScript. The rest is basic JavaScript.

__3. Save changes.

Part 3 - Run the Code

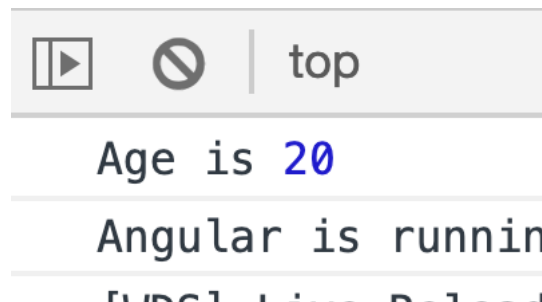
__1. In VS Code open a new terminal by selecting **Terminal > New Terminal** from the menu.

__2. Enter the command to start the development server.

```
npm start
```

__3. Open a Chrome browser and go to **http://localhost:4200/**

__4. Open the Developer Tools (CTRL+Shift+I) and select the **Console** tab of the browser. You should see this:



__5. Now we will deliberately introduce a type error. In experiments.ts initialize the age variable like this.

```
var age: number = "Too old"
```

__6. Save experiments.ts.

__7. Verify that the terminal is showing the error message.

ERROR in `src/app/experiments.ts:1:5`

```
1 var age: number = "Too old"
   ~~~
```

This error illustrates type safety. It shows that we have incorrectly assigned a string type to a number variable.

Part 4 - Function Argument Types

__1. Delete or comment out all lines in `experiments.ts`.

__2. Add a function like this.

```
function printPerson(name:string, age:number) {
    console.log(`Name: ${name} age: ${age}`)
}

printPerson("Billy", 8)
```

Watch out for the string with back ticks. It is a template string where you can insert variables using `${variable}` syntax.

__3. Save changes.

__4. Verify that browser's console shows:

```
Name: Billy age: 8
```

__5. Edit `experiments.ts` and add a call to the "printPerson" function using incorrect parameter types. For example, the following call reverses the order of parameters:

```
printPerson(8, "Billy")
```

__6. Save the file. Verify that the terminal shows the compilation error.

__7. Fix the problem.

Part 5 - Function Return Type

__1. At the bottom of the file, add this function:

```
function isMinimumAge(age: number) : boolean {  
    return age >= 21  
}  
  
console.log("Is minimum age:", isMinimumAge(22))
```

__2. Save changes. Verify output in browser console.

Part 6 - Creating a Class

We will learn about class.

__1. Delete or comment out all lines in **experiments.ts**.

__2. Add a class like this.

```
class Product {  
    id: number  
    published = true  
    title!: string  
    price?: number  
  
    constructor(id: number) {  
        this.id = id  
    }  
  
    printDetails() {  
        console.log("ID:", this.id)  
        console.log("Title:", this.title)  
        console.log("Price:", this.price)  
        console.log("Published:", this.published)  
    }  
}
```

TypeScript is very strict about uninitialized variables. In this class the "id" and "published" fields are initialized after an instance is constructed. But "title" and "price" are not initialized. By using "title!: string" we are promising that somehow this field will be always initialized prior to use. The compiler then backs off from giving any error messages. The use of "price?: number" has a different meaning. It says that the field may be a number or undefined. In other words, it is an optional type.

__3. Use the class as follows.

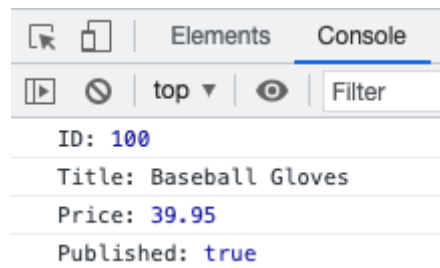
```
let p = new Product(100)
```

```
p.title = "Baseball Gloves"
p.price = 39.95

p.printDetails()
```

__4. Save changes.

__5. The output in the browser console should look like this:



Part 7 - Define Members in Constructor

Typescript allows you to define member variables in the constructor. Let's get familiar with that syntax.

__1. Change the Product class like this. Modified portion is shown in bold face.

```
class Product {
    constructor(public id: number,
        public title: string,
        public price: number,
        public published = true) {
    }

    printDetails() {
        console.log("ID:", this.id)
        console.log("Title:", this.title)
        console.log("Price:", this.price)
        console.log("Published:", this.published)
    }
}
```

Notice, how "published" has a default value. If we omit this parameter in the constructor a value of true will be assigned.

__2. Use the class like this.

```
let p = new Product(100, "Baseball Gloves", 39.95)
```



```
p.printDetails()
```

__3. Save changes.

__4. The output will look same as before.

Part 8 - Using Generics

Generics lets us delay specifying the data types to a later time. For example, the Map class can not know what the data type of the key and value are going to be. It will depend on what you decide to store in the Map.

__1. In experiments.ts, add this code at the bottom.

```
let database = new Map<number, Product>()
```

This will create a map that will only accept number as keys and Product as values. Anything else will lead to a compile error.

__2. Add the following code.

```
let p1 = new Product(100, "Baseball Gloves", 39.95)
let p2 = new Product(350, "Golf Club", 129.95)

database.set(p1.id, p1)
database.set(p2.id, p2)

database.forEach((p, key) => {
    console.log("Key is:", key)
    p.printDetails()
})
```

__3. Save changes.

__4. Verify the output in browser console.

Part 9 - Using Generics in a Function

Functions can also defer specifying types using generics. Consider a function that returns a value from a map using its key. If no value exists it returns a default value. The function has no idea about the types of keys and values. It has to use generics.

__1. At the bottom of experiments.ts add this code. Here K is a placeholder for the type of the key and V is the same for the value.

```
function getOrDefault<K, V>(
    key: K,
    map: Map<K, V>,
    defaultValue:V) : V {
    let data: V = map.get(key) ?? defaultValue
    return data
}
```

__2. Use the function like this.

```
let p3 = getOrDefault(11, database, new Product(0, "Unknown", 0.0))
p3.printDetails()
```

__3. Save changes.

__4. Verify that browser console shows output from the unknown product.

Part 10 - Clean Up

__1. Open `src/app/app.component.ts`.

__2. Delete the line that imports `experiments.ts`.

```
import './experiments'
```

__3. Save changes.

Part 11 - Review

In this lab, we took a look at some commonly used features of TypeScript.

In summary:

- Variables and function parameters can have strong typing.
- You can develop classes.
- You can define class variables in the constructor.
- Generics are a powerful way to write classes and methods without knowing the exact nature of the types.

Lab 3 - Introduction to Components

Components are the basic building blocks in Angular. They are used to:

- Render data on a page
- Handle events
- Accept and verify user input

Every Angular application must at minimum have the root AppComponent. Most applications will need to develop additional components.

In this lab we will write our very first component. It will show how many times a button was clicked.

You have clicked 5 times.

Click Here

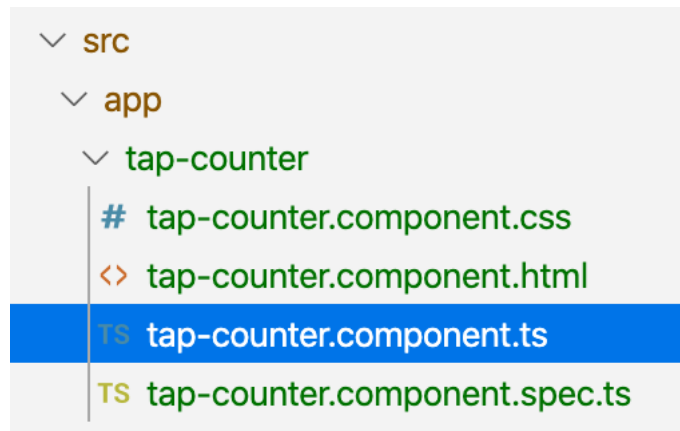
Part 1 - Get Started

- __ 1. From VS Code open the **C:\LabWork\my-test-app** folder if not already open.
- __ 2. From VS Code open a terminal if not already open.
- __ 3. If the development server is running in the terminal close it by hitting Control+C.

Part 2 - Create the Component

- __ 1. Enter this command in the terminal to create the component.

```
ng generate component tap-counter
```



__2. Open **src/app/tap-counter/tap-counter.component.ts**. Observe that this contains the component's class `TapCounterComponent`. Note that "app-tap-counter" is the selector of `TapCounterComponent`. We will need this later.

__3. Open the application module file **src/app/app.module.ts**.

__4. Notice that the newly created component was added to the declarations list. Without that we can't use a component in a template.

```
declarations: [  
    AppComponent,  
    TapCounterComponent  
],
```

Part 3 - Write the Component Code

__1. Open **tap-counter.component.ts**.

__2. In the `TapCounterComponent` class, add a member variable that will keep track of how many times the button was clicked.

```
export class TapCounterComponent implements OnInit {  
    tapCount = 0  
    ...  
}
```

__3. Add a method to the class that will increase the counter.

```
handleClick() {  
    this.tapCount += 1  
}
```

__4. Save changes.

Part 4 - Write the Template

__1. Open the template file **tap-counter.component.html**.

__2. Delete all content.

__3. Add these lines.

```
<div>
  <p>You have clicked {{tapCount}} times.</p>

  <button (click)="handleClick()">Click Here</button>
</div>
```

Here **{{tapCount}}** displays the current value of the tapCount class variable. And **(click)="handleClick()"** sets up a click event handler for the button. Rest is regular HTML.

__4. Save changes.

Part 5 - Use the Component

__1. Open the root component's template file **src/app/app.component.html**.

__2. Delete all contents.

__3. Add this line to add the tap counter component to this template.

```
<div>Click on the button below.</div>

<app-tap-counter></app-tap-counter>
```

Note that "app-tap-counter" is the selector of TapCounterComponent.

__4. Save.

Part 6 - Test

__1. From the terminal of VS Code run.

```
npm start
```

__2. In the browser visit **http://localhost:4200/**. You should see this.

You have clicked 0 times.

Click Here

__3. Click the button a few times. Make sure the counter goes up.

Part 7 - Work with Component Style

__1. Open **tap-counter.component.css**.

__2. Add this style rule.

```
div {  
  border: thin black solid;  
  max-width: 200px;  
  border-radius: 5px;  
  padding: 5px;  
}
```

__3. Save changes.

__4. The page will now look like this.

Click on the button below.

You have clicked 0 times.

Click Here

Notice how the style for `<div>` tag only applies to the child component and not for the parent. This is because style rules are private to a component. They are not visible to the parent or its children.

Part 8 - Use Multiple Copies of the Component

- ___1. Open the root component's template file **src/app/app.component.html**.
- ___2. Add multiple copies of the tap counter component like shown in boldface.

```
<div>Click on the button below.</div>
```

```
<app-tap-counter></app-tap-counter>  
<app-tap-counter></app-tap-counter>  
<app-tap-counter></app-tap-counter>
```

- ___3. Save.

Click on the button below.

You have clicked 2 times.

Click Here

You have clicked 5 times.

Click Here

You have clicked 0 times.

Click Here

- ___4. Verify that each occurrence of the component is able to maintain its own state. This proves that a separate instance of the component class is created each time the `<app-tap-counter>` tag is used in a template.

Part 9 - Review

In this lab, we developed a very simple component. But this example was enough to see important aspects of a component:

- The component needs to be added to the module's declarations list. CLI does this for us.
- A component can maintain internal state using member variables.
- To use a component in a template we use its selector as the tag such as `<app-tap-counter>`.
- A component renders display using its template. We can also setup event handling there.

Lab 4 - Component Template

In this lab we will get deeper into the template syntax. We will learn to use directives like `ngFor` and `ngClass`.

We will build an automobile dealer inventory web site. It will show a list of vehicles available for sale.

Current Inventory

2012 HONDA, Civic

VIN: Y123

70000 Miles, \$5900

Delete

2019 BMW, 328i

Featured car of the month!

VIN: P1023

42000 Miles, \$12000

Delete

Part 1 - Get Started

- ___ 1. From VS Code open the **C:\LabWork\my-test-app** folder if not already open.
- ___ 2. From VS Code open a terminal if not already open.
- ___ 3. If the development server is running in the terminal close it by hitting **Control+C**.
- ___ 4. Commit all previous code changes to Git.

Part 2 - Create the Component

- ___ 1. From the VS Code terminal run this command.

```
ng g c dealer-inventory
```

Note, this is a shortcut for **ng generate component dealer-inventory**.

Part 3 - Write the Vehicle Class

We will now develop a class called Vehicle. It will represent a car available for sale.

__1. In VS Code right click the **app** folder and select **New File**.

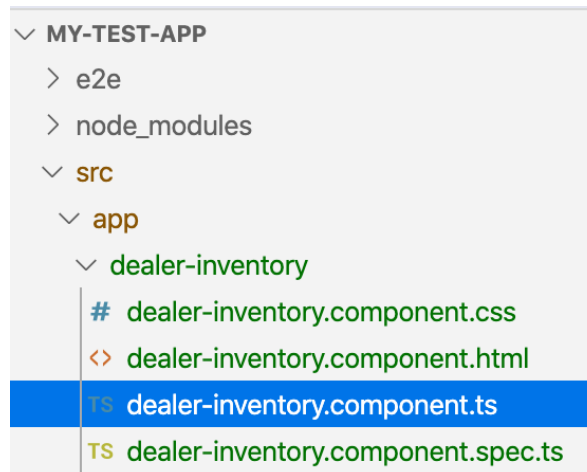
__2. Enter **vehicle.ts** as the file name and hit Enter key.

__3. In vehicle.ts file add this code.

```
export class Vehicle {
  constructor(public VIN: string,
    public year: number,
    public make: string,
    public model: string,
    public mileage: number,
    public price: number,
    public featured: boolean,
    public photos: string[]) {
  }
}
```

__4. Save.

Part 4 - Write Component Code



__1. Open **src/app/dealer-inventory/dealer-inventory.component.ts**.

__2. Add this import statement at the top.

```
import { Vehicle } from '../vehicle'
```

__3. Within the component class add a member variable to store a list of vehicles. We will hard code the vehicles for now. You can copy and paste this code from **C:\LabFiles\cars.txt**.

```
export class DealerInventoryComponent implements OnInit {
  inventory:Vehicle[] = [
    {
      VIN: "Y123",
      year: 2012,
      make: "HONDA",
      model: "Civic",
      mileage: 70000,
      price: 5900.00,
      featured: false,
      photos: []
    },
    {
      VIN: "P1023",
      year: 2019,
      make: "BMW",
      model: "328i",
      mileage: 42000,
      price: 12000.00,
      featured: true,
      photos: ["/assets/b-1.png", "/assets/b-2.png", "/assets/b-3.png",
"/assets/b-4.png"]
    },
    {
      VIN: "NM182",
      year: 2018,
      make: "KIA",
      model: "Niro",
      mileage: 31000,
      price: 7900.00,
      featured: false,
      photos: ["/assets/k-1.png", "/assets/k-2.png", "/assets/k-3.png"]
    },
    {
      VIN: "Y187",
      year: 2014,
      make: "HONDA",
      model: "Accord",
      mileage: 40000,
      price: 8900.00,
      featured: false,
      photos: []
    }
  ]
  ...
}
```

__4. Save.

Part 5 - Write the Template

We will use ngFor to loop through all the cars and display their information.

__ 1. Open **dealer-inventory.component.html**.

__ 2. Delete all contents of the file.

__ 3. Enter this code.

```
<h1>Current Inventory</h1>

<div *ngFor="let car of inventory">
  <h3>{{car.year}} {{car.make}}, {{car.model}}</h3>
  <p>VIN: {{car.VIN}}</p>
  <p>{{car.mileage}} Miles, ${{car.price}}</p>
</div>
```

__ 4. Save.

Part 6 - Use the Component

__ 1. Open **app.component.html**.

__ 2. Delete all contents.

__ 3. Add the dealer inventory component like this.

```
<app-dealer-inventory></app-dealer-inventory>
```

__ 4. Save.

Part 7 - Test

__ 1. From VS Code terminal run.

```
npm start
```

__ 2. In a browser visit **http://localhost:4200/**.

__ 3. Verify that you see this.

Current Inventory

2012 HONDA, Civic

VIN: Y123

70000 Miles, \$5900

2019 BMW, 328i

VIN: P1023

42000 Miles, \$12000

Part 8 - Show Featured Vehicle

We will now style a featured vehicle more prominently.

___ 1. Open **dealer-inventory.component.css**.

___ 2. Add this CSS class.

```
.featured-item {  
  border: thin black solid;  
  border-radius: 5px;  
  padding: 5px;  
  background-color: #cfcfcf;  
}
```

___ 3. Save.

We will now conditionally apply this CSS class.

___ 4. Open **dealer-inventory.component.html**.

___ 5. In the <div> tag add ngClass as shown in bold face below.

```
<div *ngFor="let car of inventory"  
  [ngClass]="{'featured-item': car.featured}">
```

This will apply the "featured-item" CSS class only if car.featured is true.

__6. Add a special message for featured cars. This is shown in bold face below.

```
<h3>{{car.year}} {{car.make}}, {{car.model}}</h3>
<p *ngIf="car.featured">Featured car of the month!</p>
<p>VIN: {{car.VIN}}</p>
```

__7. Save.

__8. Verify that the BMW is shown like this.

70000 Miles, \$5900

2019 BMW, 328i

Featured car of the month!

VIN: P1023

42000 Miles, \$12000

2018 KIA, Niro

Part 9 - Add trackBy Support

By default ngFor uses JavaScript object identity to uniquely identify each item in an array. This doesn't always work. You need to use trackBy to tell Angular the unique key value for each car. That will be the VIN number.

Note: Almost always consider adding trackBy with every ngFor.

__1. Open **dealer-inventory.component.ts**.

__2. Add this method to the class. It returns the unique key of a car.

```
trackByVIN(index:number, car:Vehicle) : string {
    return car.VIN
}
```

__3. Save.

__4. Open **dealer-inventory.component.html**.

__5. Use the track by function as shown in bold face below.

```
<div *ngFor="let car of inventory; trackBy: trackByVIN" ...>
```

__6. Save changes.

__7. Verify the page is still showing.

Part 10 - Event Handling from a Loop

We will now add support for deleting a vehicle from inventory.

__1. Open **dealer-inventory.component.ts**.

__2. Add this method to the class.

```
deleteVehicle(car:Vehicle) {  
  this.inventory = this.inventory.filter(c => c.VIN !== car.VIN)  
}
```

__3. Save.

__4. Open **dealer-inventory.component.html**.

__5. Add the delete button as shown in bold face.

```
<div *ngFor...>  
  ...  
  <p>  
    <button (click)="deleteVehicle(car)">Delete</button>  
  </p>  
</div>
```

Notice how we are able to pass the car object to the event handler method.

__6. Add this code shown in bold face to deal with empty inventory.

