WA2890 Comprehensive Angular 8 Programming

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Introduction to Angular

In this lab, we will build a Hello World style Angular component. The key focus is to learn how to install all the required software, create a very simple 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.

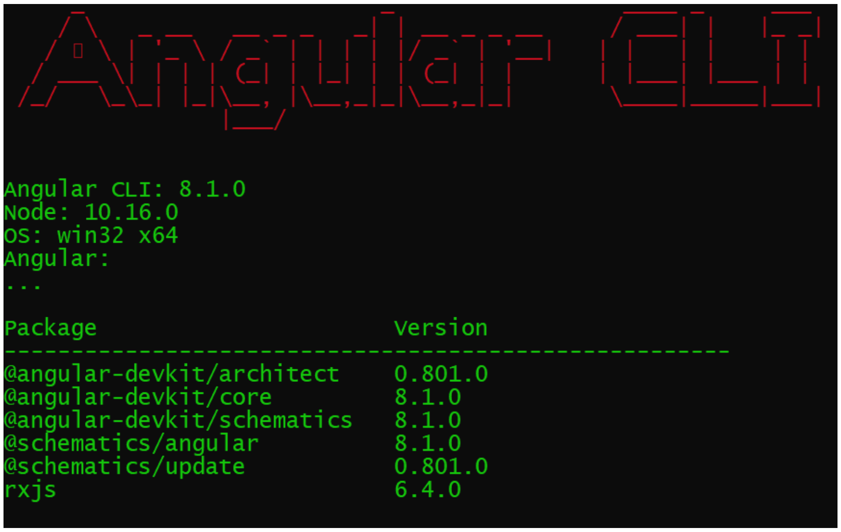
```
npm install -g @angular/cli@8.1.0
```

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.

__3. It should take a few minutes to install the tool. After installation finishes enter this command to verify everything is OK.

```
ng --version
```

Troubleshooting
Some classroom environments do not allow Internet access. In which case you will not be able to install Angular CLI. That is OK. We will provide alternate instructions for the lab steps to work in that situation.



```
     _                      _                 ____ _     ___
    / \   _ __   __ _ _   _| | __ _ _ __     / ___| |   |_ _|
   / □ \ | '_ \ / _` | | | | |/ _` | '__|   | |   | |    | |
  / ___ \| | | | (_| | |_| | | (_| | |      | |___| |___ | |
 /_/   \_\_| |_|\__, |\__,_|_|\__,_|_|       \____|_____|___|
                |___/


Angular CLI: 8.1.0
Node: 10.16.0
OS: win32 x64
Angular:
...

Package                           Version
-----------------------------------------------------------
@angular-devkit/architect         0.801.0
@angular-devkit/core              8.1.0
@angular-devkit/schematics        8.1.0
@schematics/angular               8.1.0
@schematics/update                0.801.0
rxjs                              6.4.0
```

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.

__3. Run this command to create a new project called **hello**.

```
ng new hello --defaults
```

No Internet Access?
Copy C:\LabFiles\project-seed.zip to the C:\LabWork folder.
Extract C:\LabWork\project-seed.zip.
Make sure now you have the C:\LabWork\project-seed folder.
Rename the C:\LabWork\project-seed to C:\LabWork\hello.

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

```
C:\LabWork\hello
└───src
    ├───app
    ├───assets
    └───environments
```

We are not quite ready to run this project. Before we do that, let's get to know more about the boiler plate code that is given to you.

Part 3 - The Anatomy of a Simple Angular Project

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. Manually downloading these modules can be cumbersome and error prone. This is why we use Node Package Manager (NPM) to download them.

You can use your favorite editor to edit code during these labs. For your convenience we have included the Visual Studio Code editor from Microsoft which is free to downloaded and use. VS Code is written in TypeScript and has out of the box support for that language. Which makes a great free editor for Angular development.

__1. Open **package.json** from the **hello** folder in an editor.

This 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.

__2. In a command prompt window go to the **C:\LabWork\hello** directory.

```
cd C:\LabWork\hello
```

__3. Run the following command to list the installed packages.

```
npm list --depth=0
```

__4. You will see some errors but scroll up and verify you get something like this [version may be different]

```
hello@0.0.0 C:\LabWorkAng8\hello
+-- @angular-devkit/build-angular@0.801.0
+-- @angular/animations@8.1.0
+-- @angular/cli@8.1.0
+-- @angular/common@8.1.0
+-- @angular/compiler@8.1.0
+-- @angular/compiler-cli@8.1.0
...
```

The **ng new** command has run NPM and installed these packages for us already.

__5. These packages are physically located inside the **node_modules** folder. Have a quick look in the folder.

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.

__6. Open the component class **hello/src/app/app.component.ts** in an editor.

```
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.

__7. Briefly look at the **app.component.html** file. This is the template code of AppComponent. A template is used to dynamically generate HTML in the browser. It looks like a regular HTML. But then it has things like this:

```
  <h1>
    Welcome to {{ title }}!
  </h1>
```

That **{{ title }}** piece will render the current value of the **title** variable of the AppComponent class instance.

Templates may look like plain HTML, but they actually get compiled into code. This is necessary for the template to dynamically display data and contain other display logic.

__8. Next open the application module file **hello/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.

__9. Next open **hello/src/main.ts**. This file bootstraps or attaches the application module to the browser.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';

...

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));
```

When the AppModule is bootstrapped, Angular goes through index.html and looks for the selectors for the bootstrappable components in the module. If found, an instance of the component is then inserted into the DOM of index.html where the matching selector is located. In this case AppComponent is displayed inside the index.html page wherever the "my-app" tag appears.

Did You Know?
Here we are bootstrapping our application module into the browser. But Angular can also be used to write native mobile applications for Android and iOS. In that case we will use something else other than platformBrowserDynamic() to bring the module into display.

__10. Finally, open **hello/src/index.html**.

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8">
 <title>Hello</title>
 <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

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 Java Script 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 4 - Run the Development Server.

During development you should access your Angular application site using the Angular CLI provided web server. It has many benefits. Among which are:

- Rapidly rebuild the application when you modify any file.
- Automatically refresh the browser after build finishes.

__1. From the command prompt go to the root folder of our project **C:\LabWork\hello**.

__2. Run this command to compile TypeScript.

```
npm start
```

Alternatively you could also enter ng serve. But npm start has an advantage. It lets you specify extra arguments to ng serve in package.json. It's better to get used using npm start.

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

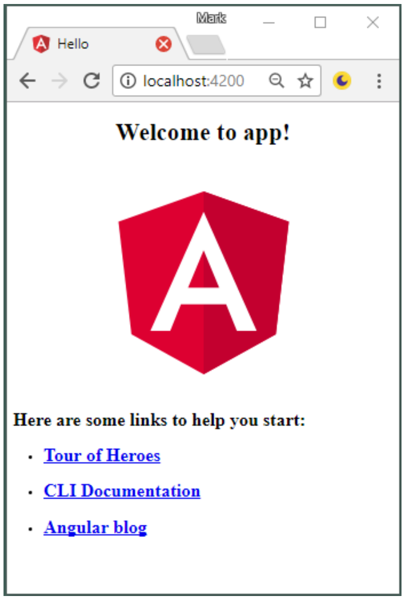
```
chunk {main} main.js, main.js.map (main) 9.74 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 251 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.09 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 16.3 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.8 MB [initial] [rendered]
Date: 2019-07-08T14:30:50.306Z - Hash: f72f630d7028c10a3f49 - Time: 7187ms
** Angular Live Development Server is listening on localhost:4200, open your browser …
i □wdm□: Compiled successfully.
```

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

__3. 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:



Welcome to app!

Here are some links to help you start:

- Tour of Heroes
- CLI Documentation
- Angular blog

__4. Leave the browser and the server running for the next section.

__5. 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 5 - Change the Component Code

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

__1. Open **hello/src/app/app.component.ts** in an editor.

__2. Replace the 'title' property with a 'name' property. Give the 'name' property a value of 'Android'. When you are done the class portion of the file should look like this:

```
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>
```

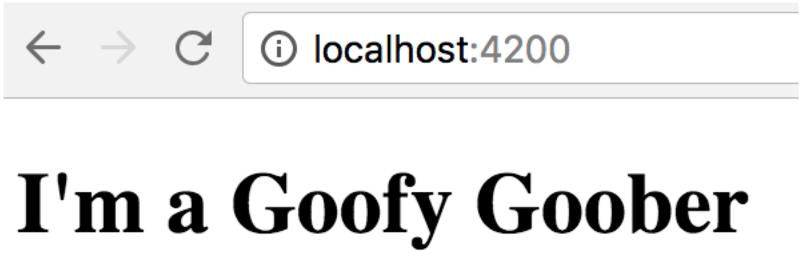
__6. Save the file.