```
<h1>Current Inventory</h1>  
<p *ngIf="inventory.length==0">There are no cars in inventory.</p>
```

__7. Save changes.

__8. Test the page by clicking the Delete buttons. Make sure that the vehicles are removed correctly.

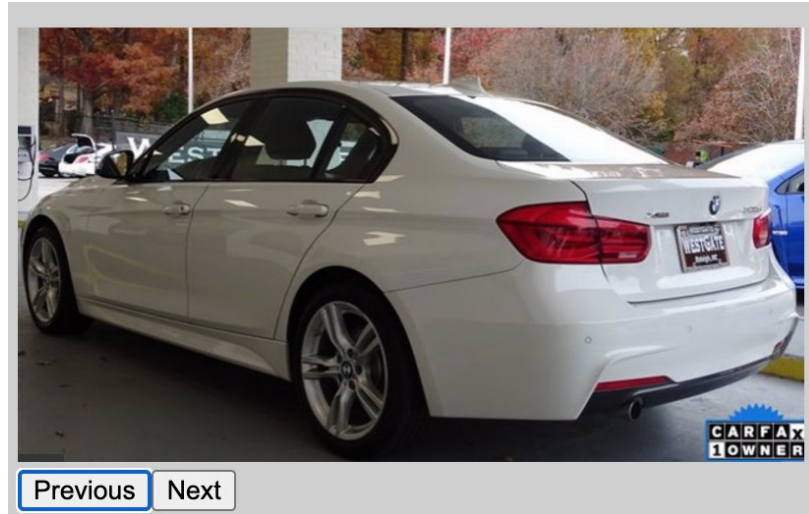
Part 11 - Review

In this lab we learned to use ngFor, ngIf and ngClass attributes. We also learned how to handle events from within a loop.

Lab 5 - Create a Photo Gallery Component

In this lab we will build a component that shows a slide show of photos. We will later use it with the dealer inventory project.

This lab will show you how to exchange data and events between components using `@Input` and `@Output`.



Part 1 - Get Started

- ___ 1. From VS Code open the **C:\LabWork\my-test-app** folder if not already open.
- ___ 2. From VS Code open a terminal if not already open.
- ___ 3. If the development server is running in the terminal close it by hitting Control+C.
- ___ 4. Commit all previous code changes to Git.

Part 2 - Create the Component

- ___ 1. In the VS Code terminal enter this command.

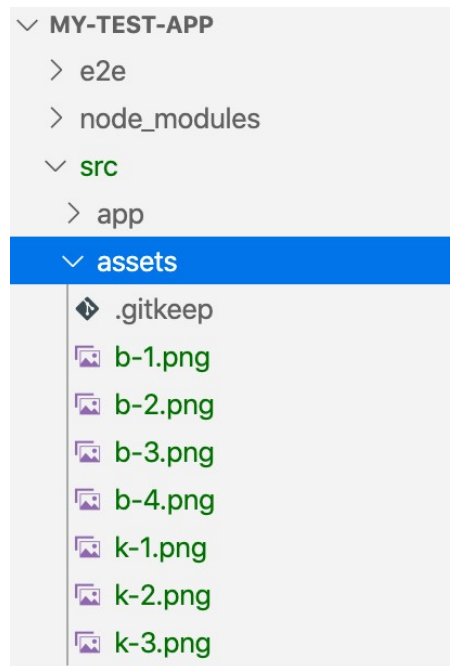
```
ng g c photo-gallery
```

- ___ 2. We need a few photos. Copy all images:

From: C:\LabFiles\car-photos

To: C:\LabWork\my-test-app\src\assets

___3. Make sure you can see the photos from VS Code like this.



Part 3 - Write the Component

___1. Open `src/app/photo-gallery/photo-gallery.component.ts`.

___2. Add a few names to the imports list. They are all available from the core package.

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

___3. In the `PhotoGalleryComponent` class, add these two member variables.

```
@Input("photos")
imageList: string[] = []

@Output("on-navigate")
emitter = new EventEmitter

currentIndex = 0
```

A list of photo URLs will be supplied as input to this component. Every time user navigates to a new photo we will raise the "on-navigate" event.

__4. Add these methods to the class.

```
moveNext() {
  if (this.currentIndex < this.imageList.length - 1) {
    this.currentIndex += 1
    //Raise event with index as payload
    this.emitter.emit(this.currentIndex)
  }
}

movePrevious() {
  if (this.currentIndex > 0) {
    this.currentIndex -= 1
    //Raise event with index as payload
    this.emitter.emit(this.currentIndex)
  }
}
```

__5. Save changes.

Part 4 - Write the Template

__1. Open **photo-gallery.component.html**.

__2. Delete all content.

__3. Add the following code.

```
<div>
  <div>
    <img [src]="imageList[currentIndex]" />
  </div>
  <div>
    <button (click)="movePrevious()">Previous</button>
    <button (click)="moveNext()">Next</button>
  </div>
</div>
```

Note how we are using the [src] syntax to bind an expression to the src DOM attribute of the img tag.

__4. Save.

Part 5 - Write the CSS

__1. Open **photo-gallery.component.css**.

__2. Add this style rule.

```
img {  
    width: 100%;  
}
```

This will limit the image within its enclosing element.

__3. Save changes.

Our gallery component is now ready for use.

Part 6 - Use the Gallery Component

We will now use the gallery component to show photos of cars.

__1. Open **dealer-inventory.component.html**.

__2. Bellow the line:

```
<h3>{{car.year}} {{car.make}}, {{car.model}}</h3>
```

Add:

```
<p class="gallery">  
  <app-photo-gallery  
    *ngIf="car.photos.length > 0" [photos]="car.photos">  
  </app-photo-gallery>  
</p>
```

We are doing a few things here:

- Adding a gallery for every car that has photos.
- We are supplying the list of photos using [photos] syntax.

__3. Save changes.

__4. Open **dealer-inventory.component.css**.

__5. Add this CSS class.

```
.gallery {  
    max-width: 400px;  
}
```

__6. Save changes.

Part 7 - Test

__1. From VS Code terminal start the server.

```
npm start
```

__2. In a browser visit **http://localhost:4200/**.

__3. Verify that the gallery shows up for some of the cars.



__4. Try clicking the Previous and Next buttons. The images should change.

Part 8 - Disable Buttons

Right now we are properly bounds checking currentIndex. But we should also disable the Previous and Next buttons if clicking on them will do nothing.

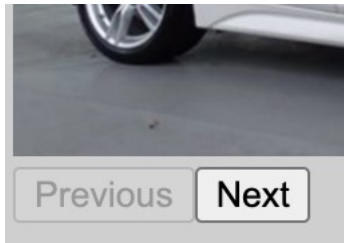
__1. Open **photo-gallery.component.html**.

__2. Bind the disabled attribute of the buttons using the code shown in bold face below.

```
<button [disabled]="currentIndex==0"  
  (click)="movePrevious()" ">Previous</button>  
<button [disabled]="currentIndex==imageList.length-1"  
  (click)="moveNext()" ">Next</button>
```

__3. Save changes.

__4. Verify that the buttons are getting disabled when they can not be clicked.



Part 9 - Handle Event

We will now show a promotional message if the user navigates to the last photo.

__1. Open **dealer-inventory.component.ts**.

__2. Add this method to the class.

```
handlePhotoNavigation(photoIndex:number, car:Vehicle) {  
  if (photoIndex == car.photos.length - 1) {  
    alert("Come visit us in our showroom!")  
  }  
}
```

__3. Save.

__4. Open **dealer-inventory.component.html**.

__5. Attach an event handler like shown in bold face.

```
<app-photo-gallery  
  (on-navigate)="handlePhotoNavigation($event, car)" ...>
```

Recall the event was named "on-navigate" and the event carried the index of the current photo as the payload.

__6. Save changes.

__7. Verify that when you navigate to the last photo you see the promotional message.

Part 10 - Review

In this lab we learned:

- How to send data to a child component using `@Input`.
- How a child can raise a custom event using `@Output`.
- How to use the `[]` syntax to set DOM attribute value.

Lab 6 - Template Driven Form

In this lab we will create a form that users can use to add a new vehicle to the inventory.

Angular provides two different ways of building a form:

- Template driven – The form is entirely defined in HTML template.
- Reactive – Several aspects of the form are defined in component code. This makes your form much easier to unit test.

This lab focuses on the template driven approach.

Part 1 - Design Decisions

We can define the form in the dealer inventory component. But we need to consider the fact that we will need a second form in future to edit a vehicle. Much of the code will then need to be duplicated. It will be better if we can isolate the form handling part in a separate component. That is exactly what we will do.

Part 2 - Get Started

- __ 1. From VS Code open the **C:\LabWork\my-test-app** folder if not already open.
- __ 2. From VS Code open a terminal if not already open.
- __ 3. If the development server is running in the terminal close it by hitting Control+C.
- __ 4. Commit all previous code changes to Git.

Part 3 - Create the Form Handler Component

- __ 1. From VS Code terminal run:

```
ng g c vehicle-form
```

Part 4 - Import FormsModule

Template driven forms use directives such as `ngForm` and `ngModel` that are available from the `FormsModule`. We must import it now.

- __ 1. In VS Code open **app.module.ts**

__2. Add this import statement at the top.

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

__3. Add the FormsModule to the imports list of the NgModule decorator. Now all directives from FormsModule will be available throughout our application.

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

__4. Save changes.

Part 5 - Write the Component

__1. Open `src/app/vehicle-form/vehicle-form.component.ts`.

__2. Add these additional imports.

```
import { Component, OnInit, Output, EventEmitter } from  
'@angular/core';  
  
import { NgForm } from '@angular/forms';  
import { Vehicle } from '../vehicle';
```

__3. This component will raise an event when the form is submitted. Add this event emitter to the class.

```
@Output("on-submit")  
emitter = new EventEmitter
```

__4. Add a method that will be called when the submit button is clicked.

```
handleSubmit(nForm:NgForm) {  
  const input = nForm.value  
  const v = new Vehicle(  
    input.veh_vin,  
    input.veh_year,  
    input.veh_make,  
    input.veh_model,  
    input.veh_mileage,  
    input.veh_price,  
    input.veh_featured === "" ? false : input.veh_featured,  
    [])  
  
  this.emitter.emit(v)  
}
```

Note:

- We are accessing user input using property names like "veh_vin" and "veh_year". We need to use these same names for the <input> tags.
- Working with checkbox in HTML is always tricky. If user never clicks on it Angular returns an empty string as value. Else it returns true or false depending on if the checkbox was checked or not.

__ 5. Save.

Part 6 - Write the Template

We will create the form template now. We will keep things very basic and not get into validation yet.

__ 1. Open **vehicle-form.component.html**.

__ 2. Delete all contents.

__ 3. Add this code to create the form.

```
<form ngForm #myForm="ngForm">
  VIN:<br/>
  <input type="text" name="veh_vin" ngModel/><br/>
  Year:<br/>
  <input type="number" name="veh_year" ngModel/><br/>
  Make:<br/>
  <input type="text" name="veh_make" ngModel/><br/>
  Model:<br/>
  <input type="text" name="veh_model" ngModel/><br/>
  Mileage:<br/>
  <input type="number" name="veh_mileage" ngModel/><br/>
  Price:<br/>
  <input type="number" name="veh_price" ngModel/><br/>
  <label>
    <input type="checkbox" name="veh_featured" ngModel/>
    Featured vehicle
  </label>
<br/>

  <button (click)="handleSubmit(myForm)">Submit</button>

</form>
```

Note these things:

- We used ngForm directive with the <form>.
- We used ngModel with the input elements.
- Year, price and mileage inputs are of type number. The browser will only allow

number input and Angular will convert the text to number.

__ 4. Save.

Part 7 - Use the Form

The dealer inventory component will now use this form to add a new vehicle to the inventory.

__ 1. Open **dealer-inventory.component.ts**.

__ 2. Add this method.

```
addVehicle(v:Vehicle) {  
    this.inventory.push(v)  
}
```

__ 3. Save.

__ 4. Open **dealer-inventory.component.html**.

__ 5. Below the line:

```
<h1>Current Inventory</h1>
```

Add:

```
<app-vehicle-form (on-submit)="addVehicle($event)"></app-vehicle-form>
```

Recall that the "on-submit" event carries a Vehicle object as payload. We are accessing that using the \$event variable.

__ 6. Save.

Part 8 - Test

__ 1. From VS Code terminal run.

```
npm start
```

__ 2. Open a browser and go to **http://localhost:4200/**.

__ 3. Make sure the form shows up.

__ 4. Enter some valid input like below.

VIN:

Year:

Make:

Model:

Mileage:

Price:

☐ Featured vehicle

__5. Click the **Submit** button.

__6. Scroll down the vehicle list and make sure that the car shows up at the very end.

2018 Honda, Civic

VIN: V0101

15000 Miles, \$23000

Part 9 - Add Basic Validation

__1. Open **vehicle-form.component.html**.

__2. For the VIN number input text, add these validators shown in bold face.

```
<input type="text" name="veh_vin" ngModel required minlength="3"/>>
```

__3. Similarly, add validators for the following inputs.

- Year: required
- Make: required
- Model: required
- Mileage: required

- Price: required

Min/max Validation

You may be tempted to use min/max validation for the price and mileage fields. But Angular does not support these validators for template driven form. Please see issue <https://github.com/angular/angular/issues/16352> for details.

__4. Display an error message above the button if the form is invalid and disable the submit button.

```
<p *ngIf="myForm.invalid" class="error">Please enter valid values.</p>
```

```
<button (click)="handleSubmit(myForm)"  
[disabled]="myForm.invalid">Submit</button>
```

__5. Save.

__6. Open **vehicle-form.component.css**.

__7. Add this CSS class.

```
.error {  
    color: red  
}
```

__8. Save.

Part 10 - Test

__1. Verify that by default the form is invalid and the error message shows and the button is disabled.

Please enter valid values.

Submit

__2. Enter valid values in all fields. Now the button should become enabled.

Part 11 - Field Specific Error Message

It is helpful if we can show an error message right next to an invalid input field. But we should do so only after the user has started entering data there.

In this lab we will only configure the VIN input field to show error. But if you have time you can do this for all the other fields.

__1. Open **vehicle-form.component.html**.

__2. Add a template variable for the `ngModel` directive applied to the VIN input field.

```
<input type="text" name="veh_vin" #vin="ngModel" ngModel required  
minlength="3"/>
```

__3. Add the error message after the end of the input tag.

```
<input type="text" name="veh_vin" #vin="ngModel" ngModel required  
minlength="3"/>  
<span class="error" *ngIf="vin.invalid && vin.dirty">Must be 3  
characters</span>  
<br/>
```

__4. Save.

__5. Test and make sure the error message is working.

VIN:

Must be 3 characters

Year:

Part 12 - Review

In this lab we built a template driven form. We did that in a separate component so that we can reuse it to edit a vehicle in a future lab. We learned:

- How to gather user input from `ngForm` directive.
- How to validate user input and show error message.

Lab 7 - Create an Edit Form

We will now modify the vehicle form component to add support for editing vehicle data. An edit form is special in the sense that it needs to be pre-populated with the existing data. We will achieve that using the [ngValue] binding.

Part 1 - Write the Component

The vehicle form component needs to receive as input the vehicle object that will be edited.

__1. Open **vehicle-form.component.ts**.

__2. Add this import.

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

__3. Add an **@Input** for the vehicle that will be edited.

```
@Input()
vehicle = new Vehicle("", 0, "", "", 0, 0, false, [])
```

Important: Here we are creating a default value for the vehicle variable. This makes the input optional. In fact when we use the form to add a new vehicle there is no need to supply a vehicle object as input.

__4. Save.

Part 2 - Pre-populate the Form

We will now initialize the form with values of the vehicle member variable.

__1. Open **vehicle-form.component.html**.

__2. For the VIN number input field set the initial value like shown in bold face below.

```
<input type="text" name="veh_vin" #vin="ngModel"
 [ngModel]="vehicle.VIN" required minlength="3"/>
```

__3. For the year field it will be like this.

```
<input type="number" name="veh_year" [ngModel]="vehicle.year" required/
>
```

__4. Similarly bind the ngModel attribute for all other fields.

__ 5. Save changes.

Part 3 - Use the Edit Form

__ 1. Open **dealer-inventory.component.ts**.

__ 2. Add a member variable as follows. This will keep track of the vehicle user is editing at any point in time.

```
vehicleToEdit?:Vehicle
```

__ 3. Add these methods.

```
beginEditing(v:Vehicle) {  
    this.vehicleToEdit = v  
}  
  
commitEdit(v:Vehicle) {  
    //Copy the edited data  
    Object.assign(this.vehicleToEdit, v)  
  
    this.vehicleToEdit = undefined  
}
```

__ 4. Save changes.

__ 5. Open **dealer-inventory.component.html**.

__ 6. Below the line:

```
<p>{{car.mileage}} Miles, ${{car.price}}</p>
```

Add this code to show the edit form:

```
<app-vehicle-form  
    *ngIf="vehicleToEdit == car"  
    [vehicle]="car"  
    (on-submit)="commitEdit($event)">  
</app-vehicle-form>
```

__ 7. Below the Delete button add the Edit button as shown in bold face below.

```
<button (click)="deleteVehicle(car)">Delete</button>  
<button (click)="beginEditing(car)">Edit</b>>
```

__ 8. Save.

Part 4 - Test

2012 HONDA, Civic

VIN: Y123

70000 Miles, \$5900

Delete

Edit

- __1. Click the Edit button for HONDA Civic.
- __2. Verify that the edit form shows and all fields are accurately initialized.

VIN:

Y123

Year:

2012

Make:

HONDA

Model:

Civic

- __3. Make some changes and submit the form.
- __4. Verify that the list now shows updated data.
- __5. Verify that validation is still working for the edit form.

Part 5 - Extra Credit - Add a Cancel Button to the Edit Form

Hints:

- Add a boolean `@Input` to the form component to optionally show a cancel button. The default value should be false. Show the cancel button only if the option is set to true.
- If the cancel button is clicked then raise a "on-cancel" custom event.
- From the dealer inventory component choose to display the cancel button for the edit form (and not the add form).

- Handle the "on-cancel" event and simply set vehicleToEdit variable to undefined. This will close the edit form and won't commit the changes.

Part 6 - Full Two-way Data Binding

Right now we are getting user input using the **NgForm.value** object. Another way to do this is to use two-way data binding. This will not add any new feature to our application. In fact we will see a defect happening.

__1. Commit all changes to Git before going forward. This will help you discard the work we are about to do.

__2. Open **vehicle-form.component.html**.

__3. For all occurrences of [ngModel] change it to [(ngModel)]. For example:

```
<input type="text" name="veh_vin" #vin="ngModel"
[(ngModel)]="vehicle.VIN" required minlength="3"/>
```

This will do two things:

- Angular will pre-populate the text box with the current value of vehicle.VIN.
- As the user types text Angular will save user input in vehicle.VIN.

__4. Make sure you have converted all ngModel to two-way binding.

__5. Save.

__6. Open **vehicle-form.component.ts**.

__7. Change the handleSubmit() method to be this.