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. This is an example of one-way data binding where data is taken from the component and shown in DOM.

__7. 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.

__8. 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

__9. Let's clean up. First, close out the server by entering Ctrl-C in the command prompt window. Then close out the browser tab where you were viewing the application.

__10. Close any open files and command prompt windows.

Part 6 - Review

Our goal in this lab was to take a look at a basic Angular application and go through the workflow of building and testing it.

First, we learned that our code is written in TypeScript which needs to be transpiled into JavaScript.

Next, we saw how a build system is used to:

- Transpile TypeScript,
- Create *.js bundle files for the app,
- Serve the index.html and bundle files

Finally we accessed the application using a browser.

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 - Lab Setup

For this lab, we will create a new directory and install TypeScript.

__1. Within the **C:\LabWork** directory create a subdirectory named **'tslab'** the result should be the following:

```
C:\LabWork\tslab
```

__2. Open a command prompt window and navigate into the '**tslab**' directory.

__3. Execute the following to create a new package.json file. The command will prompt you for more information. Take all the default values and proceed until the command is complete:

```
npm init
```

__4. Install TypeScript into the local project directory ('tslab') by executing the following command:

```
npm install typescript@3.4.3 --save-dev
```

You can ignore any WARN messages.

__5. Copy a **tsconfig.json** file from the **C:\LabFiles** directory into the **C:\LabWork\tslab** directory. This file configures the TypeScript compiler and must be present in the folder where the compiler is run from.

__6. Edit the **package.json** file.

__7. Add the line shown in bold below to the scripts section of the file (don't forget the comma):

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "tsc": "./node_modules/.bin/tsc"
 },
```

__8. Save the file.

__9. Try running tsc from inside the 'tslab' directory using the following command:

```
npm run tsc
```

__10. You should get the following error indicating that no source files were found. We will be creating a source file in the next section:

```
error TS18003: No inputs were found in config file...
```

Note: In this lab, we are calling the TypeScript compiler directly from the command prompt. In other labs, the same compilation step is executed behind the scenes by the Angular build system.

Part 2 - Variable Types

__1. Within the **C:\LabWork\tslab** directory create a file called **play.ts**. Make sure the file does not have an extension like '.txt' automatically added to it.

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

```
var age: number = 20

console.log(age)
```

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

__3. Save changes.

__4. From the command prompt go to the **C:\LabWork\tslab** directory. Then run this command to compile the code.

```
npm run tsc
```

If all goes well the command you should see the following output:

```
C:\LabWork\tslab>npm run tsc

> tslab@1.0.0 tsc C:\LabWork\tslab
> tsc
```

__5. The above command will transpile the play.ts file and place the result in the ./out directory. Do a directory listing to confirm this:

```
C:\LabWork\tslab>dir /b out
play.js
play.js.map
```

__6. Enter the following command to run the code:

```
node out/play.js
```

The output should look like this:

```
C:\LabWork\tslab>node out/play.js
20
```

__7. Now we will deliberately introduce a type error. In play.ts initialize the age variable like this.

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

__8. Save play.ts.

__9. Transpile the code again using the tsc command:

```
npm run tsc
```

This time the results will be an error like this:

```
C:\LabWork\tslab>npm run tsc
play.ts(2,5): error TS2322: Type '"too old"' is not assignable to type
'number'.
```

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

Part 3 - Function Argument Types

__1. Delete or comment out all lines in **play.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. Compile the code.

```
npm run tsc
```

__5. Run the file using the following command:

```
node out/play.js
```

The output should look like this:

```
C:\LabWork\tslab>node out/play.js
Name: Billy age: 8
```

__6. Edit play.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")
```

__7. Save and run the compiler again and make sure that you see an error message.

Part 4 - Class and Inheritance

We will learn about class and inheritance in this section. We will model various products sold by a travel company like tours, shows, and dining.

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

__2. Add a class like this.

```
class Product {
  title: string;
  price: number;
  id: number;

  constructor(id: number) {
    this.id = id
  }

  printDetails() {
    console.log(`Title: ${this.title}`)
    console.log(`ID: ${this.id}`)
    console.log(`Price: ${this.price}`)
  }
}
```

__3. Add another class that inherits from Product.

```
class Tour extends Product {
  duration: string;

  constructor(id: number, duration: string) {
    super(id);

    this.duration = duration
  }

  printDetails() {
    super.printDetails()

    console.log(`Duration: ${this.duration}`)
  }
}
```

__4. Write a function named 'test' that takes a Product as an argument. Note: This is a standard function and not part of the classes you just created. Make sure the 'test' function is not inside the brackets of either of the classes you just created.

```
function test(p: Product) {
  p.printDetails()
}
```

__5. Next, create an instance of Tour and call the 'test' method. This code should also be on its own outside of any class definition.

```
var t = new Tour(1, "8 hours")

t.title = "Trip to the Taj Mahal"
t.price = 1200.00

test(t)
```

The key thing to observe here is that even though test() takes as an argument a Product, it will actually invoke the printDetails() method of the Tour class.

Also, you may notice that the lines of code above do not end with a semicolon. This is perfectly alright. In these cases, JavaScript uses the carriage return at the end of the line to determines the end of the command.

__6. Save changes

__7. Compile the code.

__8. Run the code:

```
node out/play.js
```

The output should look like this:

```
C:\LabWork\tslab>node out/play.js
Title: Trip to the Taj Mahal
ID: 1
Price: 1200
Duration: 8 hours
```

__9. You can do this optional step all on your own. Add a class called **Dining** that extends Product and has these fields:

- **cuisine** of type string.
- **childPrice** of type number.

Create the constructor and a reasonable implementation of the 'printDetails' method.

Create an instance of it and pass it to the test() function.

This ability for a Tour object to appear as Product is called polymorphism. Inheritance is one of the ways to achieve polymorphism. The other being interface. We will get into that next.

Part 5 - Interface

An interface in TS describes the shape of an object. An interface lists a set of variables and methods. An object or class claiming to conform to that interface must declare those variables and methods.

In our ongoing example, we have received a few new requirements.

- Products like Tour and Dining are bookable. We need to store a list of available dates for them. During booking a user will select a date.
- A Tour is cancelable and there is a cancelation fee associated with it. There may be other cancelable products in the future.

We realize that there may be similar requirements coming in the future. These features (bookable, cancelable etc.) can be associated with products in an independent manner. This makes it hard to model them using inheritance alone (TypeScript allows inheriting from a single class only). Interface is the right approach here.

__1. Create an interface called Bookable like this.

```
interface Bookable {
    availableDates: [Date]
}
```

__2. Create an interface called Cancelable like this.

```
interface Cancelable {
    cancelationFee: number
}
```

__3. Make the Tour class implement these two interfaces. This is shown in bold below.

```
class Tour extends Product implements Bookable, Cancelable {
...
```

__4. Add these two fields to the Tour class.

```
availableDates: [Date]
cancelationFee: number
```

__5. Save play.ts.

__6. Compile the code and make sure there are no errors.

__7. Add the following as a global method (one that is outside of any classes at the global level) that cancels a booking:

```
function cancelBooking(c: Cancelable) {
  console.log("Canceling booking. Charges: %d", c.cancelationFee)
}
```

__8. Add these lines of code in bold at the end of the file to test the cancelBooking() function.

```
var t = new Tour(1, "8 hours")

t.title = "Trip to the Taj Mahal"
t.price = 1200.00
t.cancelationFee = 100.00

cancelBooking(t)
```

__9. Save. Let the file compile. Then run. The output should be:

```
C:\LabWork\tslab>node out/play.js
Canceling booking. Charges: 100
Title: Trip to the Taj Mahal
ID: 1
Price: 1200
Duration: 8 hours
```

A Tour class object can now appear as a Product, a bookable item, and a cancelable item. The last two polymorphic behaviors are achieved using interfaces.