```
handleSubmit(nForm:NgForm) {
  this.emitter.emit(this.vehicle)
}
```

Because of two-way binding Angular saves all user input in the vehicle variable. We can simply supply that as the event payload.

__8. Save.

__9. Use the add form and make sure you can add a new vehicle same as before.

__10. Now use the edit form. Notice something interesting happening.

2019 HERO, Civic

VIN: Y122

70000 Miles, \$5900

VIN:

Y122

Year:

2019

Make:

HERO

Model:

As you type in the edit form the vehicle information above changes in real time. You don't have to hit the Submit button. This may or may not be what you want. For example, a cancel button will have no use here since we are committing all changes in real time.

Using `NgForm.value` gathers user input outside your application's model (component state). This lets us cancel a form. Two-way binding directly updates application state. There's no easy to to cancel the form.

__11. Feel free to discard changes made in this part. To do that go to the source control view. Then click the ... icon and select **Discard All Changes**.



Part 7 - Review

In this lab we reused the vehicle form component to edit a vehicle. The key thing we learned is how to pre-populate a form using `[ngModel]`. We also learned how to use full two-way data binding as an alternative to `NgForm.value` to obtain user input.

Lab 8 - Reactive Form

We will now re-write the vehicle data entry form using reactive forms API. The main advantages of reactive forms are:

- Much easier to unit test as we will see later.
- Easier to write custom validators.

Part 1 - Get Started

- __ 1. From VS Code open the **C:\LabWork\my-test-app** folder if not already open.
- __ 2. From VS Code open a terminal if not already open.
- __ 3. If the development server is running in the terminal close it by hitting Control+C.
- __ 4. Commit all previous code changes to Git.

Part 2 - Create the Component

- __ 1. From VS Code terminal run:

```
ng g c vehicle-form-reactive
```

Part 3 - Import ReactiveFormsModule

API used by reactive forms is available from the ReactiveFormsModule. We will import it now.

- __ 1. In VS Code open **app.module.ts**
- __ 2. Add this import statement at the top.

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

- __ 3. Add the ReactiveFormsModule to the imports list of the NgModule decorator.

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

- __ 4. Save changes.

Part 4 - Write the Component

__1. Open **vehicle-form-reactive.component.ts**.

__2. Add these import statement at the top.

```
import { Component, Input, Output, EventEmitter, OnInit } from
'@angular/core';
import { FormGroup, FormControl, Validators, AbstractControl } from
'@angular/forms';
import { Vehicle } from '../vehicle';
```

__3. Add a member variable for the form to the class.

```
vehicleForm !:FormGroup
```

__4. Add these two member variables. They can be copied from the template driven form class.

```
@Input()
vehicle = new Vehicle("", 0, "", "", 0, 0, false, [])

@Output("on-submit")
emitter = new EventEmitter
```

__5. In the **ngOnInit()** lifecycle callback method create the form.

```
ngOnInit(): void {
  this.vehicleForm = new FormGroup({
    veh_vin: new FormControl(
      this.vehicle.VIN, [Validators.minLength(3), Validators.required]),
    veh_year: new FormControl(
      this.vehicle.year, [Validators.required]),
    veh_make: new FormControl(
      this.vehicle.make, Validators.required),
    veh_model: new FormControl(
      this.vehicle.model, Validators.required),
    veh_mileage: new FormControl(
      this.vehicle.mileage, Validators.required),
    veh_price: new FormControl(
      this.vehicle.price, [Validators.required, Validators.min(100)]),
    veh_featured: new FormControl(
      this.vehicle.featured),
  })
}
```

Note that we set the current values of the vehicle properties as the initial values of the input controls.

__6. Add the form submission handler method.

```
handleSubmit() {
  const input = this.vehicleForm.value
  const v = new Vehicle(
    input.veh_vin,
    input.veh_year,
    input.veh_make,
    input.veh_model,
    input.veh_mileage,
    input.veh_price,
    input.veh_featured === "" ? false : input.veh_featured,
    [])

  this.emitter.emit(v)
}
```

__7. Save.

Part 5 - Write the Template

__1. Open `vehicle-form-reactive.component.html`.

__2. Delete all content.

__3. Enter this code to create the form.

```
<form [formGroup]="vehicleForm">
  VIN:<br />
  <input type="text" formControlName="veh_vin" />
  <span class="error" *ngIf="vehicleForm.controls.veh_vin.invalid &&
vehicleForm.controls.veh_vin.dirty">Must be 3 characters</span>
  <br />
  Year:<br />
  <input type="text" formControlName="veh_year"/><br />
  Make:<br />
  <input type="text" formControlName="veh_make"/><br />
  Model:<br />
  <input type="text" formControlName="veh_model"/><br />
  Mileage:<br />
  <input type="text" formControlName="veh_mileage"/><br />
  Price:<br />
  <input type="text" formControlName="veh_price"/><br />
  <label>
    <input type="checkbox" formControlName="veh_featured"/>
    Featured vehicle
  </label>
<br/>
```

```

    <p *ngIf="vehicleForm.invalid" class="error">Please enter valid
values.</p>
    <button (click)="handleSubmit()"
[disabled]="vehicleForm.invalid">Submit</button>

</form>

```

Note that all validation rules and initial values of the inputs are now in Typescript code and not in the template.

__ 4. Save.

Part 6 - Write CSS of the Component

__ 1. Open **vehicle-form-reactive.component.css**.

__ 2. Add this class.

```

.error {
    color: red
}

```

__ 3. Save.

Part 7 - Use the Component

__ 1. Open **dealer-inventory.component.html**.

__ 2. Simply replace all occurrences of "app-vehicle-form" with "app-vehicle-form-reactive". Since the reactive component works with the same exact @Input and @Output names we don't have to change anything else. The add form will be:

```

<app-vehicle-form-reactive (on-submit)="addVehicle($event)">
</app-vehicle-form-reactive>

```

And the edit form will be:

```

<app-vehicle-form-reactive
    *ngIf="vehicleToEdit == car"
    [vehicle]="car"
    (on-submit)="commitEdit($event)">
</app-vehicle-form-reactive>

```

__ 3. Save.

Part 8 - Test

- ___1. From VS Code terminal start the development server.
- ___2. Open the application in a browser.
- ___3. Verify that all aspects the application works same as before. This includes:
 - Adding a vehicle
 - Editing a vehicle
 - Input validation
 - Deleting a vehicle

Part 9 - Write a Custom Validator

We will now write a custom validator for the VIN number. Among other restrictions a VIN number can not have the I, O and Q characters to avoid confusion with the numbers 1 and 0. We will just check for that.

- ___1. Open **vehicle-form-reactive.component.ts**.
- ___2. Above the line.

```
@Component ({
```

Add:

```
function validate_VIN(control:AbstractControl) {  
  const banned = ["I", "O", "Q"]  
  
  if (banned.some(str => control.value.includes(str))) {  
    return {  
      error: "Must not contain I, O or Q"  
    }  
  } else {  
    return null //All good  
  }  
}
```

This is a standalone function and must be outside the class definition.

- ___3. Add the custom validator to the list of validators for the VIN number field.

```
veh_vin: new FormControl(this.vehicle.VIN, [validate_VIN,  
Validators.minLength(3), Validators.required]),
```


__ 4. Save.

__ 5. Open **vehicle-form-reactive.component.html**.

__ 6. Make the error message for VIN a bit broader.

```
<span class="error" *ngIf="vehicleForm.controls.veh_vin.invalid &&
vehicleForm.controls.veh_vin.dirty">Must be minimum 3 characters. I, O
and Q not allowed.</span>
```

__ 7. Save.

__ 8. Test to make sure the new validation rule is working.

VIN:

V234QW| **Must be minimum 3 characters. I, O and Q not allowed.**

Part 10 - Review

In this lab we built a reactive form. Many find reactive forms easier to work with than template driven forms. Your opinion may vary. Reactive forms are certainly easier to unit test. Writing a custom validator is also pretty easy.

Lab 9 - Develop a Service

So far the dealer inventory component has been keeping track of the inventory. It also has logic to add, update and delete vehicles. Such business logic should really belong to a service. We will now migrate those responsibilities to a service.

Part 1 - Get Started

- ___ 1. From VS Code open the **C:\LabWork\my-test-app** folder if not already open.
- ___ 2. From VS Code open a terminal if not already open.
- ___ 3. If the development server is running in the terminal close it by hitting Control+C.
- ___ 4. Commit all previous code changes to Git.

Part 2 - Create the Service

- ___ 1. From the VS Code terminal run:

```
ng generate service inventory
```

Part 3 - Write the Service

- ___ 1. Open **src/app/inventory.service.ts**.
- ___ 2. Add this import.
- ___ 3. Add a member variable to the **InventoryService** class.

```
import { Vehicle } from './vehicle';
```

```
private inventory: Vehicle[] = []
```

- ___ 4. Add these methods. Note we mark them as public so that they can be called from outside this class.

```
public getInventory() : Vehicle[] {  
    return this.inventory  
}  
  
public addVehicle(v:Vehicle) {  
    this.inventory.push(v)  
}
```

```

public updateVehicle(oldVIN:string, newVehicle:Vehicle) {
  const oldVehicle = this.inventory.find(
    v => v.VIN === oldVIN)

  if (oldVehicle !== undefined) {
    Object.assign(oldVehicle, newVehicle)
  }
}

public deleteVehicle(vehicleToDelete:Vehicle) {
  this.inventory = this.inventory.filter(v => v.VIN !==
vehicleToDelete.VIN)
}

```

__5. Save.

Part 4 - Use the Service

__1. Open `dealer-inventory.component.ts`.

__2. Add this import.

```
import {InventoryService} from '../inventory.service'
```

__3. Remove all the hard coded inventory data. The inventory variable will now be like this.

```
inventory:Vehicle[] = []
```

__4. Inject the service in the constructor.

```
constructor(private inventorySvc:InventoryService) { }
```

__5. In `ngOnInit()` get the inventory from the service.

```

ngOnInit(): void {
  this.inventory = this.inventorySvc.getInventory()
}

```

__6. Change the deleteVehicle() method as follows.

```
deleteVehicle(car:Vehicle) {  
    this.inventorySvc.deleteVehicle(car)  
    this.inventory = this.inventorySvc.getInventory()  
}
```

__7. Change the addVehicle() method like this.

```
addVehicle(v:Vehicle) {  
    this.inventorySvc.addVehicle(v)  
    this.inventory = this.inventorySvc.getInventory()  
}
```

__8. Change the commitEdit() method as follows.

```
commitEdit(v:Vehicle) {  
    this.inventorySvc.updateVehicle(this.vehicleToEdit!.VIN, v)  
    this.inventory = this.inventorySvc.getInventory()  
  
    this.vehicleToEdit = undefined  
}
```

__9. Save.

Part 5 - Test

__1. From VS Code terminal start the development server.

__2. Open the application in a browser. Now the application will start off with empty inventory.

__3. Verify that you can add, edit and delete cars.

Part 6 - Review

In this lab we migrated business logic from a component to a service. This has several advantages:

- The business logic can now be easily unit tested.
- A service can be used by any component throughout the application. This makes the code more reusable.

Lab 10 - Develop an HTTP Client

Right now the InventoryService has the inventory data maintained locally in memory. In real life the data will probably be maintained in a database somewhere and exposed via a set of web service calls. In this lab we will do exactly that.

Part 1 - Understand the Backend Web Service

__ 1. Open a command prompt window.

__ 2. Change directory to **C:\LabFiles\inventory-server**.

__ 3. Run this command.

```
npm install
```

__ 4. Start the web service server.

```
npm start
```

__ 5. Make sure you see the message:

```
Service endpoint base: http://localhost:3000/vehicle
```

__ 6. Take a moment to read through the API specification.

Endpoint	Comments
GET /vehicle	Get an array of all vehicles in inventory.
POST /vehicle	Add a vehicle. The new vehicle is sent as the JSON body.
PUT /vehicle/:VIN	Update a vehicle. Updated data is sent as the JSON body.
DELETE /vehicle/:VIN	Delete a vehicle.

In addition the service supports Cross Origin Resource Sharing (CORS). Which means a web page from any origin is will be allowed by the browser to call into this web service.

Part 2 - Import the HTTP Client Module

To inject the HttpClient service we need to import the HttpClientModule.

__1. Open **app.module.ts**

__2. Add the import of the HttpClientModule and include it in the imports of the @NgModule declaration.

```
import { HttpClientModule }    from '@angular/common/http';

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

__3. Save changes.

Part 3 - Use the HttpClient Service

We will now use the HttpClient service from InventoryService to call a web service.

__1. Open **src/app/inventory.service.ts**.

__2. Add various import statements.

```
import {Injectable} from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
```

__3. In the constructor method inject the HttpClient service.

```
constructor(private httpClient:HttpClient) {}
```

__4. Delete the **inventory** array variable. We wont need it any more.

```
private inventory: Vehicle[] = []
```

__5. Add this member variable.

```
private baseUrl = "http://localhost:3000/vehicle"
```

__6. Re-write the **getInventory()** method like this.

```
getInventory(): Observable<Vehicle[]> {  
    return this.httpClient.get<Vehicle[]>(`${this.baseUrl}`)  
}
```

__7. Similarly re-write all the other methods like this.

```
public addVehicle(v:Vehicle): Observable<any> {  
    return this.httpClient.post(`${this.baseUrl}`, v)  
}  
  
public updateVehicle(vin:string, v:Vehicle): Observable<any> {  
    return this.httpClient.put(`${this.baseUrl}/${vin}`, v)  
}  
  
public deleteVehicle(v:Vehicle): Observable<any> {  
    return this.httpClient.delete(  
        `${this.baseUrl}/${v.VIN}`  
    )  
}
```

__8. Save changes.

Part 4 - Modify the Dealer Inventory Component

__1. Open **dealer-inventory.component.ts**.

__2. Change the **ngOnInit()** method like this.

```
ngOnInit(): void {  
    this.inventorySvc.getInventory()  
        .subscribe(list => this.inventory = list)  
}
```

__3. Update the **deleteVehicle()** method.

```
deleteVehicle(car:Vehicle) {  
    this.inventorySvc.deleteVehicle(car).subscribe(() => {  
        //Update local copy of the list  
        this.inventory = this.inventory.filter(v => v.VIN !== car.VIN)  
    })  
}
```

Note we are deleting the vehicle from the local copy of the vehicle list after we successfully delete it from the backend. This is faster than reloading the whole list from the backend server.

__4. Change the **addVehicle()** method like this.

```
addVehicle(v:Vehicle) {  
    this.inventorySvc.addVehicle(v).subscribe(() => {  
        this.inventory.push(v)  
    })  
}
```

__5. Change the **commitEdit()** method like this.

```
commitEdit(v:Vehicle) {  
    this.inventorySvc.updateVehicle(this.vehicleToEdit!.VIN, v)  
        .subscribe(() => {  
            Object.assign(this.vehicleToEdit, v)  
            this.vehicleToEdit = undefined  
        })  
}
```

__6. Save changes.

__7. Verify that there are no compilation errors in VS Code terminal.

Part 5 - Test

__1. Start the development server if it is not running already.

__2. Open the application in a browser.

__3. Initially there should be no items in the inventory.

__4. Add a few new vehicles. Make sure they show up in the list.

__5. Refresh the web page. Make sure all added vehicles still show up. This proves that items got added to the backend successfully.

__6. Try out all the other operations like edit and delete. Make sure that all changes are permanent by refreshing the browser.

Part 6 - Review

In this lab we learned how to make calls to a backend web service.

Lab 11 - Use Pipes

In this short lab, we will use a few builtin pipes to format data. We will format the mileage and price fields.

Part 1 - Get Started

- ___ 1. From VS Code open the **C:\LabWork\my-test-app** folder if not already open.
- ___ 2. From VS Code open a terminal if not already open.
- ___ 3. If the development server is running in the terminal close it by hitting Control+C.
- ___ 4. Commit all previous code changes to Git.

Part 2 - Write the Component

- ___ 1. Open **dealer-inventory.component.html**.
- ___ 2. Change the line that shows mileage and price like this.

```
<p>{{car.mileage | number}} Miles, {{car.price | currency}}</p>
```

- ___ 3. Save.

Part 3 - Test

- ___ 1. Start development server.
- ___ 2. Start the inventory backend service if it is not running.
- ___ 3. Access the application in a browser.
- ___ 4. Add a few vehicles.
- ___ 5. Verify that the fields are now rendered like this.

2020 BMW, 328i

VIN: V1929

20,300 Miles, \$39,290.00

Part 4 - Change Currency

__1. By default the currency is US\$. That may not be what you want. You should explicitly specify currency. Change the currency to Euro.

```
{{car.price | currency: 'EUR'}}
```

__2. Save.

__3. Make sure the currency shown changes.

20,300 Miles, €39,290.00

Part 5 - Add International Support

In France, formatting is done like this:

- The thousand separator is a space
- Decimal separator is a comma.
- Currency symbol is shown after the number.

Let's add support for French in our application.

__1. Open **src/app/app.module.ts**.

__2. Add this import statement to load French locale data.

```
import '@angular/common/locales/global/fr';
```

__3. Save.

__4. Open **dealer-inventory.component.html**.

__5. Use the "fr" locale in the pipes.

```
<p>{{car.mileage | number:'':'fr'}} Miles, {{car.price |  
currency:'EUR':'symbol':'':'fr'}}</p>
```

__6. Save. Verify that the page now shows items like this.

20 300 Miles, 39 290,00 €

Part 6 - Extra Credit

Show values using French Canadian locale:

- Currency code is CAD.
- Locale is 'fr-CA'

Part 7 - Clean Up

- __1. Shutdown the development server.
- __2. Shutdown the inventory backend service.
- __3. Commit all code to Git.

Part 8 - Review

In this lab, we used the currency and number builtin pipes. We also learned how to support non-default locales.

Lab 12 - Basic Single Page Application Using Router

In this lab, we will develop a very simple single page application (SPA) using the Angular Component Router module. The main goal will be to understand how routing works. We will keep the business logic very simple.

Part 1 - Get Started

__1. In VS Code, if the development server is running in the terminal close it by hitting Control+C.

__2. Close the VS Code window (**File > Close Window**).

__3. Open a command prompt window.

__4. Go to the **C:\LabWork** folder.

__5. Run this command to create a new project called **route-test**.

```
ng new route-test --routing --defaults
```

__6. From VS Code, open the directory **C:\LabWork\route-test**.

Part 2 - The Business Logic

Simple SPA!

[Home](#) [News](#) [About](#)

Home

This is the Home page

We will develop an SPA that has three pages (views):

1. The home page. Shown by default.
2. The news page. Mapped to the "/news" URL.
3. The about page. Mapped to the "/about" URL.

After we develop this application we will be able to navigate between the pages without reloading the browser.

Part 3 - Create the Components

Each view in SPA is implemented by a separate component. These components can employ child components if needed. We will now create the components for our applications.