Part 6 - Generics

Currently, our cancelBooking() method takes as input a Cancelable object. Unfortunately, we do not have access to the original price of the product or the product ID. Without knowing that we can not properly issue a refund. The correct data type should be a Product that is also Cancelable. How do we model that? Generics constraints will come to the rescue.

__1. Make a copy of the cancelBooking function and call it cancelBooking2

__2. Change the signature of cancelBooking2() as shown in bold face below.

```
function cancelBooking2<T extends Cancelable & Product>(c: T) {
```

Here T is a generic place holder for a type. As per our constraints, T must extend Product and implement Cancelable.

__3. Modify the body of the cancelBooking2 function as shown below.

```
function cancelBooking2<T extends Cancelable & Product>(c: T) {
  console.log("Canceling: %s (%d)", c.title, c.id)
  console.log("Price: %d", c.price)
  console.log("Cancelation fee: %d", c.cancelationFee)
  console.log("Total refund: %d", c.price - c.cancelationFee)
}
```

__4. Add the following call to cancelBooking2 at the end of the file and comment the previous call:

```
//cancelBooking(t);
cancelBooking2(t);
```

__5. Delete or comment the call to test.

```
//test(t)
```

__6. Save, compile and run. You should see this output.

```
Canceling: Trip to the Taj Mahal (1)
Price: 1200
Cancelation fee: 100
Total refund: 1100
```

Generics let you work with completely unknown data types from a method or class. Optionally, as we did here, you can put certain constraints to that type. The end result is a mix of flexibility and type safety.

Part 7 - Implementing an Interface from an Object

Previously we have seen how a class implements an interface. In TS an object can also conform to an interface. This comes in handy for type safe programming.

Consider a typical JavaScript scenario where you supply several configuration options in an object.

```
configSomething({
    directory: "/foo",
    file: "bar.txt",
    maxSize: 1024
});
```

JavaScript cannot enforce any kind of structure to this object. In TS we can specify that this object must conform to an interface.

__1. Let's say that in the example above, **directory** and **file** are mandatory fields. And **maxSize** is optional. In **play.ts** add an interface like this to model that.

```
 interface ConfigOption {
     directory: string
     file: string
     maxSize?: number
 }
```

That ? after maxSize makes it an optional.

__2. Add a global method like this.

```
function configSomething(op: ConfigOption) {
    op.maxSize = op.maxSize || 1024

    console.log("Directory: %s", op.directory)
    console.log("File: %s", op.file)
    console.log("Max size: %s", op.maxSize)
}
```

__3. Call that method like this.

```
configSomething({
    directory: "/dir1",
    file: "persons.json"
})
```

__4. Comment the call to 'cancelBooking2'.

__5. Save and let the file compile.

__6. Run the code.

```
node out/play.js
```

__7. The output should be:

```
Directory: /dir1
File: persons.json
Max size: 1024
```

__8. Now introduce an error by changing the property name "file" to "path" as shown in bold below:

```
configSomething({
    directory: "/dir1",
    path: "persons.json"
})
```

__9. Save and let compile. You should get a compilation error like this:

```
... error TS2345: Argument of type '{ directory: string; path:
string; }' is not assignable to parameter of type 'ConfigOption'.
  Object literal may only specify known properties, and 'path' does not
exist in type 'ConfigOption'
```

__10. Fix the problem, save and compile.

__11. Close all open files and command prompts.

Part 8 - Review

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

In summary:

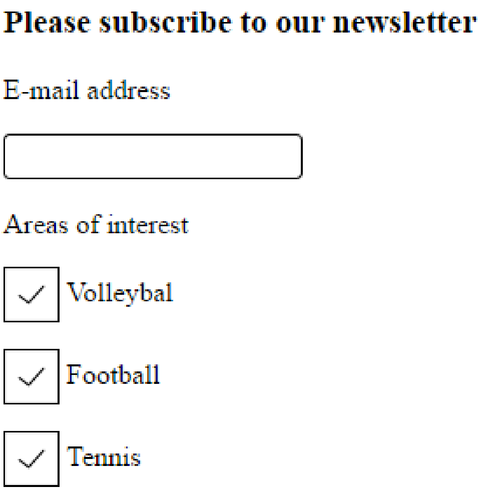
- Variables and function parameters can have strong typing.
- You can develop classes and model inheritance between them.
- A class can inherit from only one class but implement many interfaces.
- Both inheritance and interface implementation lead to polymorphism.
- Generics are a powerful way to write classes and methods without knowing the exact nature of the types. This is mostly useful for framework and library developers who may not anticipate all possible scenarios in which their code may be used. Optionally, you can define constraints for added type safety.

Lab 3 - Introduction to Components

Components are the basic building blocks in Angular. They are used to render HTML on a page and handle events. They can implement something small like a specially styled text box to something as large as an entire page. If we are to compare this with a house then the bricks, rooms and the whole house can be modeled using components.

Part 1 - Business Requirements

Over the next few labs, we will build a newsletter subscription form.



Please subscribe to our newsletter

E-mail address

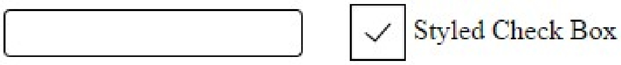
Areas of interest

☑ Volleybal

☑ Football

☑ Tennis

In this lab, we will build a few very simple components. We will create a stylized text box with rounded corners and a stylized checkbox.



☑ Styled Check Box

The goal of this lab is to mainly understand how we can build components and use them in a page.

Part 2 - Create the Project

__1. Open a command prompt window.

__2. Go to the **C:\LabWork** folder.

__3. Run this command to create a new project called **subscription**.

```
ng new subscription --defaults
```

No Internet Access?

Extract C:\LabWork\project-seed.zip.

Make sure now you have the C:\LabWork\ project-seed folder.

Rename the C:\LabWork\project-seed to C:\LabWork\subscription.

Part 3 - Create the Text Box Component

Our text box is quite simple really. It's just a regular text box with a rounded border. But first, we will just create the component without worrying about styles. Make sure that our basic code is working. Then we will add styles to the component.

Note: In real life, you will use the **ng generate component** command to quickly create a component. But in this lab, we will do everything by hand. This way we know exactly what it takes to create a component.

__1. In the **C:\LabWork/subscription/src/app** folder, create a folder called **fancy-text**

According to the Angular style guide, each component is created in its own folder within the app folder. The guide is merely a suggestion but we should follow it as much as possible. If nothing else it adds consistency. Other developers will know where to look for files within the project.

__2. In the newly created '**fancy-text**' folder, create a file called **fancy-text.component.ts**

__3. Within this file, add this code to create the basic component.

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

@Component({
  selector: "fancy-text",
  template: "<input type='text' style=''/>"
})
export class FancyText {
}
```

Let's quickly recap the basic process of writing a component. We create a TypeScript class and decorate that with the @Component decorator. At a minimum we need to specify the selector and the HTML template. The selector is the tag used to insert the component in a page. And, the template renders the component.

In this case, we are directly entering the template code within the @Component decorator using the **template** attribute. In real life, for more complex components, it is usually better to write the template in an external file and point to it from the @Component decorator using the **templateUrl** attribute.

> Note: HTML5 specifies that elements with empty bodies should not use an end tag which is why we use <input … /> here instead of <input ...></input>.

__4. Save changes.

Part 4 - Add Component to Module

A component must belong to a module. We will now add our component to the main application module.

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

__2. Import the component class by adding this to the top of the file.

```
import { FancyText } from './fancy-text/fancy-text.component';
```

__3. Add the component to **declarations** array.

```
  declarations: [
    AppComponent,
    FancyText
  ],
```

__4. Save changes.

Part 5 - Use the Component

We will now use our component to make sure everything is working.

__1. Open **subscription/src/app/app.component.html**.

__2. Delete the entire content of the file.

__3. Enter this line to use the FancyText component.