__ 1. From VS Code open a new terminal (**Terminal > New Terminal**).

__ 2. Run these commands to create the components.

```
ng g c home
ng g c about
ng g c news
```

__ 3. Open **src/app/home/home.component.html**

__ 4. Change the template like this:

```
<h2>Home</h2>
<p>This is the Home page</p>
```

__ 5. Save the file.

__ 6. Open **src/app/about/about.component.html**

__ 7. Change the template like this:

```
<h2>About</h2>
<p>This is the About page</p>
```

__ 8. Save the file.

__ 9. Open **src/app/news/news.component.html**

__ 10. Change the template like this:

```
<h2>News</h2>
<p>This is the News page</p>
```

__ 11. Save the file.

The selectors for these components do not really play any role. We never manually add these components to any template. The router system inserts these components for us based on the URL.

Part 4 - Define the Route Table

- ___1. Open **src/app/app-routing.module.ts**
- ___2. Add these import statements for the component classes.

```
import { HomeComponent } from './home/home.component';  
import { NewsComponent } from './news/news.component';  
import { AboutComponent } from './about/about.component';
```

- ___3. Set up the route table as shown in bold face below.

```
const routes: Routes = [  
  { path: '', component: HomeComponent },  
  { path: 'news', component: NewsComponent },  
  { path: 'about', component: AboutComponent }  
];
```

- ___4. Save changes.

Part 5 - Setup the Component Host

- ___1. Open **app.component.html**
- ___2. Delete all content.
- ___3. Add the code below:

```
<h1>Simple SPA!</h1>  
<a [routerLink]="['/']">Home</a>&nbsp;<br>  
<a [routerLink]="['/news']">News</a>&nbsp;<br>  
<a [routerLink]="['/about']">About</a>  
  
<div id=main>  
  <router-outlet></router-outlet>  
</div>
```

The App component will act as the root of the application. It will render HTML content that is shared by all pages in the app.

Note these key aspects of the code:

- We use the routerLink attribute directive to define navigational links. Here we are using links like "/news" and "/about". We will later map these to the corresponding components in a route table.
- Use <router-outlet> tag to define where the component for the pages should be inserted.

- __4. Save changes.
- __5. Edit **styles.css**
- __6. Add the following content and save the file:

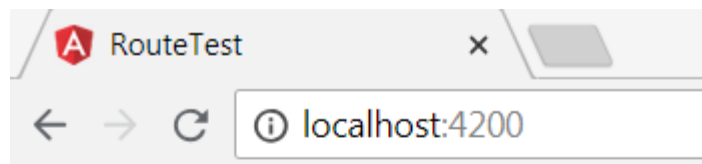
```
h2 {  
    margin:0px;  
}  
#main{  
    border: 1px solid grey;  
    width: 250px;  
    background-color: lightgray;  
    margin-top: 5px;  
    padding: 5px;  
}
```

- __7. Save changes.

Part 6 - Test

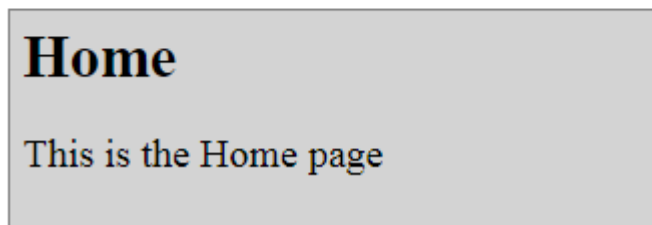
- __1. From VS Code terminal run:

`npm start`
- __2. Open a browser to **http://localhost:4200**
- __3. Verify that the home page is shown by default.



Simple SPA!

[Home](#) [News](#) [About](#)



- __4. Click the News and About links. Make sure that the corresponding pages are shown.

Verify that the URL in the browser's address bar changes as you navigate.

__ 5. Verify that the back button navigation works as expected.

__ 6. Navigate to the About page.

__ 7. Now refresh the browser. You should still see the About page.

Note: This behavior is primarily because the project uses the embedded server and all requests get routed to Angular. If you were using a web server that might first look for a local HTML file, refreshing the URL from the Router may not work and you may get a 404 error.

__ 8. In the Terminal , hit '<CTRL>-C' to terminate the embedded server.

__ 9. Close VS Code window.

__ 10. Close all.

Part 7 - Review

In this lab, we created a single page application with routing.

Lab 13 - Build a Single Page Application (SPA)

In this lab we will build a real life SPA.

Part 1 - Business Requirements

In this lab, we will create a book database management system that can be used in a public library. Our application will let users add, edit and delete books.

The back-end web service exposes this interface.

URL Path	HTTP Method	Notes
/books	GET	Returns an array of all books.
/books/ <i>ISBN</i>	GET	Returns a specific book given its ISBN. Example: /books/123-456
/books/ <i>ISBN</i>	DELETE	Deletes a specific book given its ISBN. Example: /books/123-456
/books/ <i>ISBN</i>	PUT	Adds or updates a book.

Our application will need to develop these pages.

Book Library

Add a Book
167 - Harry Potter **23.99** DELETE
136 - Linux in a Nutshell **14.99** DELETE

Book List Page

Add a Book

136
Linux in a Nutshell
14.99
<input type="button" value="Save"/>

Add Book Form

Update the Book

156
Linux in a Nutshell
14.95
<input type="button" value="Save"/>

Update Book Form

Part 2 - Get Started

We will now create a new Angular application.

- __1. Close any VS Code window if open.
- __2. Open a command prompt window.
- __3. Go to the **C:\LabWork** folder. Create the folder if it hasn't been created.
- __4. Run this command to create a new project called **rest-client**.

```
ng new rest-client --routing --defaults
```

- __5. Open the **C:\LabWork\rest-client** folder in VS Code.

Part 3 - Import Additional Modules

We will make web service calls using the HttpClient Angular service. This is available from the HttpClientModule. We need to import this module and a few others from our application module.

- __1. If you haven't already done so, open the **C:\LabWork\rest-client** folder in VS Code.
- __2. Open **src\app\app.module.ts**
- __3. Add these import statements.

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

- __4. Import the modules as shown in bold face below.

```
imports: [  
  BrowserModule,  
  AppRoutingModule,  
  HttpClientModule,  
  FormsModule  
],
```

- __5. Save changes.

Part 4 - Create the Service

We will now create an Angular service where we will isolate all web service access code. Generally speaking create the services before starting to work on components.

__1. In VS Code open a new terminal window.

__2. In the terminal run this command.

```
ng generate service data
```

This will create a service class called DataService in data.service.ts file.

__3. Open `src\app\data.service.ts`

__4. Add these import statements.

```
import { HttpClient } from '@angular/common/http';  
import { Observable } from 'rxjs';
```

__5. Add the Book class like this.

```
export class Book {  
  isbn!: string  
  title!: string  
  price!: number  
}
```

__6. In the constructor of the DataService class inject the HttpClient service.

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

__7. Save changes.

__8. Add the '**getBooks**' method to the DataService class as follows.

```
getBooks() : Observable<Book[]> {  
  return this.http.get<Book[]>("/books")  
}
```

This means the method returns an Observable that delivers an array of Book objects.

__9. Add the 'getBook' method.

```
getBook(isbn: string): Observable<Book> {  
    return this.http.get<Book>(`/books/${isbn}`)  
}
```

__10. Add the deleteBook() method as follows.

```
deleteBook(isbn: string): Observable<any> {  
    return this.http.delete(`/books/${isbn}`)  
}
```

Note: In this case we don't really care for the data type of the response from the server. As a result the method returns an Observable<any>.

__11. Add the 'saveBook' method as follows. We will use this to both add and update a book.

```
saveBook(book: Book): Observable<any> {  
    return this.http.put(`/books/${book.isbn}`, book)  
}
```

__12. Save changes.

Part 5 - Generate the Components

__1. In the VS Code terminal, run these commands to create a component for each page in our application.

```
ng g c book-list  
ng g c add-book  
ng g c edit-book
```

Part 6 - Setup the Route Table

__1. Open app-routing.module.ts

__2. Import the component class names.

```
import { BookListComponent } from '../book-list/book-list.component';
import { AddBookComponent } from '../add-book/add-book.component';
import { EditBookComponent } from '../edit-book/edit-book.component';
```

__3. Setup the route table as shown in bold face below.

```
const routes: Routes = [
  {path: "", component: BookListComponent},
  {path: "add-book", component: AddBookComponent},
  {path: "edit-book/:isbn", component: EditBookComponent},
];
```

Note a few things:

1. EditBookComponent receives the ISBN number of the book as a path parameter.
2. Never start a path with a "/".
3. BookListComponent has the default path (empty string).

__4. Save changes.

__5. Open **src/app/app.component.html**

__6. Delete all lines.

__7. Add this line.

```
<router-outlet></router-outlet>
```

The routed components will be rendered there.

__8. Save changes.

Part 7 - Write the BookListComponent

__1. Open **src/app/book-list/book-list.component.ts**

__2. Add this import statement.

```
import { DataService, Book } from '../../data.service'
```

___3. Inject the service from the constructor.

```
constructor(private dataService: DataService) { }
```

___4. Add a member variable to the class that will store the list of books.

```
books:Book[] = []
```

___5. From the `ngOnInit()` method fetch the books as shown in bold face below.

```
ngOnInit() {  
    this.dataService.getBooks().subscribe(bookList => {  
        this.books = bookList  
    })  
}
```

The **`dataService.getBooks()`** method returns an `Observable<Book[]>`. We must subscribe to this Observable to get the data.

___6. Save changes.

We will now work on the template of the component.

___7. Open **book-list.component.html**

___8. Set the contents of the file like this.

```
<h3>Book Library</h3>  
  
<div *ngIf="books.length > 0">  
    <div *ngFor="let book of books">  
        {{book.isbn}} - {{book.title}} <b>{{book.price}}</b>  
    </div>  
</div>  
  
<div *ngIf="books.length == 0">  
    There are no books in the library.  
</div>
```

___9. Save changes.

Before we develop the other components we should do a test and make sure whatever we have so far works. But we have to do a bit of administration work before we can do any testing.

Part 8 - Setup Reverse Proxy

The index.html file for our application is served by the Angular CLI web server. But the backend web service will run on another server (port 3000). If our application tries to directly call the web service, it will violate the same origin policy. This web service does not support CORS. We need to setup Angular CLI as a reverse proxy for the web service calls. This way all HTTP requests from the browser will first go to the Angular CLI server. If the request is for a web service (/books/*) the Angular CLI server will forward it to the backend web service.

__1. Copy the file:

```
proxy.config.json
from: C:\LabFiles\rest-client\
to: C:\LabWork\rest-client\
```

__2. Edit the **package.json** file from **C:\LabWork\rest-client**, replace the start line in the scripts section with the following and save the file:

```
"start": "ng serve --proxy-config proxy.config.json",
```

__3. Save changes.

Part 9 - Start the Servers

__1. The backend web service is available in the **C:\LabFiles\book-server** directory. It is written in Node.js. But in real life the backend web service can be written in anything. To start it open a command prompt and navigate to the **C:\LabFiles\book-server** directory. Then execute the following command:

```
npm start
```

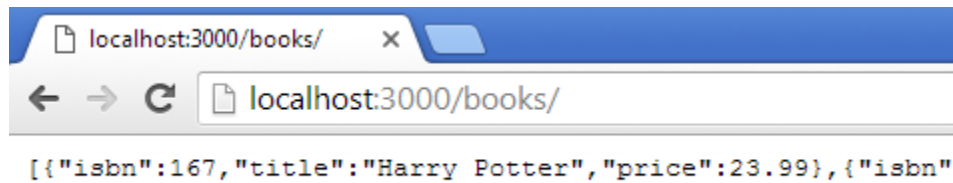
__2. To start the Angular development server run this from the VS Code terminal:

```
npm start
```

__3. Test to make sure the REST server is working by opening a browser and entering the following address:

```
http://localhost:3000/books/
```

You should see some JSON data show up in the browser.



__4. Next test to make sure that the reverse proxy configuration for the REST server is working by opening a browser and entering this address:

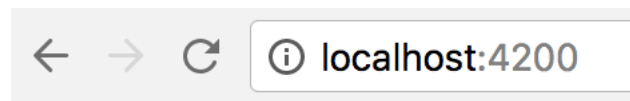
`http://localhost:4200/books/`

You should get the same results as before - some JSON data should show up in the browser.

__5. Finally check to see that the Angular application is coming up by opening the following URL in the browser:

`http://localhost:4200`

You should see the following in the browser:



Book Library

167 - Harry Potter **23.99**
267 - Lord of the Rings **25.49**

Part 10 - Implement Book Removal

__1. Open `src/app/book-list/book-list.component.ts`

__2. Add the deleteBook() method like this.

```
deleteBook(book: Book) {  
  if (!window.confirm('Are you sure you want to delete this item?')) {  
    return  
  }  
  
  this.dataService.deleteBook(book.isbn).subscribe(_ => {  
    //Delete local copy of the book  
    this.books = this.books.filter(b => b.isbn !== book.isbn)  
  })  
}
```

__3. Save changes.

__4. Open **book-list.component.html**

__5. Add a DELETE button as shown in bold face.

```
<div *ngFor="let book of books">  
  {{book.isbn}} - {{book.title}} <b>{{book.price}}</b>  
  <button (click)="deleteBook(book)">DELETE</button>  
</div>
```

__6. Save changes.

__7. Return to the browser. It should still display the same list of books.

__8. Click on the '**DELETE**' button next to one of the entries in the list and then click on the '**OK**' button on the confirmation dialog to confirm you want to delete the item.

__9. Refresh the page to make sure the book is removed from the list by the backend service.

Only delete one book. It is best to avoid deleting the last book if possible.

Note: If while testing you no longer have any books in the list, you can always return to the REST service command prompt and use 'CTRL-C' to stop running the REST Server. Restart the server with 'npm run start' and the initial list of books will be restored.

Part 11 - Implement AddBookComponent

__1. Open **src/app/add-book/add-book.component.ts**

__2. Add these import statements.

```
import { DataService, Book } from '../data.service'  
import { Router } from '@angular/router'
```

__3. Inject the services from the constructor.

```
constructor(private dataService: DataService, private router: Router)  
{ }
```

__4. Add a member variable to the class that will store user's input for a book.

```
book:Book = new Book
```

__5. Write in the addBook() method like this.

```
addBook() {  
  this.dataService.saveBook(this.book).subscribe(_ => {  
    //Go back to the home page  
    this.router.navigate(['/'])  
  })  
}
```

__6. Save changes.

We will now develop the form.

__7. Open **src/app/add-book/add-book.component.html**

__8. Develop the form like this.

```
<h3>Add a Book</h3>  
  
<div>  
  <input type="text" [(ngModel)]="book.isbn" placeholder="ISBN">  
</div>  
<div>  
  <input type="text" [(ngModel)]="book.title" placeholder="Title">  
</div>  
<div>  
  <input type="number" [(ngModel)]="book.price" placeholder="Price">  
</div>  
<div>  
  <button (click)="addBook()">Save</button>  
</div>
```

Note, here we are using two-way data binding to obtain user input.

__9. Save changes.

We now have to add a button the BookListComponent to navigate to the AddBookComponent.

__10. Open **src/app/book-list/book-list.component.html**

__11. Add a button to go to the add book form as shown in bold face below.

```
<h3>Book Library</h3>
<div>
  <button [routerLink]="['/add-book']">Add a Book</button>
</div>
```

__12. Save changes.

__13. Return to the command prompt running the project and check that the files are compiled without any errors.

Part 12 - Test Changes

__1. Return to the browser and refresh the main URL of the application. It should still display the same list of books.

http://localhost:4200/

__2. Click on the '**Add a Book**' button.

__3. Enter some data like this.

Add a Book

136
Linux in a Nutshell
14.99
Save

__4. Click the '**Save**' button to submit the data.

__5. Verify that the book list page is shown again with the newly added book in the list.

Book Library

Add a Book

167 - Harry Potter **23.99** DELETE

136 - Linux in a Nutshell **14.99** DELETE

__6. Add a few books. Use the browser's network developer tool to verify that only Ajax (XHR) requests are taking place. There should be no page reload (document load) in a single page application.

Part 13 - Implement EditBookComponent

__1. Open `src/app/edit-book/edit-book.component.ts`

__2. Add these import statements.

```
import { DataService, Book } from '../data.service'  
import { ActivatedRoute, Router } from '@angular/router'
```

__3. Inject the services from the constructor.

```
constructor(private dataService: DataService,  
             private activeRoute: ActivatedRoute,  
             private router: Router) { }
```

__4. Add a member variable to the class that will store information about the book we are editing.

```
book!: Book
```

This component will need to retrieve the ISBN number from the path parameter and then fetch the book by calling into the backend web service. We will do that from the `ngOnInit()` method.

___5. Write the ngOnInit() method as shown in bold face below.

```
ngOnInit() {  
  this.activeRoute.paramMap.subscribe(params => {  
    let isbn = params.get('isbn')!  
  
    this.dataService.getBook(isbn).subscribe(book => {  
      this.book = book  
    })  
  })  
}
```

___6. Write in the updateBook() method like this.

```
updateBook() {  
  this.dataService.saveBook(this.book).subscribe(_ => {  
    //Go back to the home page  
    this.router.navigate(['/'])  
  })  
}
```

___7. Save changes.

We will now develop the form.

___8. Open **src/app/edit-book/edit-book.component.html**

___9. Develop the form like this.

```
<h3>Update the Book</h3>  
  
<div *ngIf="book != undefined">  
  <div>  
    <input type="text" [(ngModel)]="book.isbn" placeholder="ISBN"  
disabled>  
  </div>  
  <div>  
    <input type="text" [(ngModel)]="book.title" placeholder="Title">  
  </div>  
  <div>  
    <input type="number" [(ngModel)]="book.price" placeholder="Price">  
  </div>  
  <div>  
    <button (click)="updateBook()">Save</button>  
  </div>  
</div>
```

___10. Save changes.

Finally, we need to add a button for each book in the book list page to go to the edit page.

___11. Open **src/app/book-list/book-list.component.html**

__12. Add the EDIT button below the DELETE button as shown in bold face.

```
<button (click)="deleteBook(book)">DELETE</button>  
<button [routerLink]='['/edit-book', book.isbn]'>EDIT</button>
```

Notice how we supply the ISBN path parameter in the link.

__13. Save changes.

Part 14 - Test

__1. Back in the browser you should now see the EDIT button for each book.

Book Library

Add a Book

156 - Linux in a Nutshell **14.95** DELETE **EDIT**

__2. Click the **EDIT** button.

__3. Verify that the form is pre-populated with the book's data.

Update the Book

156

Linux in a Nutshell

14.95

Save

__4. Change the title and the price and click **Save**.

__5. Verify that the changes are reflected in the book list.

Part 15 - Clean up

__1. In the REST Server Application command prompt press 'CTRL-C' to stop the server.

__2. Hit Control+C to close the Angular dev server.

__3. Close all open text editors and browser windows.

Part 16 - Review

In this lab you implemented a fairly realistic application that calls a back-end web service and uses routing.

Lab 14 - Advanced HTTP Client

In this lab we will use some of more advanced aspects of HttpClient and RxJS API to:

- Show error messages when a HTTP call fails.
- Implement caching of data.
- Recover from error by returning cached data.
- Attempt to recover from error by retrying HTTP calls.

We will continue to the book database application and add these features there.

Part 1 - Add Error Handling

Right now our DataService class is not handling any kind of errors that may happen when invoking a web service. There are two types of errors we have to worry about:

- There may be a business rule violation in the server. For example, you are trying to delete a book that doesn't exist. These errors are reported by the web service by setting a 4XX or 5XX response status code.
- There may be a network issue and the front end is simply not able to connect to the server. These are reported by HttpClient by raising an error with the Observable.

We will learn to handle both errors.

__1. In VS Code open **book-list.component.ts**

__2. In the deleteBook() method supply a second argument to the subscribe() method call. This is an arrow function that gets called if the response code indicates an error.

```
this.dataService.deleteBook(book.isbn).subscribe(_ => {  
    //Delete local copy of the book  
    this.books = this.books.filter(b => b.isbn !== book.isbn)  
},  
err => {  
    alert("Oops! There was a problem at the server.")  
})
```

__3. Save changes.

Let's test this out by first simulating a status code error.

__4. Open **data.service.ts**

__5. In the deleteBook() method deliberately create a problem by entering a bad URL. This will cause 404 to be returned by the web service.

```
deleteBook(book: Book): Observable<any> {  
    return this.http.delete(`/books/bad-url/${isbn}`)  
}
```

__6. Save changes.

Part 2 - Start the Servers

__1. The backend web service is available in the **C:\LabFiles\book-server** directory. To start it open a command prompt and navigate to the **C:\LabFiles\book-server** directory. Then execute the following command:

```
npm start
```

__2. To start the Angular application server run this from VS Code terminal:

```
npm start
```

Part 3 - Test

__1. Open a browser to:

```
http://localhost:4200
```

__2. Try to delete a book. You should see the alert error message.

__3. Fix the error introduced in **data.service.ts**.

__4. Save changes.

Now, let's simulate a network communication error.

__5. Shutdown the Angular CLI server (not the REST server) by going to the VS Code terminal and pressing 'CTRL-C'.

__6. Back in the browser try to delete a book. You will get the error alert once again.

Part 4 - Custom Error Handling

Right now the component is deciding what error message to display. In some complex cases the message may depend on exactly what went wrong. That kind of logic should be isolated in a service. The service should simply hand an error message to the component which will render in some form to the user.

We can intercept any error happening to an Observable using the **catchError** operator. We can then create a new Observable with a custom error object using the **throwError** operator.

__ 1. Open **data.service.ts**

__ 2. Import the **HttpErrorResponse** class.

```
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
```

__ 3. Import the **throwError** operator like shown in bold face below. Operators that create a new Observable are available from the 'rxjs' module.

```
import { Observable, throwError } from 'rxjs';
```

__ 4. Import the **catchError** operator.

```
import { catchError } from 'rxjs/operators';
```

__ 5. In the **deleteBook()** method use the **catchError** operator to deal with an error raised for the Observable. Operator functions are supplied to the **pipe()** method as a comma separated list.

```
deleteBook(isbn: string): Observable<any> {  
  return this.http.delete(`/books/bad-url/${isbn}`) .pipe(  
    catchError((err:HttpErrorResponse) => {  
      if (err.status == 0) {  
        return throwError("Oops! Please check your network connection  
and try again.")  
      } else {  
        return throwError("Sorry there was a problem at the server.")  
      }  
    })  
  )  
}
```

This shows how the service is constructing a different error message depending on the situation.

The projection function for catchError operator needs to return a new Observable. In this case we are returning a failed Observable created using throwError.

__6. Save changes.

__7. Open **book-list.component.ts**. Now the error handler receives a string message. In deleteBook() change the error handler like this.

```
err => {  
  alert(err)  
}
```

__8. Save changes.

__9. Start the Angular CLI dev server.

npm start

__10. Repeat the following error scenarios to verify that now you see the error message set by the service.

- Use an invalid URL in deleteBook. You should see the message "Sorry there was a problem at the server."
- Shutdown the development server. When you try to delete a book you should see "Oops! Please check your network connection and try again."

__11. Undo all the errors you have introduced to simulate problems.

Part 5 - Cache Book Data

In this part we will cache results from the getBook() method of DataService. It will have dual purpose:

1. It will show you how to cache HTTP calls that are expensive in nature.
2. We will use the cached data to recover from network connectivity problems. In case of a problem getBook() will return the previously cached data.

__1. Open **data.service.ts**

__2. Import the of() function which we will use to create a new Observable.

```
import { Observable, throwError, of } from 'rxjs';
```

__3. Import a few additional operators that we will need soon.

```
import {catchError, tap, retryWhen, delay, scan} from 'rxjs/operators';
```

__4. In the DataService class, add a member variable like this.

```
bookCache: {[isbn: string]: Book} = { }
```

(add it in the line above the 'constructor(...)')

bookCache is a dictionary like data structure where the key is always a string (the ISBN) and the value is a Book.

__5. Modify the getBook(isbn: string) method to add support for caching. The changes are highlighted in bold face.

```
getBook(isbn: string): Observable<Book> {  
  let cachedBook = this.bookCache[isbn]  
  
  if (cachedBook !== undefined) {  
    console.log("Got a cache hit")  
    return of(cachedBook)  
  }  
  
  return this.http.get<Book>(`/books/${isbn}`).pipe(  
    tap(book => this.bookCache[isbn] = book) //Populate cache  
  )  
}
```

__6. Save changes.

Part 6 - Test Caching

__1. Start the development server if it is not already running.

```
npm start
```

__2. Open a browser and enter the URL:

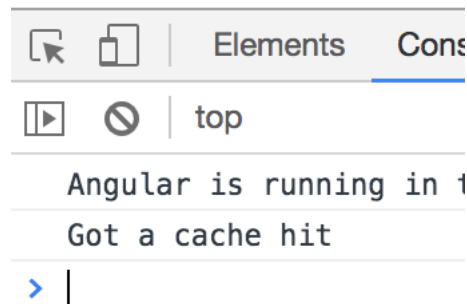
```
http://localhost:4200/
```

__3. Click the EDIT button for a book. This will open the EditBookComponent page. That component calls the getBook() method of DataService. That will cache the book.

__4. Click the back button of the browser.

__5. Click the EDIT button for the same book again.

__6. Press F12 and select the **Console** tab and verify that you see the "Got a cache hit" message.



This verifies that our caching is working.

__7. Go back to the main page.

`http://localhost:4200/`

Part 7 - Add Error Recovery

Now we will add recovery to the `getBook()` method. If the HTTP call fails the method will respond with a cached book if available.

__1. In the `getBook()` method of `data.service.ts`, after the `tap` operator add the `catchError` operator like shown in bold face below.

```
return this.http.get<Book>(`/books/${isbn}`) .pipe(  
  tap(book => this.bookCache[isbn] = book),  
  catchError(err => cachedBook ? of(cachedBook) : throwError(err))  
)
```

Basically, now the projection function of `catchError` returns a successful Observable that will emit the cached book if it is available. Otherwise we return a failed Observable with the original error.

__2. Save changes.

Part 8 - Test Recovery

__1. In the browser click the EDIT button for a book a few times and make sure that the data is cached. (Look for the "Got a cache hit" message in the console).

__2. Make sure you are in the main page.

`http://localhost:4200/`

__3. Shutdown the development server.

__4. Click the EDIT button again. The app will seamlessly show the edit form as if no error has occurred. This is possible because we are serving the data from a client side cache.

Part 9 - Not So Fast!

Caching is fun and greatly improves application performance. But as we all know it is a complicated business to keep the cache current and relevant. Right now we have a major problem. A cached book can become stale if the user deletes or updates the book. Let's fix that.

__1. In the `deleteBook()` method of `data.service.ts`, remove the book from the cache using the tap operator.

```
deleteBook(isbn: string): Observable<any> {  
  return this.http.delete(`/books/${isbn}`) .pipe(  
    tap(() => delete this.bookCache[isbn]),  
    catchError(...)  
  )  
}
```

__2. In the `saveBook()` method update the cache.

```
saveBook(book: Book): Observable<any> {  
  return this.http.put(`/books/${book.isbn}`, book) .pipe(  
    tap(() => this.bookCache[book.isbn] = book)  
  )  
}
```

__3. Save changes.

- ___ 4. Start the Angular CLI dev server.
- ___ 5. Make sure that there are no compilation errors.
- ___ 6. Make sure you are in the main page.

`http://localhost:4200/`

- ___ 7. Test these changes using these steps:
 - 1. Edit a book and make some changes.
 - 2. Shutdown the development server.
 - 3. Click EDIT for the book. It should show the modified data.

Part 10 - Extra Credit

- ___ 1. Cache the list of books from the `getBooks()` method and add error recovery.
- ___ 2. Make sure that the cache is kept up to date from `deleteBook()` and `updateBook()`.

Part 11 - Clean up

- ___ 1. In the REST Server Application command prompt press 'CTRL-C' to stop the server.
- ___ 2. Hit Control+C to close the Angular dev server.
- ___ 3. Close all open text editors and browser windows.

Part 12 - Review

In this lab we got to play with a few advanced topics of HTTP client development. We learned how to handle error and perform caching. We also learned how to recover from error by falling back to cached data. The good thing is that all of these techniques are isolated in the service. The components do not have to know if data is cached or a recovery has taken place.

Lab 15 - Using Angular Bootstrap

Bootstrap is an open source web GUI toolkit. It has two main aspects:

1. Provide a responsive CSS layout engine. This helps us create web sites that work well in mobile devices and in desktop. The CSS is available from the **bootstrap** NPM package.
2. Provide a set of widgets. This is by default implemented using jQuery. But the ng-bootstrap project has ported them to Angular. This is available from the **@ng-bootstrap/ng-bootstrap** package.

In this lab we will learn to do both. We will build on top of the book database application we have been building.

Part 1 - Install Bootstrap Packages

Normally, you can install all the required packages using the command **ng add @ng-bootstrap/ng-bootstrap**. But at the time of this wiring there's a bug in Angular CLI that makes this command fail. We will install all the parts separately.

__ 1. Open the **C:\LabWork\rest-client** folder in VS Code if not already open.

__ 2. From the VS Code terminal run:

```
npm install bootstrap @ng-bootstrap/ng-bootstrap
ng add @angular/localize
```

__ 3. Open **rest-client/src/styles.css**

__ 4. Add this line to include bootstrap CSS within styles.css.

```
@import '~bootstrap/dist/css/bootstrap.min.css';
```

__ 5. Save changes.

__ 6. Commit all changes to Git.

Part 2 - Layout the Pages

__ 1. Open **app.component.html**

__2. Wrap the `<router-outlet>` tag in HTML shown in boldface below. This will layout the page in a grid. All content will go in the center with 8 column width.

```
<div class="container">  
  <div class="row">  
    <div class="col-md-2"></div>  
    <div class="col-md-8">  
      <router-outlet></router-outlet>  
    </div>  
    <div class="col-md-2"></div>  
  </div>  
</div>
```

__3. Save changes.

Part 3 - Use Bootstrap Buttons

__1. Open **book-list/book-list.component.html**

__2. Add the **btn-primary** class to the Add a Book button.

```
<button class="btn-primary" [routerLink]="['/add-book']">Add a  
Book</button>
```

__3. Add the **btn-danger** class to the DELETE button.

```
<button class="btn-danger" (click)="deleteBook(book)">DELETE</button>
```

__4. Save.

Part 4 - Test

__1. Start the backend web service. Open a command prompt and from **C:\LabFiles\book-server**, start the server:

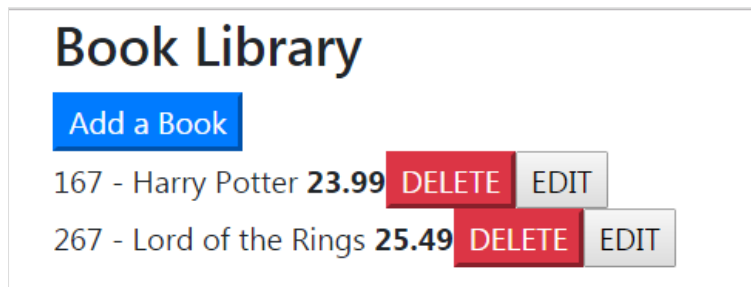
```
npm start
```

__2. Run the dev server from the VS Code terminal:

```
npm start
```

__3. Open **http://localhost:4200/** in the browser.

__4. Make sure that the content is laid out in the center of the page and buttons have colors.



Part 5 - Style the Forms

__1. Open **add-book.component.html**

__2. Apply the form-control class to every `<input>` element like this.

```
<input class="form-control" ... />
```

__3. Apply the btn-primary class to the button.

```
<button class="btn-primary" ...>
```

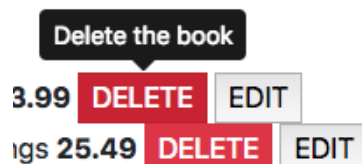
__4. Save changes.

__5. Make the same changes in **edit-book.component.html**

__6. Test the forms out.

Part 6 - Use Bootstrap Widget

We will now add tooltip widget to the button. It will look like this.



Bootstrap widgets are available from an Angular module called NgbModule. This is available from the NPM package called `@ng-bootstrap/ng-bootstrap` which we need to install first.

__1. Stop the Angular dev server by hitting Control+C in VS Code terminal.

__2. Open a new command prompt windows and from the **C:\LabWork\rest-client** run:

```
npm install @ng-bootstrap/ng-bootstrap@6.1.0
```

You can ignore warnings about peer dependencies for jQuery and popper. You can also ignore warnings about optional dependencies.

__3. Open **app.module.ts**

__4. Import the module name.

```
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';
```

__5. Then add the module to the imports list like this.

```
imports: [  
    NgbModule,  
    ...  
]
```

__6. Save changes.

__7. Open **book-list.component.html**

__8. Show tooltip for the DELETE and EDIT buttons like this.

```
<button ... placement="top" ngbTooltip="Delete the  
book">DELETE</button>  
<button ... placement="top" ngbTooltip="Edit the book">EDIT</button>
```

__9. Save changes.

__10. Start the Angular dev server from VS Code terminal.

__11. Verify that the tool tips show up when you hover your mouse over the buttons.

Part 7 - Add Pagination Support

We will now use the pagination widget to show only 4 books at a time.

__1. Open **book-list.component.ts**

__2. Add these member variables to the class.

```
page = 1; //Current page. Starts with 1
pageSize = 4;
```

These variables can be added above the 'constructor(...)' line.

__3. Add this method that returns the books for the current page.

```
getDisplayList() : Book[] {
    return this.books.slice(
        (this.page - 1) * this.pageSize, this.page * this.pageSize)
}
```

This method can be added just before the 'deleteBook' method.

__4. Save changes.

__5. Open **book-list.component.html**

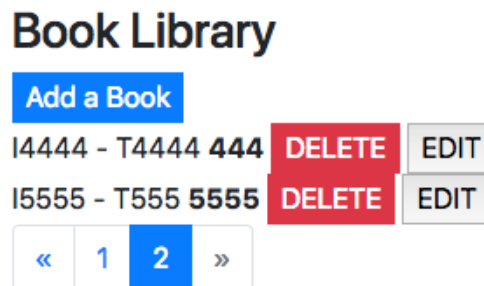
__6. Make changes shown in bold face below.

```
<div *ngFor="let book of getDisplayList()">
    ...
</div>

<ngb-pagination [collectionSize]="books.length"
    [pageSize]="pageSize" [(page)]="page"></ngb-pagination>
</div>
```

__7. Save changes.

__8. Test the changes. Add a whole bunch of new books. Make sure pagination is working.



Part 8 - Clean up

- ___1. In the REST Server Application command prompt press 'CTRL-C' to stop the server.
- ___2. Hit Control+C to close the Angular dev server.
- ___3. Close all open text editors and browser windows.

Part 9 - Review

In this lab you learned how to use a third-party Angular library. You also learned how to apply layout to an Angular application using Bootstrap.

Lab 16 - Lazy Module Loading

In this lab you will learn how to lazily load a module. The main app module will have these components:

- HomeComponent - The main home page. Path: /
- CatalogComponent - Product catalog. Path: /catalog

We will develop a feature module called AccountModule. It will have these components:

- AccountHomeComponent - The main account management page. Path: /account
- OrderHistoryComponent - Shows order history. Path: /account/orders
- AddressbookComponent - User's shipping addresses. Path: /account/address

We will load the AccountModule only when user navigates to one of its components.

Part 1 - Get Started

We will now create a new Angular application.

- ___ 1. Close all VS Code windows.
- ___ 2. Open a command prompt window.
- ___ 3. Go to the **C:\LabWork** folder.
- ___ 4. Run this command to create a new project called **large-app**.

```
ng new large-app --routing --defaults
```

Part 2 - Create the Main Module Components

- ___ 1. Open the **C:\LabWork\large-app** folder in VS Code.
- ___ 2. In VS Code open a new terminal.
- ___ 3. Run these commands in the terminal.

```
ng g component home
```

```
ng g component catalog
```

Part 3 - Create the Feature Module

__1. In VS Code terminal run this command to create the AccountModule feature module. We enable routing for it.

```
ng g module account --routing
```

Part 4 - Create the Components in AccountModule

__1. Run these commands to create a few components in the account module.

```
ng g component account/account-home  
ng g component account/order-history  
ng g component account/addressbook
```

Note: By adding "account/" before the component name we are creating them in the AccountModule.

__2. Open **src/app/account/account.module.ts** and verify that the components were added to AccountModule.

```
declarations: [AccountHomeComponent, OrderHistoryComponent,  
AddressbookComponent]
```

__3. Close the file.

Part 5 - Setup Routing for AccountModule

We will now setup routing for the components in AccountModule.

__1. Open **src/app/account/account-routing.module.ts**

__2. Import the component class names.

```
import { AccountHomeComponent } from './account-home/account-  
home.component';  
import { OrderHistoryComponent } from './order-history/order-  
history.component';  
import { AddressbookComponent } from  
'./addressbook/addressbook.component';
```

___3. Add the components to the routes table.

```
const routes: Routes = [  
  {path: "", component: AccountHomeComponent},  
  {path: "orders", component: OrderHistoryComponent},  
  {path: "address", component: AddressbookComponent}  
];
```

Note, the paths are relative to the feature module. We will later map AccountModule to the path "/account".

Observe how a feature module uses RouterModule.forChild to install the route table.

```
@NgModule({  
  imports: [RouterModule.forChild(routes)],  
  exports: [RouterModule]  
})
```

This is different from RouterModule.forRoot() that the main application module needs to use.