```
<fancy-text></fancy-text>
```

Note: You must use an end tag for a component in a template. Using <fancy-text/> will not work.

__4. Save changes.

Part 6 - Test Changes

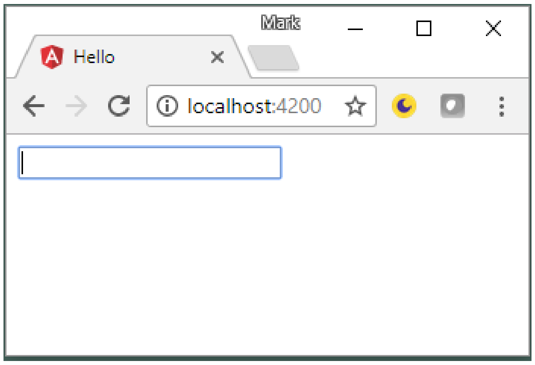
__1. From the **C:\LabWork\subscription** folder run the following command. It will compile the code and start the server. The first time this command may take several seconds to run.

```
npm start
```

__2. Now, open up the following URL in the Chrome browser:

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

You should see something like this:



__3. Verify that you see the text box after few seconds. If you don't see the text box open the JavaScript console (hit F12) and inspect the error messages.

Part 7 - Apply Styles to the Text Box

Awesome. We are now ready to make our fancy text box fancy for real. But we will do so gradually. First, we will set the style directly in the template.

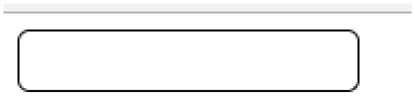
__1. Open fancy-text.component.ts.

__2. Add the styles as shown in bold below. The style should all be on one line although it wraps below.

```
@Component({
  selector: "fancy-text",
  template: "<input type='text' style='border-color:black; border-
width:thin; border-radius:6px; height:20pt;'/>"
})
```

__3. Save the file.

__4. Back in the command prompt, you should see the new code automatically being compiled, and the web browser should refresh automatically. You should see the rounded corners.



Right now the styles are working fine. But there is a problem. The styles are not reusable. Also, the style is cluttering up the template. To solve this problem, we need to externalize the styles from the tags similar to the way CSS works. There are several ways to do this in Angular. We will start by using the **styles** property of the @Component decorator.

__5. Change the @Component decorator as shown below.

```
@Component({
  selector: "fancy-text",
  styles: ["input[type='text'] {border-color:black; border-width:thin;
border-radius:6px; height:20pt;}"],
  template: "<input type='text'/>"
})
```

The styles property is an array of CSS style rules.

__6. Save changes.

__7. Check there are no issues recompiling the code and the web browser should refresh automatically. The text box should look same as before.

Important: A huge advantage of setting styles like this is that these styles are only applicable to this specific component. Other components can define their own style rules for input type text. They will not collide with each other. This makes components highly self sufficient and reusable.

We have improved the way styles are set for our text box. But there's still a problem. In real life, designers will hand us CSS files (or Less/Sass files that are compiled into CSS). It will be tedious and error prone to copy paste these styles into our TS code. It will be better if a component could refer to an external style file. We will now do exactly that.

__8. In the **src/app/fancy-text** folder, create a new file called **fancy-text.component.css**

__9. In the CSS file, add this style rule.

```
input[type='text'] {
    border-color:black;
    border-width:thin;
    border-radius:6px;
    height:20pt;
}
```

__10. Save changes.

__11. In **fancy-text.component.ts**, from the @Component decorator refer to the CSS file like this. Note the 'styles' property is removed and replaced with the 'styleUrls'.

```
@Component({
  selector: "fancy-text",
  styleUrls: ["./fancy-text.component.css"],
  template: "<input type='text'/>"
})
```

__12. Save changes.

Several things to note about the style URLs.

- The URL for the CSS file is relative to the component declaration.
- Styles are private to this component. They will not be applied to other components. For that to work correctly, Angular build system actually compiles your CSS into JavaScript code!

__13. Check there are no issues recompiling the code and the web browser should refresh automatically. Make sure that the text box continues to get the styles.

Part 8 - Import the FancyCheckbox Component

We will now develop the styled checkbox component. Checkboxes are notoriously hard to style. We will resort to CSS trickery to use custom images to indicate the checked and unchecked states.

To save time we have already created the FancyCheckbox component. The principles used were exactly the same as we used for the text box. We will now import the component and inspect its code.

__1. From **C:\LabFiles** folder, copy the **fancy-checkbox** folder into **C:\LabWork\subscription\src\app** folder.

__2. Open **subscription/src/app/app.module.ts**

__3. Import the FancyCheckbox class.

```
import { FancyCheckbox } from
  './fancy-checkbox/fancy-checkbox.component';
```

__4. Add the component to the declarations list.

```
  declarations: [
    AppComponent,
    FancyText,
    FancyCheckbox
  ],
```

__5. Save changes.

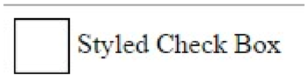
__6. Open **app.component.html**

__7. Use the component like this.

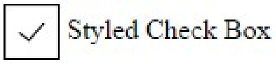
```
<fancy-checkbox></fancy-checkbox>
```

__8. Save changes.

__9. Check there are no issues recompiling the code and the web browser should refresh automatically. You should see the checkbox as shown below.



__10. Check the box.



Review the code of the component to better understand it. It's CSS is pretty intense. This component uses an image. Everything is self contained in the component's folder.

__11. In the command prompt, use '**<CTRL>-C**' to terminate the batch job.

__12. Close all open files.

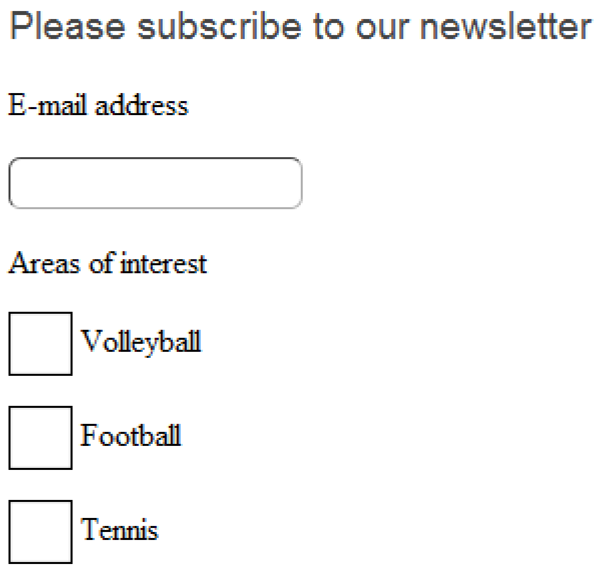
Part 9 - Review

In this lab, we developed a very simple text box component. We explored various ways to apply styles to this text box. We also implemented a styled checkbox component.

In the next few labs, we will continue to work on this application.

Lab 4 - Create the Subscription Form Component

So far, we have a text box and a checkbox component. Now let's create a new component that uses them both to implement the newsletter subscription form. This will give you an idea about how to combine multiple components to create a higher level component. At the end of the lab, the component will look like this.



Please subscribe to our newsletter

E-mail address

Areas of interest

□ Volleyball

□ Football

□ Tennis

This lab picks up where the previous lab, "Introduction to Components", left off.

Part 1 - Create the Component

__1. From the **C:\LabWork\subscription** folder, run this command.

```
ng generate component SubscriptionForm
```

If you do not have Angular CLI installed globally then run:

npm run ng generate component SubscriptionForm

__2. Verify that the component class was generated in **src/app/subscription-form/subscription-form.component.ts**

__3. Open **src/app/app.module.ts** and verify that Angular CLI has automatically added the component to the declarations list.

Part 2 - Write the Component Code

__1. Open the component template file **src/app/subscription-form/subscription-form.component.html**

__2. Change the template like this.

```
<h3>Please subscribe to our newsletter</h3>

<p>E-mail address</p>
<p><fancy-text></fancy-text></p>
<p>Areas of interest</p>
<p><fancy-checkbox></fancy-checkbox></p>
<p><fancy-checkbox></fancy-checkbox></p>
<p><fancy-checkbox></fancy-checkbox></p>
```

__3. Save changes.

Part 3 - Use the Form Component

__1. Open **src/app/app.component.html**

__2. Use the newly created component replacing the entire content.