___4. Save changes and close the file.

Part 6 - Setup Routing for the Main Module

___1. Open **src/app/app-routing.module.ts**

___2. Import the class names for the components that belong directly to the main module.

```
import { HomeComponent } from '../home/home.component';  
import { CatalogComponent } from '../catalog/catalog.component';
```

___3. In the routes table add the two components like this.

```
const routes: Routes = [  
  {path: "", component: HomeComponent},  
  {path: "catalog", component: CatalogComponent},  
];
```


___4. Now add a route for the AccountModule feature module.

```
const routes: Routes = [  
  {path: "", component: HomeComponent},  
  {path: "catalog", component: CatalogComponent},  
  {  
    path: "account",  
    loadChildren: () => import('./account/account.module').then(  
      m => m.AccountModule)  
  }  
];
```

Note:

1. We do not import the name AccountModule from here. Nor do we import AccountModule from the main application module AppModule. This is necessary for lazy loading to work. This is different from the way say FormsModule is imported into AppModule.
2. The value of loadChildren is a lambda that dynamically loads AccountModule.

___5. Save changes and close the file.

Part 7 - Add Links

Now we will add links for all our components.

___1. Open **src/app/app.component.html**

___2. Delete all contents.

___3. Set the template to be like this.

```
<a href [routerLink]="['/']">Home</a>&nbsp;    
<a href [routerLink]="['/catalog']">Shop</a>&nbsp;    
<a href [routerLink]="['/account']">My Account</a>&nbsp;    
<a href [routerLink]="['/account/orders']">My Orders</a>&nbsp;    
<a href [routerLink]="['/account/address']">Address Book</a>  
  
<router-outlet></router-outlet>
```

___4. Save changes and close the file.

Part 8 - Test

We will now use the application and verify that AccountModule is getting loaded lazily when user navigates to one of its components.

__ 1. From the VS Code terminal.

```
npm start
```

__ 2. Open a browser and go to **http://localhost:4200/**

__ 3. You should see this.

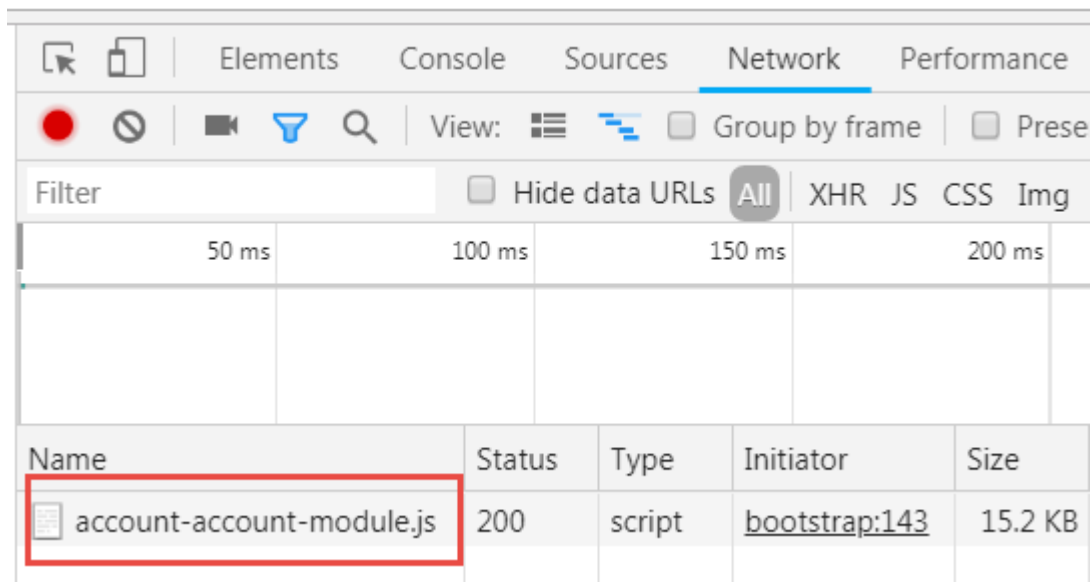
[Home](#) [Shop](#) [My Account](#) [My Orders](#) [Address Book](#)

home works!

__ 4. Open developer tool (F12) of the browser and view the **Network** tab.

__ 5. Click the **My Account** link.

__ 6. Verify that the JS code for the AccountModule just got downloaded.



This proves that the account module code was lazily loaded.

__ 7. Test the other links.

__ 8. Hit Control+C to end the Angular dev server.

Part 9 - Run a Build

__1. From the Terminal line run:

```
ng build
```

__2. Look at the **dist\large-app** folder.

```
dir dist\large-app
```

__3. You should see the file **account-account-module-*.js** in the folder large-app. Lazy loaded feature modules are built into a separate file. This is necessary for it to be loaded separately from the rest of the application.

Note, in production build the name **account-account-module.js** will be lost. Instead the file will have a name something like 4.d95fa8de5207b6b2f2c2.js. It will still be lazy loaded.

__4. Close VS Studio window.

Part 10 - Review

In this lab we learned how to lazily load modules. As you can expect this can speed up the initial load of the application.

Lab 17 - Advanced Routing

In this lab you will get hands-on experience with the following Angular Routing features:

- Defining Default Routes
- Adding Error Pages
- Using the "routerLinkActive" directive
- Adding Router Guards
- Adding Child Routes

Part 1 - Get Started

We will now create a new Angular application.

- ___ 1. Close all VS Studio window if open.
- ___ 2. Open a command prompt window.
- ___ 3. Go to the **C:\LabWork** folder.
- ___ 4. Run this command to create a new project called **advanced-routing**.

```
ng new advanced-routing --routing --defaults
```

Part 2 - Setup Starter Files

To save time basic parts of the application have already been implemented.

- ___ 1. Copy files as directed below:

```
copy contents of: C:\LabFiles\advanced-routing\app  
into this dir: C:\LabWork\advanced-routing\src\app
```

Choose **Yes to overwrite** files in the destination directory.

- ___ 2. Copy the following file as directed below:

```
copy this file: C:\LabFiles\advanced-routing\styles.css  
into this dir: C:\LabWork\advanced-routing\src\
```

Choose **Yes to overwrite** the file in the destination directory.

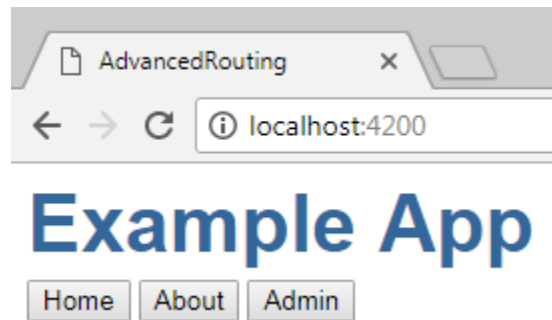
- ___3. From VS Code open the **C:\LabWork\advanced-routing** folder.
- ___4. In VS Code open a terminal.
- ___5. Run the following command in the terminal to start the dev server:

```
npm run start
```

- ___6. Open a browser window to the following address:

```
http://localhost:4200/
```

- ___7. The browser window should appear like this:



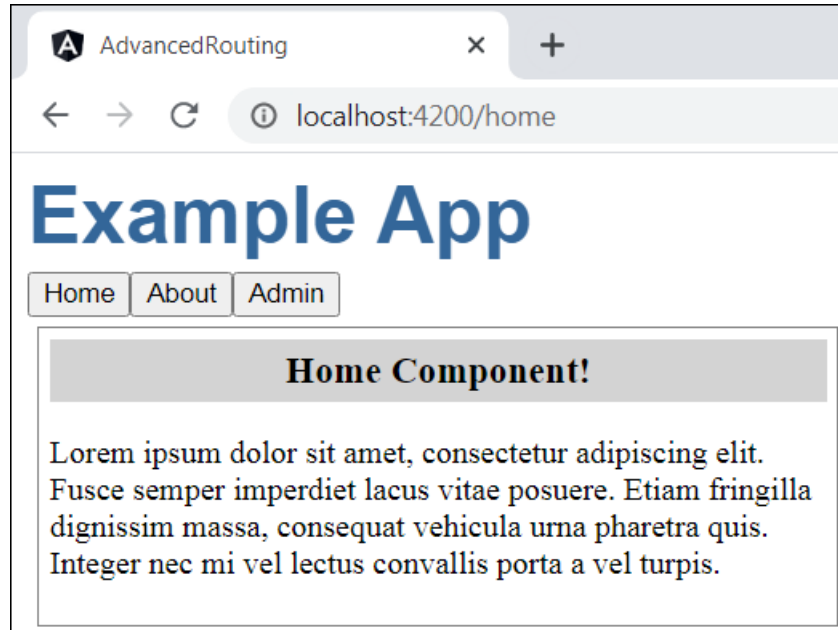
The lab setup is complete.

Part 3 - Defining a Default Route

Description.

- ___1. Take a look at the browser window and take note of the following:
- The address bar shows: localhost:4200
 - The title **Example App** in large blue letters
 - Three buttons labeled: "Home", "About" and "Admin"
- ___2. Now click on the "Home" and notice how:
- The corresponding path is added in the address bar: **localhost:4200/home**

- The "Home" component now shows below the buttons.



From this we can see that the basic routing is working - the button takes us to the Home component and "/home" is appended to the address in the address bar.

But the application does not know which component to show on startup when there is no path at the end of the address. In this part of the lab we will modify the app so that it does show a specific component by default.

___3. Open **src\app\app-routing.module.ts**.

___4. Add the highlighted text to the start of the `appRoutes` array. Don't forget the comma - "," between the new element and the ones that follow!

```
const appRoutes: Routes = [
  {path: '', component: HomeComponent}
  ,{path: 'home', component: HomeComponent}
  ,{path: 'about', component: AboutComponent }
  ,{path: 'admin', component: AdminComponent }
];
```

___5. Save the file.

___6. Make sure the browser refreshes the following address:

http://localhost:4200/

___7. Notice that:

- The address in the address bar is "localhost:4200" with no path at the end.

- The Home component is shown by default.
- The buttons still work the same as before - each navigating to a specific component.

Part 4 - Adding a Route Error Page

In this part of the lab we will add an error page and add it to our routing setup so that it is displayed whenever a non-existent path (non-valid route) is added in the address bar.

The app is currently setup to handle four different routes:

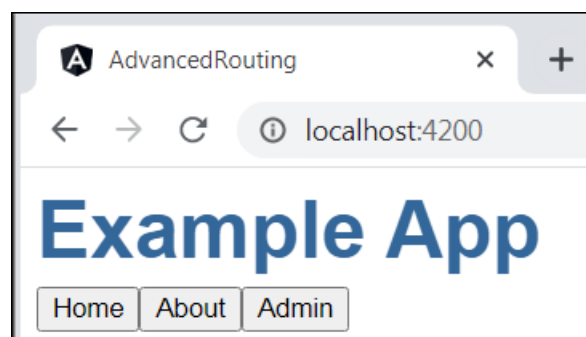
- "" (no path)
- "/home"
- "/about"
- "/admin"

__1. Try entering the following path in the browser's address bar. Notice that it still points to the current app ("localhost:4200") but includes an invalid path ("/pathx"):

`http://localhost:4200/pathx`

The app comes up but:

- The path "/pathx" has been stripped out of the address bar.
- No component is shown



__2. Open up the browser's developer tools (F12 on Chrome) and check the JavaScript console for exceptions. You should find one with the following text:

Error: Cannot match any routes. URL Segment: 'pathx'

What we would like to see here is a component indicating the error condition.

__3. Take a look at the following file in your text editor:

src\app\error\route.error.component.ts

Notice the following text in the component's template:

```
<h3>Route Error!</h3>
<p>The Current route is invalid!</p>
```

We will display this component whenever there is an incorrect route.

__4. Open the following file in VS Code:

\src\app\app-routing.module.ts

__5. Add the following import after the other imports at the top of the file:

```
import { RouteErrorComponent } from './error/route.error.component';
```

__6. Add the lines in bold to the end of the appRoutes array:

```
const appRoutes: Routes = [
  {path: '', component: HomeComponent}
  ,{path: 'home', component: HomeComponent}
  ,{path: 'about', component: AboutComponent }
  ,{path: 'admin', component: AdminComponent }
  ,{path: 'error', component: RouteErrorComponent }
  ,{path: '**', redirectTo: '/error' }
];
```

__7. Save and close **app-routing.module.ts**

__8. Close the current browser window.

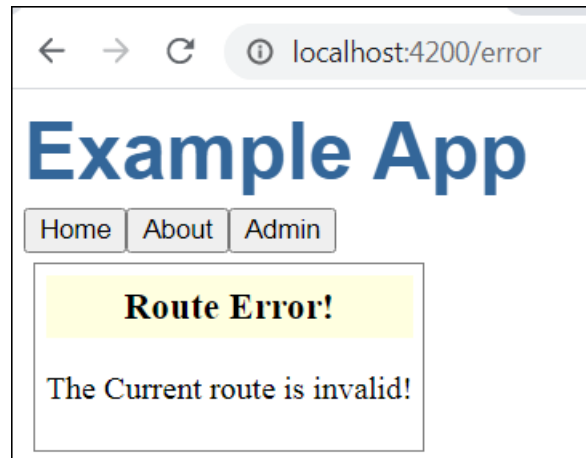
__9. Open a new browser window with the following url which contains an invalid path:

http://localhost:4200/pathx

__10. This time you will see the following:

The invalid path "/pathx" is replaced in the address bar with "/error" indicating the route to the new component.

The `RouteErrorComponent` is shown, indicating the error condition:



__ 11. Click on any of the buttons to again display any of the valid routes.

Part 5 - Highlighting the Active Router Link

Clicking on the buttons in the app changes the URL in the address bar to indicate which route is currently active. The Angular component router then changes the visible component to match. To make the change more apparent in this part of the lab we will highlight the button that caused the most recent navigation using the *routerLinkActive* directive.

__ 1. Open up the following file:

`\src\app\app.component.html`

__ 2. Take a look at the code used to create the buttons:

```
<input type=button [routerLink]="['/home']" value="Home" >  
<input type=button [routerLink]="['/about']" value="About" >  
<input type=button [routerLink]="['/admin']" value="Admin" >
```

__ 3. Add the following attribute to each of the button input elements:

`routerLinkActive="active"`

___4. The resulting code should look like this:

```
<input type=button [routerLink]="['/home']" routerLinkActive="active"
value="Home" >
<input type=button [routerLink]="['/about']" routerLinkActive="active"
value="About" >
<input type=button [routerLink]="['/admin']" routerLinkActive="active"
value="Admin" >
```

This will apply the CSS class called "active" to the currently activate button.

___5. Save and close app.component.html.

The routerLinkActive directive used above tells Angular to apply the "active" class to a button when its routerLink represents the current route. This means that after clicking on a given button Angular adds the "active" class to its list of classes and at the same time removes the "active" class from all the other input elements.

___6. Open the following file from the project root directory:

\src\styles.css

___7. Scroll down until you see the "active" class style section and notice how it changes the font color and weight. These are the changes you should when the buttons are pressed.

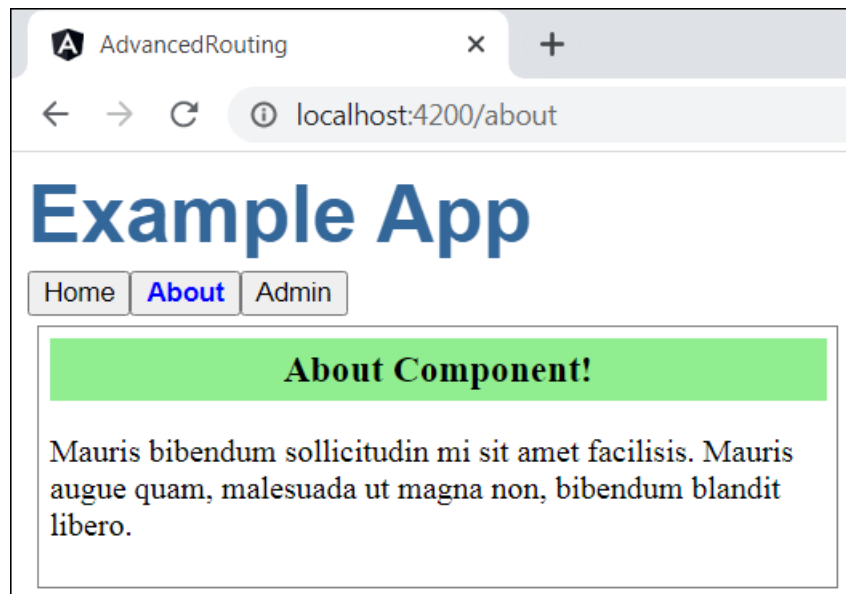
```
.active{
  color: blue;
  font-weight:bold;
}
```

___8. Close styles.css.

___9. Take a look at the browser window. It should have updated automatically after you saved your changes. If not then close it and open a new window with the app url:

http://localhost:4200

__10. Click on the "About" button. This will navigate to the About component and now at the same time it will set the About button text to the "active" style - blue & bold.



__11. Click on the other buttons and notice how the "active" style gets applied. As you can see the current route is clearer to the user when using the routerLinkActive directive.

Part 6 - Add a Guarded Route

Right now anyone using the site can access the Admin page. In this part of the lab we will add a guard to the Admin route that requires users to log in before they can access it. This will involve the following tasks:

1. Add a Login button and an indicator for logged in status.
2. Add a guard service that implements "CanActivate".
3. Add a login() function to the component.
4. Add the guard to the route in the route table.
5. Test.

Task 1 - Add a login button and indicator

- ___ 1. Open `src/app/app.component.html` in your text editor.
- ___ 2. Insert the following after the end tag of the button div and before the `<main>` tag.

```
</div>
<div class=login >
<input class=user type=button (click)="login()" value="{{loginLabel}}"
>
<span class=user [hidden]="user==='na'">User: {{user}}</span>
</div>
<main>
```

- ___ 3. Save and close the file.

The above code adds a button on screen that shows in a box below the navigation buttons. If you have the JavaScript console open you will notice some errors have appeared. They will go away after we complete a few more steps.

Task 2 - Add a guard service

- ___ 4. Open `auth.service.ts`
- ___ 5. The `auth.service.ts` file contains a service that implements the *canActivate* guard method as well as managing the *isLoggedIn* state variable:

```
export class AuthService implements CanActivate {
  isLoggedIn: boolean = false;

  login() {
    this.isLoggedIn = true;
    console.log('AuthService: logging in');
  }
  logout() {
    this.isLoggedIn = false;
    console.log('AuthService: logging out');
  }
  constructor() {}
  canActivate() {
    return this.isLoggedIn;
  }
}
```

We will use this service to both guard the Admin route and perform login/logout. In real life this can be done in separate services.

- ___ 6. Close the file.

Task 3 - Add a login() function to the component

__7. Open \src\app\app.component.ts

__8. Add the following line after the other imports at the top of the file to import the AuthService:

```
import { AuthService } from '../auth.service';
```

__9. Modify the constructor to inject the AuthService:

```
constructor( private auth: AuthService, private router: Router ) {}
```

__10. Insert the following new method into the AppComponent class:

```
login() {  
  if(this.loginLabel === 'Log in'){  
    this.auth.login();  
    this.user = 'Admin';  
    this.loginLabel = 'Log out';  
  }else{  
    this.auth.logout();  
    this.user = 'na';  
    this.loginLabel = 'Log in';  
    let link = ['/home'];  
    this.router.navigate(link);  
  }  
}
```

__11. Save and close the file.

Task 4 - Add the guard to the route table

__12. Open \src\app\app-routing.module.ts

__13. Add the following line after the other imports at the top of the file to import the AuthService:

```
import { AuthService } from '../auth.service';
```

__14. Add the canActivate guard to the admin route in the appRoutes array. The updated route should match the following:

```
, {path: 'admin', component: AdminComponent, canActivate:  
  [AuthService] }
```

__15. Save and close the file.

Task 5 - Test

__16. Open the app in a new browser window with the following URL:

`http://localhost:4200/`

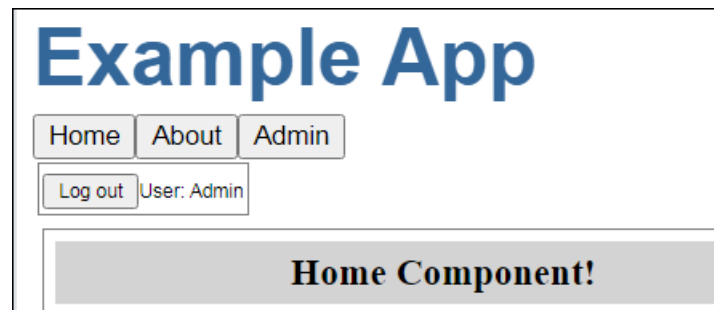
Errors that were previously visible in the JavaScript console should have gone away at this point. If you do see any errors make sure to fix them before you move on.

You should notice a "log in" button below the navigation buttons. Users need to click this button to log in as an administrator. Don't log in just yet. First we want to test how the app works when the user is Not logged in.

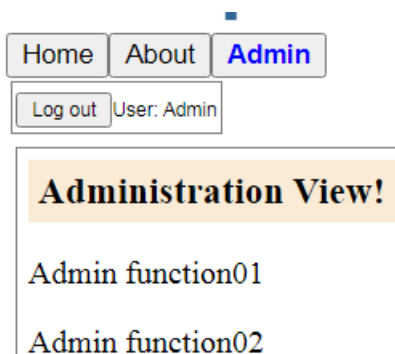


__17. Try clicking on the **Admin** button. Notice how the button is no longer active and no longer takes you to the admin screen.

__18. Click on the "**Log in**" Button. The screen should change to indicate that you have logged in.



__19. Now try again to click on the **Admin** button. You should notice that the app now navigates as requested to show the Admin view.



__20. Click on the "**Log out**" Button. The application logs out the user and navigates away from the restricted page.



Part 7 - Adding Child Routes - Extra Credit (Optional)

In this part of the lab we will add a Products component view. Products include two child components: *list* and *detail*, accessible through child routes. In summary:

- `/products` will map to the `ProductsComponent`.
- `/products/list` child route will map to `ProductsListComponent`

- `/products/detail/:id` child route will map to `ProductsDetailComponent`

The `Products` component and its child components already exist in the `\src\app\products` directory. We will add the `Product` component to the app and make any adjustments needed to access its children.

Adding child routes will involve the following tasks:

- Add the `ProductsModule`
- Add a route for the `Products` component
- Add a button to allow navigation to the `Products` component.
- Review Routing Code

Task 1 - Add the `ProductsModule`

__ 1. Open `\src\app\app.module.ts`

__ 2. Add the following line to import the `ProductsModule`. Place it after all the other imports at the top of the file:

```
import { ProductsModule } from '../products/products.module';
```

__ 3. Add `ProductsModule` to the *imports* array as shown below. Don't forget to add a comma `,` between `ProductsModule` and `AppRoutingModule`.

```
imports: [  
  BrowserModule,  
  FormsModule,  
  HttpClientModule,  
  ProductsModule,  
  AppRoutingModule  
],
```

Warning: The app will not work properly unless the `ProductsModule` appears BEFORE the `AppRoutingModule` in the imports array!

__ 4. Save and close the file.

Task 2 - Add a route for the Products component

__5. Open `\src\app\app-routing.module.ts`

__6. Add the following line to import the products component. Place it after all the other imports at the top of the file:

```
import { ProductsComponent } from '../products/products.component';
```

__7. Add the following route to the `appRoutes` array. Place it after the `about` route and before the `admin` route:

```
, {path: 'about', component: AboutComponent }  
, {path: 'products', component: ProductsComponent }  
, {path: 'admin', component: AdminComponent, canActivate:  
  [AuthService] }
```

__8. Save and close the file.

__9. Open `\src\app\products\products-routing.module.ts`.

__10. Notice how the child route table is setup.

```
const productsRoutes: Routes = [  
  {  
    path: 'products',  
    component: ProductsComponent,  
    children: [  
      { path: '', redirectTo: 'list', pathMatch: 'full' },  
      { path: 'list', component: ProductsListComponent },  
      { path: 'detail/:id', component: ProductsDetailComponent }  
    ]  
  }  
];
```

__11. Open `\src\app\products\products.component.html`.

__12. Notice that it has `<router-outlet></router-outlet>` where the components matching the child routes will be inserted.

__13. Close the files.

Task 3 - Add a button to allow navigation to the Products component

__14. Open the \src\app\app.component.html file in your text editor.

__15. Add the following products button element right after the admin button element:

```
<input type=button [routerLink]="['/admin']"
      routerLinkActive="active" value="Admin" >
<input type=button [routerLink]="['/products']"
      routerLinkActive="active" value="Products" >
</div>
```

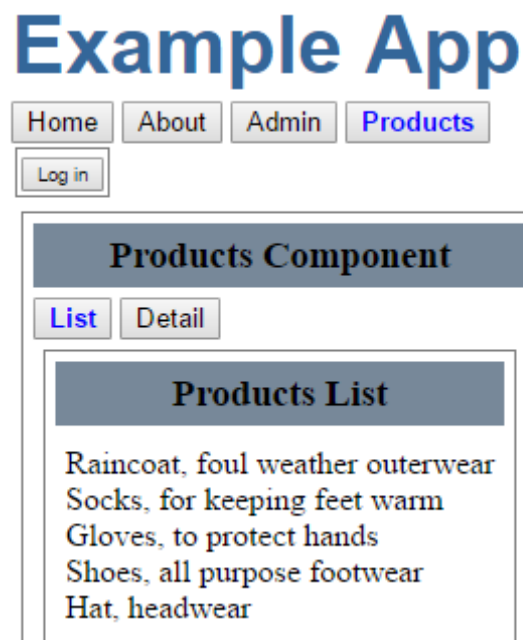
__16. Save and close the file.

Part 8 - Test

__1. Run the app in the browser and you will see a Products button in addition to the other navigation buttons.

__2. Click on the Products button, the ProductsComponent will appear.

__3. The default child component will be the product list. Clicking on the List or Detail buttons will navigate between the two child components. Clicking on an item in the list will also navigate to the detail screen.



Part 9 - Review Routing Code

Now that we have a working app with child routes let's take a look at the code and see how it works.

__1. Routing to the main ProductsComponent is set up as part of the appRoutes array in the \src\app\app-routing.modules.ts file:

```
const appRoutes: Routes = [
  {path: '', component: HomeComponent}
  ,{path: 'home', component: HomeComponent}
  ,{path: 'about', component: AboutComponent }
  ,{path: 'products', component: ProductsComponent }
  ,{path: 'admin', component: AdminComponent, canActivate:
[AuthService] }
  ,{path: 'error', component: RouteErrorComponent }
  ,{path: '**', redirectTo: '/error' }
];
```

Once the ProductsComponent route is in place we navigate to it using the button we added to the \src\app\app.component.html file:

```
<input type=button [routerLink]="['/products']"
routerLinkActive="active" value="Products" >
```

The routerLink here simply points to the root product route. The root product route tells angular to load the ProductComponent into the <router-outlet> in the \src\app\app.component.html file.

__2. If we take a look at the \src\app\products\products.component.html file though we will see another <router-outlet> that needs to be filled:

```
<div>
<h3>Products Component</h3>
<input type=button [routerLink]="['list']" routerLinkActive="active"
value="List" >
<input type=button [routerLink]="['detail', 0]"
routerLinkActive="active" value="Detail" >
<router-outlet></router-outlet>
</div>
```

__3. To provide a component for the <router-outlet> in the Product component we create the \src\app\products\products-routing.module.ts file:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductsComponent } from '../products.component';
import { ProductsListComponent } from '../list/products-list.component';
import { ProductsDetailComponent } from '../details/products-
detail.component';
```

```

const productsRoutes: Routes = [
  {
    path: 'products',
    component: ProductsComponent,
    children: [
      { path: '', redirectTo: 'list', pathMatch: 'full' },
      { path: 'list', component: ProductsListComponent },
      { path: 'detail/:id', component: ProductsDetailComponent }
    ]
  }
];

@NgModule({
  imports: [ RouterModule.forChild(productsRoutes) ],
  exports: [ RouterModule ]
})

export class ProductsRoutingModule {}

```

This file defines two child routes and redirects the default route to the list component. This file is then imported and added to the imports array in the `\src\app\products\products.module.ts` file.

___ 4. Buttons on the `products.component.html` allow us to navigate between the child routes:

```

<input type=button [routerLink]="['list']" routerLinkActive="active"
value="List" >
<input type=button [routerLink]="['detail', 0]"
routerLinkActive="active" value="Detail" >

```

Notice there is no backslash defined on the 'list' and 'detail' routerLink entries. This indicates navigation to child rather than root routes. The equivalent full root specified entries would be `\products\list` and `\products\details`.

Part 10 - Clean Up

- ___ 1. Close the browser.
- ___ 2. In VS Code terminal, hit '**<CTRL>-C**' to terminate the server.
- ___ 3. Close all open files.

Part 11 - Review

In this lab we explored various features of Angular Routing including:

- Defining Default Routes
- Adding Error Pages
- Using the "routerLinkActive" directive
- Adding Router Guards
- Adding Child Routes

Lab 18 - Unit Testing

Angular relies on these technologies for testing:

- Jasmine – Provides the API to write test cases
- Karma – Provides the runtime to execute the test cases. By default it will use Chrome browser's JavaScript VM to run tests.

Writing test cases in Jasmine is pretty easy. A single fact makes things complicated though – asynchronous programming. Most real life applications will make back end service calls and will be inherently asynchronous. In this lab we will learn how to test such applications. We will go back to our Dealer Inventory application and write test cases for it.

Part 1 - Get Started

- ___ 1. Close all VS Code windows.
- ___ 2. Shutdown any backend web service servers you may be running.
- ___ 3. Open **C:\LabWorks\my-test-app** in VS Code.
- ___ 4. Open a terminal in VS Code.

We will first manually make sure the app is working properly.

- ___ 5. Run the Angular dev server from the terminal.

```
npm start
```

- ___ 6. Start the backend inventory web service server. To do that open a command prompt window. Then:

```
cd C:\LabFiles\inventory-server
```

```
npm start
```

- ___ 7. Open a browser and go to **http://localhost:4200/**.
- ___ 8. Make sure you can add, edit and delete a vehicle.
- ___ 9. In VS Code terminal shutdown the dev server. We don't need it to run our unit tests. But we still need the backend web service to be running.

Part 2 - Run Unit Tests

When you generate a service or component using the CLI, accompanying test case files are also created. We have made a lot of changes to the code but never paid any attention to these test cases. Let's run unit test and see what happens.

___ 1. In VS Code terminal enter this command to run unit test.

```
npm test
```

___ 2. You will see this compilation error on terminal.

```
ERROR in src/app/app.component.spec.ts:22:16 - error TS2339: Property 'title'
component'.
```

```
22     expect(app.title).toEqual('my-test-app');
      ~~~~~
```

___ 3. Karma launches a separate Chrome window. It will show.

Karma v5.0.9 - connected

Chrome 83.0.4103.116 (Mac OS 10.15.5) is idle

 Jasmine 3.5.0

Incomplete: No specs found, , randomized with seed 10833

The compilation error is preventing any test cases from running.

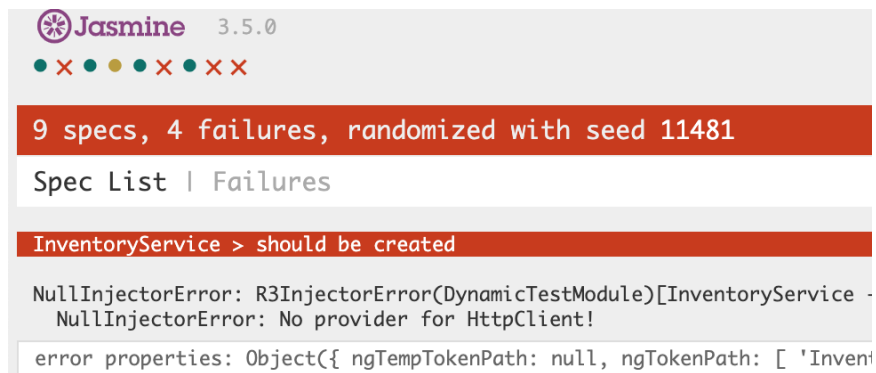
___ 4. Open `src/app/app.component.spec.ts`.

___ 5. Comment out the offending line.

```
it(`should have as title 'my-test-app'`, () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  //expect(app.title).toEqual('my-test-app');
});
```

___ 6. Save changes.

___ 7. ng test will pick up the change, recompile and run test cases again. Now test cases will run but many will fail.



No worries there. As we develop test cases these problems will be sorted out.

Part 3 - Unit Test the Service

Services are easier to test than components. At minimum you should write test cases for services. We will now test `InventoryService`.

__1. Open the test script file for the service **src/app/inventory.service.spec.ts**.

A test script runs in it's own Angular module. It does not use the `AppModule`. This has many advantages. For example, we can supply an alternate provider for a service in the test module.

The following line sets up the test module.

```
TestBed.configureTestingModule({});
```

We need to import `HttpClientModule` from this test module. Otherwise the service won't be able to inject `HttpClient`. In fact that is one of the problems reported by Karma in the test browser window.

__2. Add these import statements at the top.

```
import { HttpClientModule } from '@angular/common/http';
import { Vehicle } from './vehicle';
```

__3. Set up the test module as shown in bold face below.

```
TestBed.configureTestingModule({
  imports: [
    HttpClientModule
  ]
});
```

__4. Save.

Now we are ready to write test cases. First let's add a sample Vehicle object.

__5. Below the line:

```
let service: InventoryService;
```

Add:

```
let sampleVehicle = new Vehicle(  
    "V1000", 2019, "BMW", "330i", 25000, 34000, false, [])
```

We will add this vehicle before each test case is run. We will delete the vehicle at the end of each test case. This will clean up the slate.

__6. Modify the beforeEach() function as shown in bold face. Note we are now taking the done function as a parameter in the lambda.

```
beforeEach((done) => {  
    TestBed.configureTestingModule({  
        imports: [  
            HttpClientModule  
        ]  
    });  
    service = TestBed.inject(InventoryService);  
  
    service.addVehicle(sampleVehicle).subscribe(() => {  
        done()  
    })
```

Note, use of the done() function is essential when making asynchronous calls. Otherwise Karma wont wait for the addVehicle() method to complete.

__7. Add an afterEach() function like this. It deletes the sample vehicle.

```
afterEach((done) => {  
    service.deleteVehicle(sampleVehicle).subscribe(() => {  
        done()  
    })  
})
```

__8. Below the existing test case:

```
it('should be created', () => {  
  expect(service).toBeTruthy();  
});
```

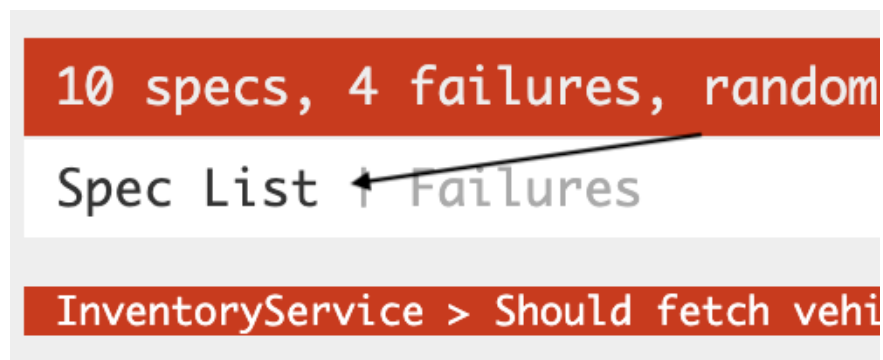
Add a new test case that will test the `getInventory()` method:

```
it('Should fetch vehicles', (done) => {  
  service.getInventory().subscribe(list => {  
  
    let v = list.find(item => sampleVehicle.VIN === item.VIN)  
    expect(v).toBeDefined();  
  
    done()  
  })  
});
```

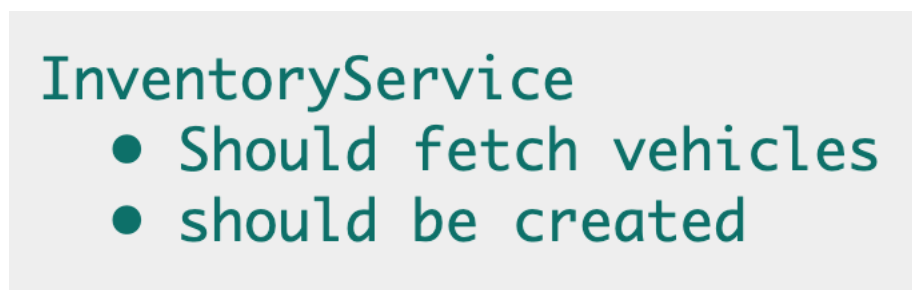
__9. Save changes.

__10. ng test will recompile and re-run the tests.

__11. In the test browser click **Spec List**.



__12. Verify that both InventoryService test cases have passed.



Let's write one more test case. We will test the `updateVehicle()` method.

__13. Add this test case.

```
it("Should update vehicle", (done) => {
  let updated = new Vehicle("V1000", 2018, "BMW", "750i", 25000, 54000,
false, [])

  service.updateVehicle(sampleVehicle.VIN, updated).subscribe(() => {
    service.getInventory().subscribe(list => {
      let v = list.find(item => sampleVehicle.VIN === item.VIN)!

      expect(v).toBeDefined();
      expect(v.model).toBe("750i")
      expect(v.price).toBe(54000)

      done()
    })
  })
})
```

__14. Save.

__15. Wait for Karma to recompile and run the test cases. In the browser click **Spec List** again to make sure all test cases for InventoryService are passing.

Part 4 - Test the Photo Gallery Component

This component will be easy to test since it doesn't have any asynchronous operations.

__1. Open **src/app/photo-gallery/photo-gallery.component.spec.ts**.

__2. Notice how we lookup the component instance before every test case.