```
<app-subscription-form></app-subscription-form>
```

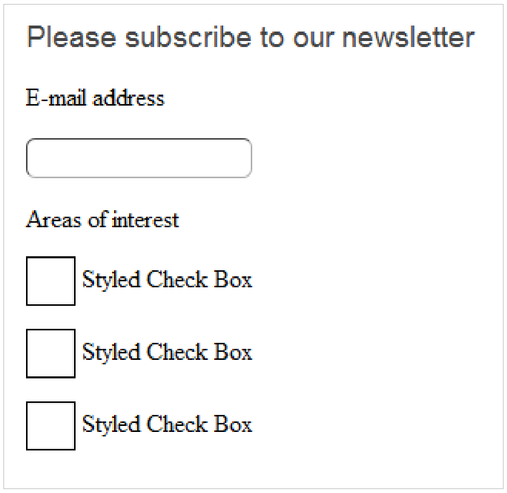
__3. Save changes.

Part 4 - Test Changes

__1. From the **C:\LabWork\subscription** folder, run this command.

```
npm start
```

__2. Open **http://localhost:4200** in your browser and verify that the page looks like the screen shot below:



Please subscribe to our newsletter

E-mail address

Areas of interest

□ Styled Check Box

□ Styled Check Box

□ Styled Check Box

Part 5 - Provide Input to a Component

Our checkbox looks and works great. But there is one flaw. The label for the checkbox is currently hard coded in the template as "Styled Check Box". That is not terribly useful. We must be able to set any label we like. We will do that now by supplying the label to the checkbox component as an input.

The input mechanism lets the user of a component set the value of the component's fields. To do this, we must decorate these fields with @Input.

> With @Input we are actually implementing one-way data binding. More on data binding comes later.

__1. Open **src/app/fancy-checkbox/fancy-checkbox.component.ts**

__2. First, import the Input decorator name from the Angular core module.

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

__3. Add a field to the FancyCheckbox class called **label** and decorate it with @Input like this.

```
export class FancyCheckbox {
    @Input() label: string
}
```

__4. Now, show the value of the label field in the template as shown in bold below.

```
@Component({
    selector: "fancy-checkbox",
    styleUrls: ["fancy-checkbox.component.css"],
    template: "<label><input
type='checkbox'/><span></span>{{label}}</label>"
})
```

__5. Save changes.

Now, when we use the checkbox component in an HTML template, we can supply our own label text.

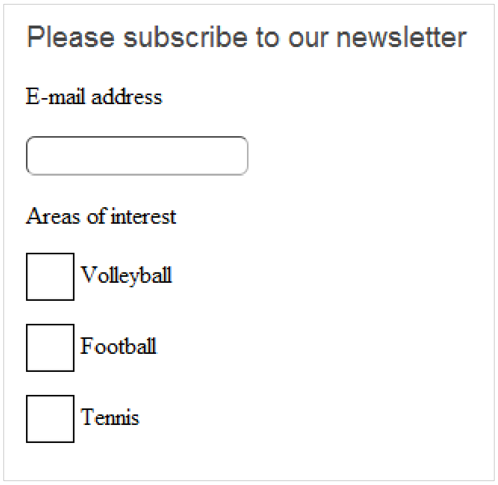
__6. Open **src/app/subscription-form/subscription-form.component.html**

__7. Supply values for the label field like shown in bold face below.

```
<p><fancy-checkbox label="Volleyball"></fancy-checkbox></p>
<p><fancy-checkbox label="Football"></fancy-checkbox></p>
<p><fancy-checkbox label="Tennis"></fancy-checkbox></p>
```

__8. Save changes.

__9. Check there are no issues recompiling the code and the web browser should refresh automatically. The page should look like this.



Please subscribe to our newsletter

E-mail address

Areas of interest

□ Volleyball

□ Football

□ Tennis

Note, the fully qualified way to supply input to a component is like this:

```
<fancy-checkbox [label]="'Volleyball'">
```

This syntax lets you supply complex data structure like objects and arrays as input. If the input is a simple string you can take the shortcut we have taken here:

```
<fancy-checkbox label="Volleyball">
```

Part 6 - Clean Up

__1. In the command prompt, hit '**<CTRL>-C**' to terminate the dev server.

__2. Close all files.

Part 7 - Review

In this lab, we built a higher level component that combined multiple lower level components to implement a form. Eventually, the whole page is implemented as a large component.

Lab 5 - Understanding Data Binding

Data binding is a crucial underpinning of the Angular architecture. Simply stated, data binding connects your application data to the web page. For example, you can display the price of a product using data binding. Users can enter text in an input field and update the value of the application's data structure.

You can bind your application's data to a DOM element, directive, and components. The syntax used in a template to setup binding is the same in all cases. But data binding to custom components and directives is a more advanced topic that we will cover later. For now, in this lab, we will focus on data binding for DOM elements and some out of the box directives that come with Angular.

Part 1 - Create the Project

__1. In the '**C:\LabWork**' folder run this command.

```
ng new binding-test --defaults
```

No Internet Access?

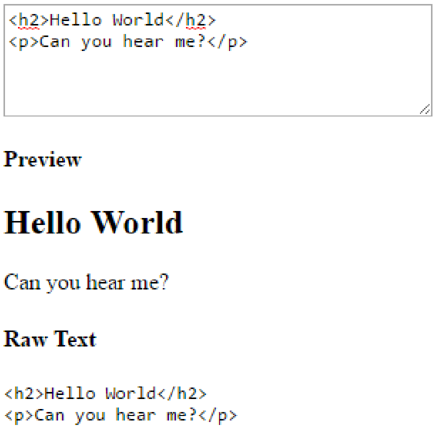
Extract C:\LabFiles\project-seed.zip into C:\LabWork

Make sure now you have the C:\LabWork\project-seed folder.

Rename the C:\LabWork\project-seed to C:\LabWork\binding-test.

Part 2 - Create an HTML Editor Component

We will now build a component that lets users enter HTML code in an editor area. The component then shows a live preview of the HTML.



```
<h2>Hello World</h2>
<p>Can you hear me?</p>
```

Preview

Hello World

Can you hear me?

Raw Text

```
<h2>Hello World</h2>
<p>Can you hear me?</p>
```

This component will help us understand how basic data binding works.

__1. From the **binding-test** folder run this command.

```
ng generate component HtmlEditor
```

If you do not have Angular CLI installed globally then run:

npm run ng generate component HtmlEditor

__2. Verify that the newly added component got added to the declarations list of the application module in **\binding-test\src\app\app.module.ts**.

__3. Open **\binding-test\src\app\html-editor\html-editor.component.ts** file.

__4. Add the userInput variable like this.

```
export class HtmlEditorComponent implements OnInit {
    userInput = "Enter HTML here"
...
```

__5. Save changes.

__6. Open **\binding-test\src\app\html-editor\html-editor.component.html** file.

__7. Set the template as follows.

```
<textarea [(ngModel)]="userInput" rows="5" cols="40">
</textarea>
```

__8. Save changes.

Take a moment to understand what is going on here. First, note that the HtmlEditor class has a field called **userInput**. That is our model or application data. Next, notice the **textarea** tag in the template. We added an attribute like this: [(ngModel)]="userInput". This is how we setup data binding in a template. But there is a lot going on in this compact syntax. Let's break it down one by one:

- **ngModel** is a directive available from Angular. Its job is to connect application data to the input value of a form control like an <input> element or <textarea> in this case. Here we are stating that bind the **userInput** field with this textarea.
- The [(…)] syntax is used for two-way data binding. When the textarea is first rendered it will contain the initial value of the **userInput** field ("Enter HTML here"). As the user types in some text Angular will take the user's input and update the **userInput** field. That is how two-way data binding works.

We could compile and test this component now. But it's not terribly useful. We should try to do something with the text that the user has entered. We will now show a preview of the text that user has entered.

__9. Add the lines shown in bold to the template.

```
<textarea [(ngModel)]="userInput" rows="5" cols="40">
</textarea>
<h4>Preview</h4>
<div [innerHTML]="userInput"></div>
```

Here we added a <div> tag with the attribute [innerHTML]="userInput". This is an example of one-way data binding. The […] syntax is used for one-way data binding. This connects the **userInput** field of the component to the **innerHTML** property of the <div> tag.

A key thing to note here is that HTML does not define any attribute called innerHTML. It is, in fact, a property of a DOM element that can only be set using the DOM API. For example:

```
someElement.innerHTML = "Text"
```

Angular lets you bind to these properties using the attribute like syntax.

Most DOM properties also have corresponding attributes in the HTML specification. For example the **src** attribute of an <img> tag. But keep in mind, Angular is really binding to the DOM property and not just setting the HTML attribute value. Example:

```
<img [src]="someField"></img>
```

__10. Save changes.

Part 3 - Use the HTML Editor Component

__1. Open **binding-test/src/app/app.component.html**

__2. Change the template.

```
<app-html-editor></app-html-editor>
```

__3. Save changes.

Part 4 - Import FormsModule

The NgModel directive is provided by the FormsModule module. This is not available by default. So we must import FormsModule.

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

__2. Import the FormsModule name.

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

__3. Add FormsModule to the imports list of the @NgModule decorator.

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

__4. Save changes. Now any directive, component, pipe etc. exported by FormsModule will be available throughout our application.

Part 5 - Test

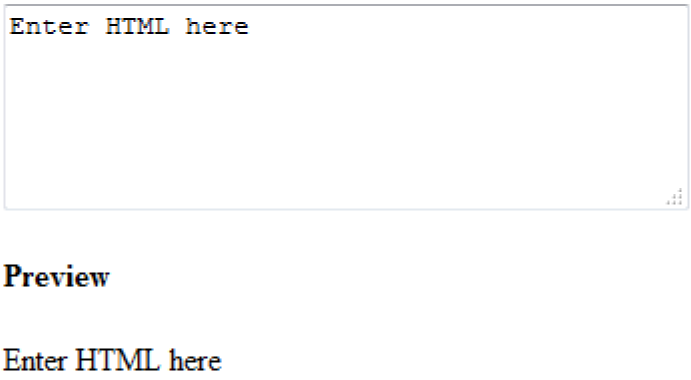
__1. Open a command prompt window.

__2. Change directory to the **C:\LabWork\binding-test** folder.

__3. Execute the following command to compile TypeScript and run the server:

```
npm start
```

__4. Open Chrome and navigate to: **http://localhost:4200/**



Enter HTML here

Preview

Enter HTML here

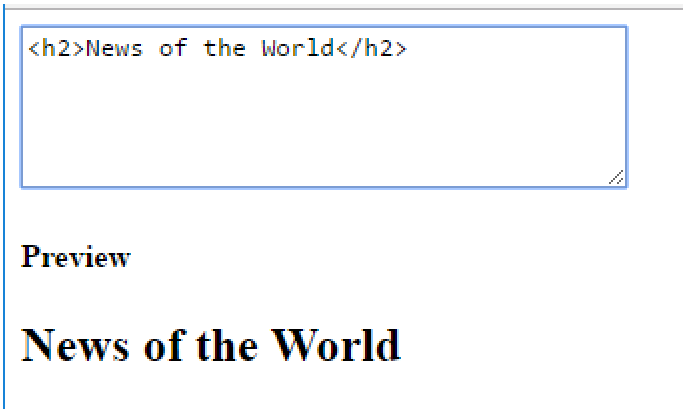
Verify that the textarea shows the initial value of the **userInput** field of the component class:

```
userInput = "Enter HTML here"
```

__5. Enter this HTML text into the editor:

```
<h2>News of the World</h2>
```

__6. Make sure that the preview section shows the rendered HTML. Here application data is getting updated from the DOM.



```
<h2>News of the World</h2>
```

Part 6 - Use Interpolation

Interpolation in HTML template is another example of one-way data binding. Except instead of the […] syntax it uses the {{…}} syntax. It is useful for setting body text of an element.

We will now display the raw text entered by the user without rendering it as HTML.

__1. In **html-editor.component.html** file add these lines in boldface to the template.

```
<textarea [(ngModel)]="userInput" rows="5" cols="40"></textarea><br/>

<h4>Preview</h4>
<div [innerHTML]="userInput"></div>

<h4>Raw Text</h4>
<pre>{{userInput}}</pre>
```

This will set the body of the <pre> tag with the **userInput** field of the component class. That constitutes a one-way binding.

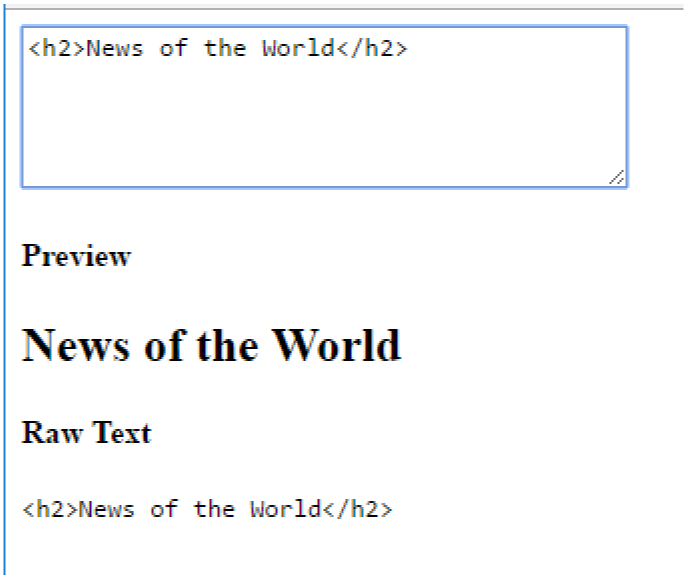
__2. Save changes.

__3. Check there are no issues recompiling the code. The web browser should refresh automatically.

__4. Enter this HTML text into the editor:

```
<h2>News of the World</h2>
```

__5. Make sure that the raw text is showing correctly.



```
<h2>News of the World</h2>
```

Preview

News of the World

Raw Text

```
<h2>News of the World</h2>
```

Part 7 - Peeking Behind the Mystery

How does the [(ngModel)]="userInput" actually achieve two way data binding? Fundamentally Angular provides two separate syntax:

- [someComponentField]="initialValue" : The [ ] syntax sends data to a component, or directive from the component using it.
- (someEvent)="handlerFunction(payLoad)" : The ( ) syntax simply attaches a handler for an event fired by a component or directive. This is also the way a component or directive emits data to the component using it. For example, a component can handle a button click like this: <button (click)="handleClick()">OK</button>

The two way data binding syntax [(ngModel)]="userInput" is simply a shortcut that combines the above two separate syntaxes. We will now use these two syntaxes separately to gain a better understanding of what is really going on.

__1. Open **html-editor.component.html**

__2. Change the <textarea> tag in the template like this.

```
<textarea [ngModel]="userInput" (ngModelChange)="updateHTML($event)"
    rows="5" cols="40">
</textarea>
```

This works like as follows:

- [ngModel]="userInput" will set the initial value of the <textarea> to the current value of userInput. Which is "Enter HTML here".
- ngModelChange is an event fired by the NgModel directive any time user enters input and changes the input's value. We are attaching an event handler for that event. The updated value of the input is made available via the **$event** local variable.

__3. Save changes.

__4. Open **html-editor.component.ts**

__5. Write the updateHTML() method as follows inside the class.

```
updateHTML(newValue) {
  console.log("HTML updated: %s", newValue)

  this.userInput = newValue
}
```

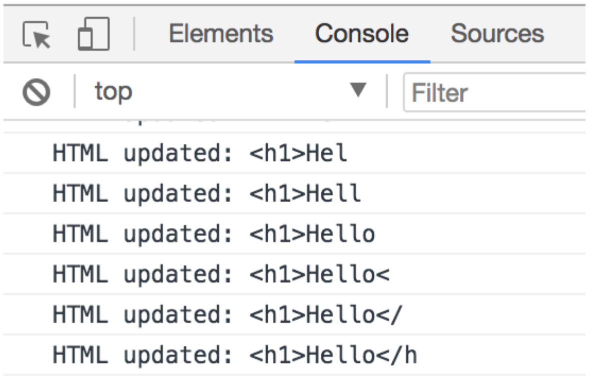
__6. Save changes.

Part 8 - Test Changes

__1. Verify that the application works same as before. We didn't change any functionality of the application.

__2. Press F12 to show the Developers Tools and select the **Console** tab.

__3. Start entering a text like **<h1>Hello</h1>** and verify that the log from updateHTML() is visible.