```
beforeEach(() => {
  fixture = TestBed.createComponent(PhotoGalleryComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
```

Once we have the component instance testing it is quite easy.

__3. Add a few test cases like this.

```
it("Should handle empty list", () => {
  component.moveToNext()
  expect(component.currentIndex).toBe(0)
})

it("Should navigate correctly", () => {
  component.imageList = ["1.png", "2.png"]
  component.moveToNext()
  expect(component.currentIndex).toBe(1)

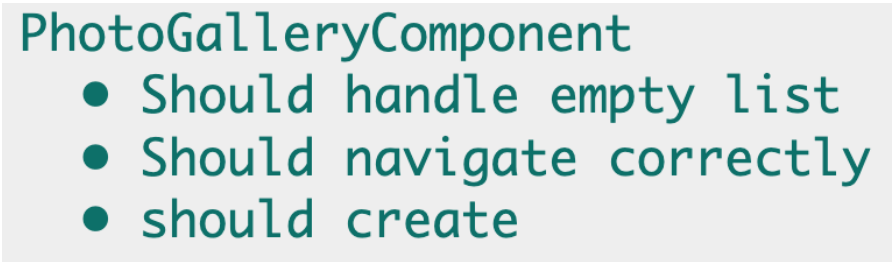
  //Out of bounds test
  component.moveToNext()
  component.moveToNext()
  component.movePrevious()

  expect(component.currentIndex).toBe(0)
})
```

Generally speaking you should have only one test condition (expect() call) in one test case. This makes it easier to find any problems when a test case fails. Otherwise you have to browse the failure messages to find out which condition had failed. Above we combined two test cases into one to save typing.

__4. Save.

__5. In the test browser click **Spec List** and make sure all tests are passing.



PhotoGalleryComponent

- Should handle empty list
- Should navigate correctly
- should create

Part 5 - Test the Reactive Vehicle Form Component

Reactive forms are easy to test.

__1. Open `src/app/vehicle-form-reactive/vehicle-form-reactive.component.spec.ts`.

___2. Add this test case that tests the custom VIN validator.

```
it("Should validate input", () => {
  let input = {
    veh_vin: "V1000",
    veh_year: 2019,
    veh_make: "BMW",
    veh_model: "330i",
    veh_mileage: 25000,
    veh_price: 22000,
    veh_featured: true
  }

  //Simulate user input
  component.vehicleForm.setValue(input)

  expect(component.vehicleForm.valid).toBeTrue()

  input.veh_vin = "V1QQQQ" //Invalid
  component.vehicleForm.setValue(input)
  expect(component.vehicleForm.valid).toBeFalse()
})
```

___3. Save.

___4. In the test browser click **Spec List** and make sure all tests are passing.

VehicleFormReactiveComponent

- should create
- Should validate input

Part 6 - Test the Dealer Inventory Component

Let's suppose we want to test the `addVehicle()` method of `DealerInventoryComponent`. This is where things get complicated. Let's look at the method.

___1. Open `src/app/dealer-inventory/dealer-inventory.component.ts`.

__2. Look at the addVehicle() method.

```
addVehicle(v:Vehicle) {  
  this.inventorySvc.addVehicle(v).subscribe(() => {  
    this.inventory.push(v)  
  })  
}
```

Once the Observable returned by inventorySvc.addVehicle(v) is subscribed nobody else can subscribe to it again. This makes it very difficult for any test script to figure out exactly when the backend service call completes before doing any test validation. The done() function approach wont work here.

Angular provides two solutions to this problem:

1. The async/whenStable approach lets a test script wait for the Observable to fire (indicating the completion of an HTTP call).
2. The fakeAsync/tick approach lets a test script wait for certain number of seconds and hope that the Observable will fire within that time. This approach does not allow any XMLHttpRequest calls. You must mock a backend service.

In this lab we will explore the async/whenStable approach.

__3. Open **src/app/dealer-inventory/dealer-inventory.component.spec.ts**.

__4. Add waitForAsync to the first import so it looks like this:

```
import { ComponentFixture, TestBed, waitForAsync } from '@angular/core/  
testing';
```

__5. Add these import statements. The last one is because we are still using the French locale.

```
import { HttpClientModule } from '@angular/common/http';  
import { Vehicle } from '../vehicle';  
import '@angular/common/locales/global/fr';
```

__6. Import HttpClientModule module in the test module.

```
beforeEach(async(() => {  
  TestBed.configureTestingModule({  
    imports: [HttpClientModule],  
    declarations: [ DealerInventoryComponent ]  
  })  
  .compileComponents();  
}));
```

___7. Add this test case.

```
it('Should add vehicle', waitForAsync(() => {
  let v = new Vehicle(
    "V3000", 2018, "BMW", "750i", 25000, 54000, false, [])

  //Wait for ngOnInit to complete
  fixture.whenStable().then(() => {
    component.addVehicle(v)

    //Wait for addVehicle to complete
    fixture.whenStable().then(() => {
      //Now do validation
      let car = component.inventory.find(item => item.VIN === v.VIN)!
      fixture.detectChanges();
      expect(car).toBeDefined()
      expect(car.make).toBe("BMW")
    })
  })
}))
```

Take a moment to understand:

- The entire test script is wrapped in an `async()` call. This is needed for us to call `whenStable()`.
- We had to call `fixture.whenStable()` twice. We want the call to `getInventory()` made from `ngOnInit()` to finish before calling `addVehicle()`. Otherwise, if the `getInventory()` call completes after `addVehicle()` then the component's internal state (the inventory variable) will be inconsistent and invalid.

___8. Save.

___9. Verify that all tests for the component passes.

```
DealerInventoryComponent
• Should add vehicle
• should create
```

Part 7 - Extra Credit

Write a test cases for these methods of `DealerInventoryComponent`:

- `deleteVehicle`
- `commitEdit`

Part 8 - Clean Up

__1. Finish testing by hitting Control+C in the VS Studio terminal. This will also close the test web browser window.

Part 9 - Review

In this lab we learned how to:

- Unit test services. We used the `done()` function to test asynchronous methods that return an Observable.
- Unit test reactive forms.
- Unit test components that make asynchronous calls to a service. We used the `async/whenStable` combo approach for this.

Lab 19 - Debugging Angular Applications

In this lab you will get hands-on experience with:

- Template parse errors
- Typescript code errors
- Accessing components at runtime with `ng.probe()`
- Breakpointing in Typescript code

We will use the ongoing book database application (\rest-client) project. Open the project in your editor.

Part 1 - Get Started

- __ 1. Shutdown all VS Code window.
- __ 2. Terminate any backend web service server running.
- __ 3. Open the **C:\LabWork\rest-client** folder from VS Code.
- __ 4. Open a terminal in VS Code.
- __ 5. From the terminal run:

```
npm start
```

- __ 6. From **C:\LabFiles\book-server** folder start the backend web service server.

```
npm start
```

- __ 7. Open a browser and enter the URL: **http://localhost:4200** and make sure you can use the app normally.
- __ 8. Open the browser developer tool (F12) and the Console pane. We will use the console a lot.

Part 2 - Simulate Typescript Compile Errors

- __ 1. In VS Code open **src/app/edit-book/edit-book.component.ts**.

__2. In the EditBookComponent class, add a variable with obvious compilation error.

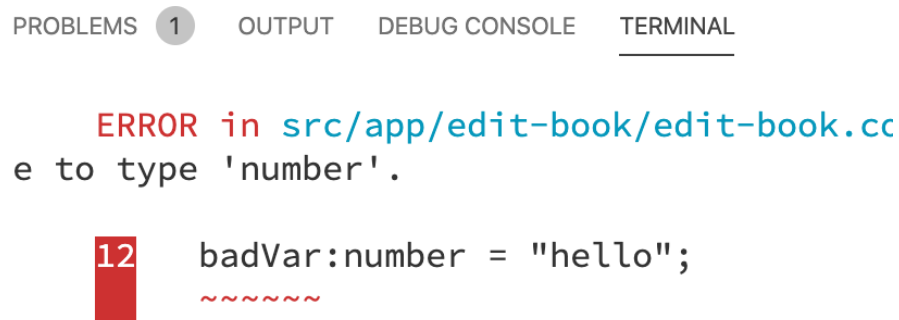
```
badVar:number = "hello";
```

You can add this var before the "constructor()" line.

This statement tries to assign a string to a number type variable. It should produce a TypeScript compile error.

__3. Save changes.

__4. Compile problems are reported in the development server console. You should see this message:



The screenshot shows the VS Code development console with the 'PROBLEMS' tab selected. It displays a TypeScript error: 'ERROR in src/app/edit-book/edit-book.cce to type 'number''. Below the error message, the code snippet is shown with a red squiggly line under the string 'hello' in the line 'badVar:number = "hello";'. The line number 12 is highlighted in a red box.

__5. Also verify that the browser was not updated. The development server does not push code if build fails.

__6. Remove the newly added variable and save changes and make sure that there are no more compile problems.

Part 3 - Simulate Template Errors

When it comes to syntax error in templates some issues are reported by the dev server, some by the browser console and some are not reported at all. Let's explore this.

__1. Open **book-list/book-list.component.html**.

__2. Change routerLink to routerLinkxxx.

```
<button class="btn-primary" [routerLinkxxx]="['/add-book']">Add a Book</button>
```

__3. Save.

__4. You should see the problem reported by the dev server.

ERROR in src/app/book-list/book-list.component.html:
nce it isn't a known property of 'button'.

3 <button class="btn-primary" [routerLinkxxx]="[
~~~~~

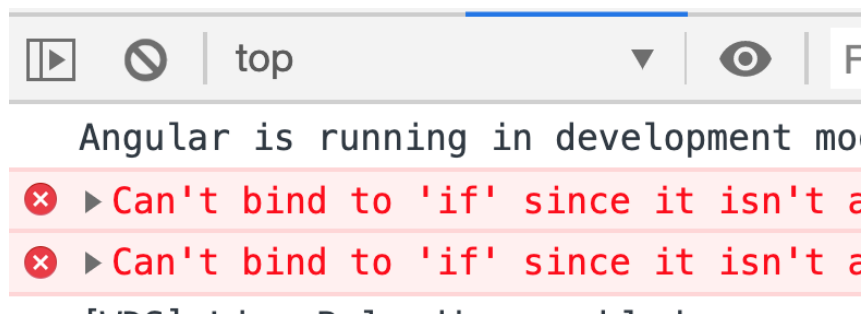
\_\_5. Fix the problem and save.

\_\_6. Change the first **\*ngIf** to **\*if**.

```
<div *if="books.length > 0">
```

\_\_7. Save.

\_\_8. Dev server does not show any error. But in browser's console you will see:



### Important

During development always watch the dev server output as well as the browser console. Problems can be reported in either of these places.

\_\_9. Fix the problem (change **if** back to **ngIf**) and save.

\_\_10. Change book.isbn to book.isbnxxx.

```
{{book.isbnxxx}} - {{book.title}}
```

\_\_11. Save.

\_\_12. This problem will be detected by the dev server and by the browser console.

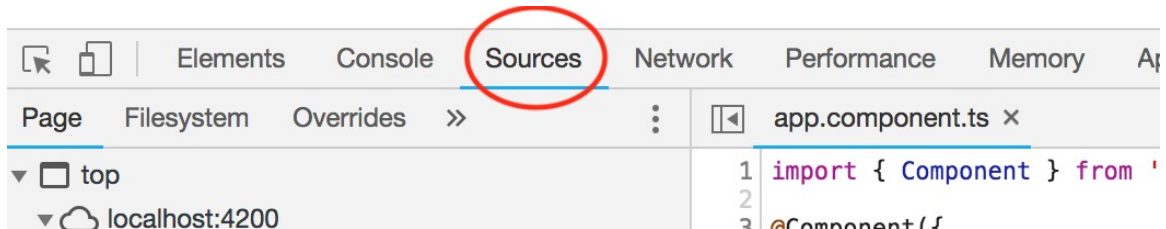
\_\_13. Fix the problem and save.

## Part 4 - Putting a Break Point

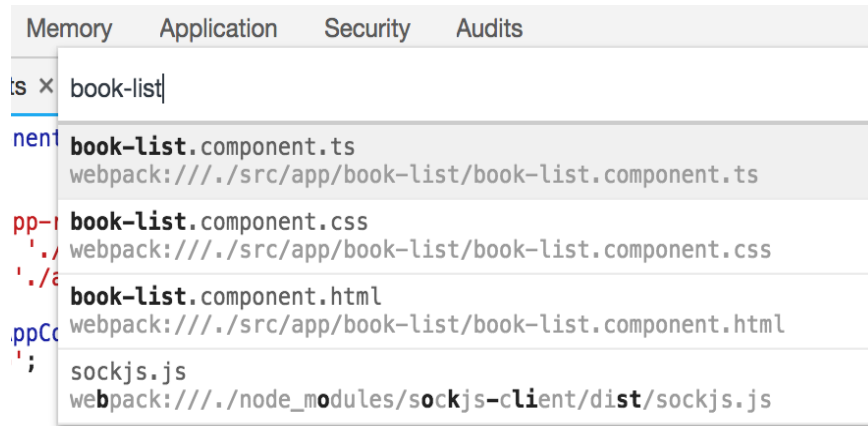
Learning how to put a break point in Typescript code is an essential skill. This will help you investigate difficult problems using the debugger tool.

\_\_1. Open **book-list/book-list.component.ts** in an editor. We will put a break point in the `deleteBook()` method.

\_\_2. In the browser's developer tool, click the **Sources** tab.



\_\_3. To search for a file hit Control+P (Cmd+P in Mac). Start typing **book-list**.



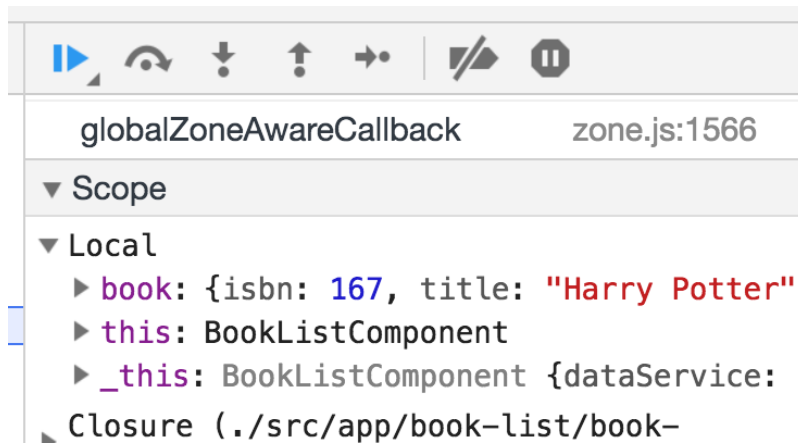
\_\_4. Pick **book-list.component.ts**.


\_\_5. Put a break point in the first line of `deleteBook()` method. (line numbers in your code may differ from those in the screenshot below)

```
20 |
21 | deleteBook(book: Book) {
22 |     if (!window.confirm('Are you sure you want to delete this book?')) {
23 |         return
24 |     }
25 |
26 |     this.dataService.deleteBook(book.id)
27 |     //Delete local copy of book
```

\_\_6. In the application, click the **DELETE** button for a book. You should now halt at the break point.

\_\_7. In the variables pane under local **Scope** you should be able to inspect the **book** variable. The **this** variable points to the instance of BookListComponent.



\_\_8. Hit the  resume button to continue. Click OK or Cancel.

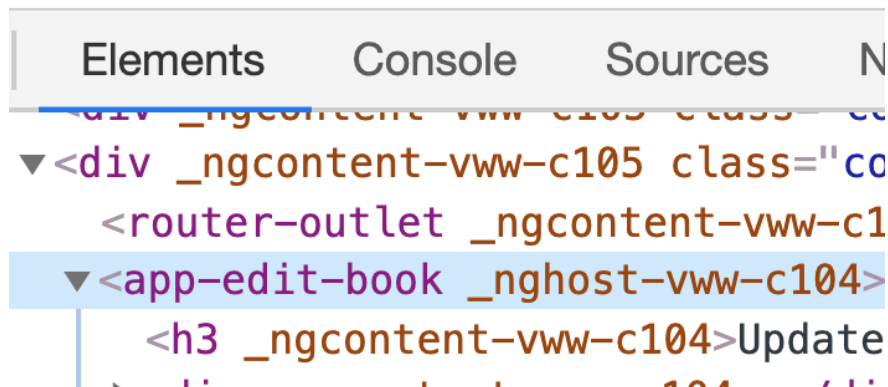
## Part 5 - Visually Inspect a Component

This is an easy way to inspect the internal state of a component without using a debugger. We will now inspect the instance of EditBookComponent.

\_\_1. In the application, click the **EDIT** button for a book. You will now be routed to the EditBookComponent. We will inspect its internal state.

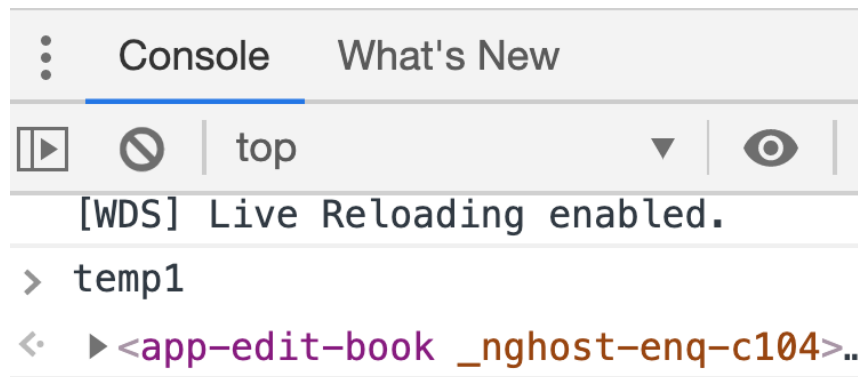
\_\_2. In the browser's developer tool click the **Elements** tab.

\_\_3. Hit Control+F and search for **app-edit-book** and locate the **<app-edit-book>** tag.



\_\_4. Right click the tag and from the menu select **Store as Global Variable**.

\_\_5. This will save the DOM Element object as a variable. By default the name will be **temp1**. (Next variable will be temp2 and so on).



\_\_6. In the **Console** type this and hit enter:

```
ng.getComponent(temp1)
```

\_\_7. You should now be able to expand the instance of EditBookComponent and look at its internal state (such as the **book** variable).

```
> ng.getComponent(temp1)
< EditBookComponent {dataService: DataService, ac
  ▶ activeRoute: ActivatedRoute {url: BehaviorSub
  ▶ book: {isbn: 167, title: "Harry Potter", price
  ▶ dataService: DataService {http: HttpClient, b
  ▶ router: Router {urlSerializer: DefaultUrlSeria
```

## Part 6 - Clean Up

\_\_1. In VS Code terminal hit Control+C to close the dev server.

\_\_2. Shutdown the backend web service.

## Part 7 - Review

In the lab we learned a few common techniques to investigate errors in Angular applications.