```
HTML updated: <h1>Hel
HTML updated: <h1>Hell
HTML updated: <h1>Hello
HTML updated: <h1>Hello<
HTML updated: <h1>Hello</
HTML updated: <h1>Hello</h
```

__4. Press F12 again to close the Console.

Part 9 - Clean Up

__1. In the command prompt, hit '**<CTRL>-C**' to shut down the server.

__2. Close all open files.

Part 10 - Review

In this lab, we got a sense of how two-way and one-way data binding works. We achieved two-way data binding using the syntax [(ngModel)]="userInput". This actually bound the ngModel directive with the userInput field.

One-way data binding was achieved using [innerHTML]="userInput". This bound the innerHTML DOM property to the userInput field.

Interpolation done using {{userInput}} sets the body text of an element. This is also an example of one-way binding. But as you may know, in DOM API setting the body text of an element is not the same as setting the innerHTML property. The latter renders the text as HTML.

Lab 6 - One Way Data Binding in a Custom Component

With one way data binding a parent component can pass data to a child from. We have already seen how to pass data to a component using the @Input decorator. The process is the same here. Except now the source of data will be one of the parent component's fields.

This lab extends the project in the **C:\LabWork\subscription** directory that was used in the "Create the Subscription Form Component" lab.

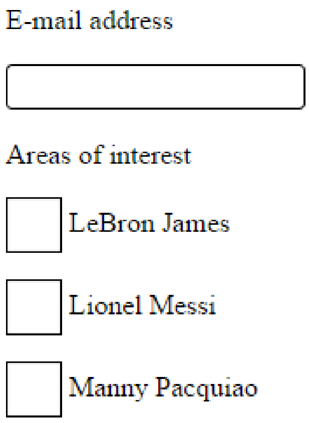
Part 1 - Business Requirement

Right now our SubscriptionForm component has a list of hard coded areas of interests in the template.

```
<p><fancy-checkbox label="Volleyball"></fancy-checkbox></p>
<p><fancy-checkbox label="Football"></fancy-checkbox></p>
<p><fancy-checkbox label="Tennis"></fancy-checkbox></p>
```

Hard coding the areas of interest like this makes the component less reusable. A truly reusable e-mail subscription form will accept a list of interests as input. We will make that change now.

In this lab we will make the SubscriptionForm component reusable. We will then use it to create a form which will let users choose from a list of top athletes to get news for.



E-mail address

Areas of interest

□ LeBron James

□ Lionel Messi

□ Manny Pacquiao

Part 2 - Make SubscriptionForm Reusable

The component will now accept a list of areas of interest as input.

__1. Open **src\app\subscription-form\subscription-form.component.ts** from the project at **C:\LabWork\subscription**.

__2. Import the @Input decorator.

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

__3. Add a field to the class.

```
export class SubscriptionFormComponent implements OnInit {
    @Input() interests:[string]

...
}
```

__4. Save changes.

We will now change the HTML template to show a checkbox for each area of interest. We will use a structural directive called ngFor for this. We haven't covered directives yet. But just think of it as a way to iterate over a collection and render some HTML for each element.

__5. Open **subscription-form.component.html**

__6. Delete all the lines that show checkboxes.

```
    <p><fancy-checkbox label="Volleyball"></fancy-checkbox></p>
    <p><fancy-checkbox label="Football"></fancy-checkbox></p>
    <p><fancy-checkbox label="Tennis"></fancy-checkbox></p>
```

__7. In their place add this line.

```
<p *ngFor="let interest of interests">
  <fancy-checkbox label="{{interest}}"></fancy-checkbox>
</p>
```

Basically, the code is iterating over the interests collection variable and showing a checkbox for each item.

__8. Save changes. Our component is now more reusable.

Part 3 - Create the Player Subscription Form

We will now create a component that will let users subscribe to news about athletes.

__1. From the **C:\LabWork\subscription** folder run this command.

```
ng generate component PlayerSubscribe
```

If you do not have Angular CLI installed globally then run:

npm run ng generate component PlayerSubscribe

__2. In **src\app\player-subscribe\player-subscribe.component.ts** add a field to the PlayerSubscribeComponent class.

```
export class PlayerSubscribeComponent implements OnInit {
   players = ["LeBron James", "Lionel Messi", "Manny Pacquiao"]
...
```

Note: Here we did not explicitly state the type of the players variable. The TS compiler will infer the type to be [string] from the initializer value on the right hand side.

__3. Save changes.

__4. Open **player-subscribe.component.html**. Change the template for the component like this.

```
<app-subscription-form [interests]='players'></app-subscription-form>
```

The crucial part is [interests]='players'. This will bind the **players** variable of PlayerSubscribeComponent with the **interests** class variable of the child component SubscriptionForm.

__5. Save changes.

Part 4 - Use PlayerSubscribeComponent

__1. Open **subscription\src\app\app.component.html**

__2. Use the PlayerSubscribeComponent.

```
<app-player-subscribe></app-player-subscribe>
```

__3. Save changes.

Part 5 - Test Changes

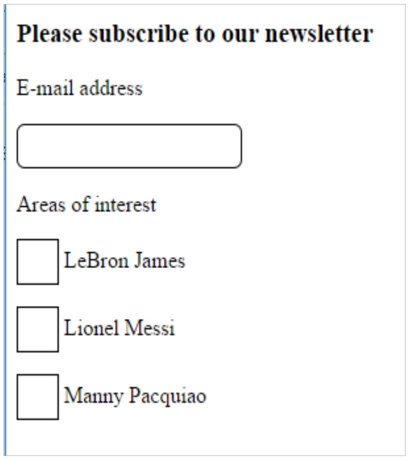
__1. Open a command prompt and navigate to the root folder of the project:

```
cd C:\LabWork\subscription
```

__2. Run the following command to compile TypeScript and start the server:

```
npm start
```

__3. Open a browser to **http://localhost:4200/** and you should see the page below.



Please subscribe to our newsletter

E-mail address

Areas of interest

□ LeBron James

□ Lionel Messi

□ Manny Pacquiao

Note: Binding is more than simply setting a variable's value. Angular will constantly monitor the value of the players variable. If it changes in the future then binding will cause the template to be re-rendered to show the updated state.

__4. In the command prompt, hit '**<CTRL>-C**' to shut down the server.

__5. Close all open files.

Part 6 - Review

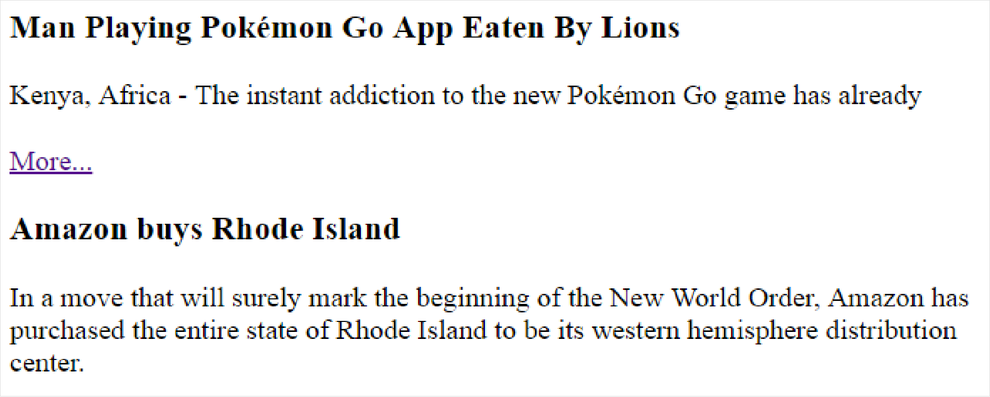
In this lab, we learned how to do implement one way data binding from a parent component to a child component.

Lab 7 - Using Basic Angular Directives

Directives have much narrower responsibilities compared to components. Directives manipulate DOM elements to do things like change color and font or show and hide the elements.

NgClass is a very popular directive that comes out of the box from Angular. It is used to dynamically assign class names to DOM elements. In this lab, we will learn how to use it.

In this lab we will build a page that lists a set of news items. Clicking the "More…" link for a news item expands it.



Man Playing Pokémon Go App Eaten By Lions

Kenya, Africa - The instant addiction to the new Pokémon Go game has already

More...

Amazon buys Rhode Island

In a move that will surely mark the beginning of the New World Order, Amazon has purchased the entire state of Rhode Island to be its western hemisphere distribution center.

We will use the NgClass directive to show/hide sections of the page.

Part 1 - Get Started

__1. Open a command prompt window.

__2. Go to the **C:\LabWork** folder.

__3. Run this command to create a new project called **directive-test**.

```
ng new directive-test --defaults
```

No Internet Access?

Extract C:\LabFiles\project-seed.zip into the C:\LabWork directory.

This should create a C:\LabWork\project-seed folder.

Rename the C:\LabWork\project-seed to C:\LabWork\directive-test.

Part 2 - Create the Component

__1. Go into the **directive-test** folder.

__2. Run this command to create the NewsList component.

```
ng generate component NewsList
```

__3. To save time the starter code for the component's template is given to you. Open **C:\LabFiles\basicDirectives\news-list.component.html**. Copy it contents.

__4. Open and then paste it inside **src/app/news-list/news-list.component.html** replacing the existing code.

Study the template code. At this point it's just a static HTML with a bunch of fake news items.

__5. Save the file.

__6. Edit the **app.component.html** file to match the following:

```
<app-news-list></app-news-list>
```

__7. Save the file.

Part 3 - Run the Application

__1. Open a command prompt and navigate to the root folder of the project:

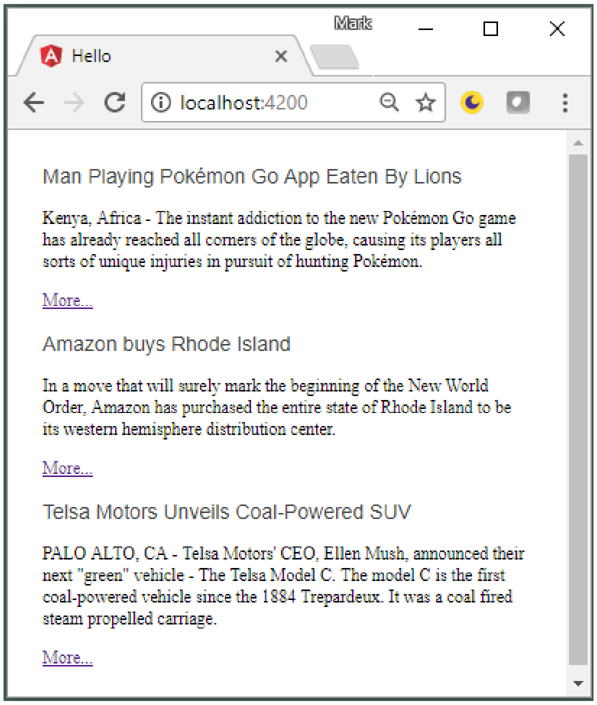
```
C:\LabWork\directive-test
```

__2. Run the following command to compile TypeScript and start the server:

```
npm start
```

__3. Open a browser at **http://localhost:4200/**

You should see three news items. All fake (courtesy www.thespoof.com) and all expanded by default:



Man Playing Pokémon Go App Eaten By Lions

Kenya, Africa - The instant addiction to the new Pokémon Go game has already reached all corners of the globe, causing its players all sorts of unique injuries in pursuit of hunting Pokémon.

More...

Amazon buys Rhode Island

In a move that will surely mark the beginning of the New World Order, Amazon has purchased the entire state of Rhode Island to be its western hemisphere distribution center.

More...

Telsa Motors Unveils Coal-Powered SUV

PALO ALTO, CA - Telsa Motors' CEO, Ellen Mush, announced their next "green" vehicle - The Telsa Model C. The model C is the first coal-powered vehicle since the 1884 Trepardeux. It was a coal fired steam propelled carriage.

More...

Part 4 - Add Style Class

We will now add a style class to the component that will be applied to any news item that is collapsed. Any expanded news item will not have this class applied.

__1. Open **directive-test/src/app/news-list/news-list.component.css**

__2. Add the CSS class as shown below.

```
.collapsed {
    display: none;
}
```

__3. Save changes.

Part 5 - Manage Expansion State

We will allow only one news item to be expanded at a time. We will now modify the component class logic to manage this state.

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

__2. In the class, add the code shown in bold.

```
export class NewsListComponent implements OnInit {

  selectedNewsId:number

    expandNews(id:number) {
        this.selectedNewsId = id

        return false
    }

  constructor() { }

  ngOnInit() {
  }

}
```

__3. Save changes.

Part 6 - Update the Template

We will now update the template to do these things:

- Apply the "collapsed" to any news item that is not currently selected.
- When the "More…" link is clicked set the corresponding news item as selected.

__1. Open **news-list.component.html**

__2. For the first news item change the template as shown in bold below.

```
<div>
    <h3>Man Playing Pokémon Go App Eaten By Lions</h3>
    <p [ngClass]="{collapsed: selectedNewsId != 0}">
        Kenya, Africa - The instant addiction ...
    </p>
    <a href (click)="expandNews(0)">More...</a>
</div>
```

This will conditionally apply the "collapsed" class if the selected news ID is not 0.

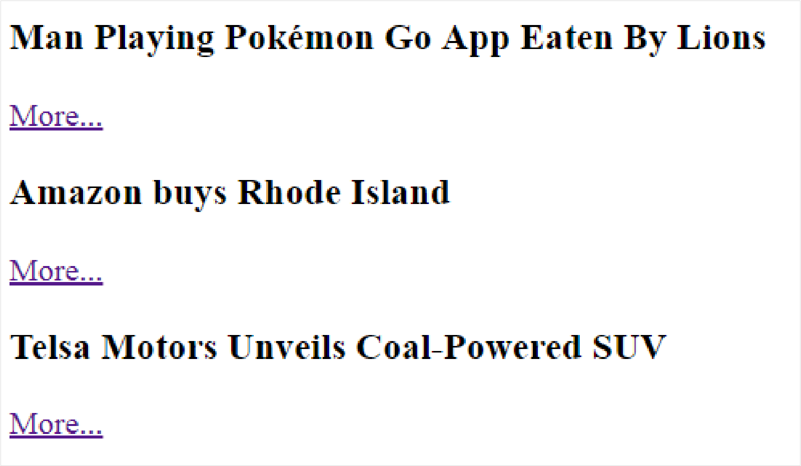
__3. Similarly, modify the template for the other news items.

```
<div>
        <h3>Amazon buys Rhode Island</h3>
        <p [ngClass]="{collapsed: selectedNewsId != 1}">
        In a move that will...
        </p>
        <a href (click)="expandNews(1)">More...</a>
</div>

<div>
        <h3>Telsa Motors Unveils Coal-Powered SUV</h3>
        <p [ngClass]="{collapsed: selectedNewsId != 2}">
        PALO ALTO, CA - Telsa Motors' CEO...
        </p>
        <a href (click)="expandNews(2)">More...</a>
</div>
```

__4. Save changes.

Note: When attaching an event handler for a hyperlink (<a> tag), make sure the method returns false. Otherwise the browser will navigate the link and reload the page. Our expandNews() method does this.

__5. Check there are no issues recompiling the code and the web browser should refresh automatically.



Man Playing Pokémon Go App Eaten By Lions

More...

Amazon buys Rhode Island

More...

Telsa Motors Unveils Coal-Powered SUV

More...

__6. Make sure all news items are collapsed by default. You may want to reduce the width of your browser to make sure the news items don't display on one line since this will make it harder to tell if they expand.

__7. Click on the "More…" link for a news item. It should expand. If you click the "More…" link for another news item, it should expand, and the previous one should collapse.

Part 7 - Hide the More Link if Expanded

If a news item is currently expanded, we should not show the "More…" link for it. This makes sense because if a news item is already expanded, clicking the link doesn't do anything anyway. We need to hide it by setting the element's **hidden** DOM property.

__1. For the first news item modify the template as shown in bold below.

```
<a href (click)="expandNews(0)"
  [hidden]="selectedNewsId == 0">More...</a>
```

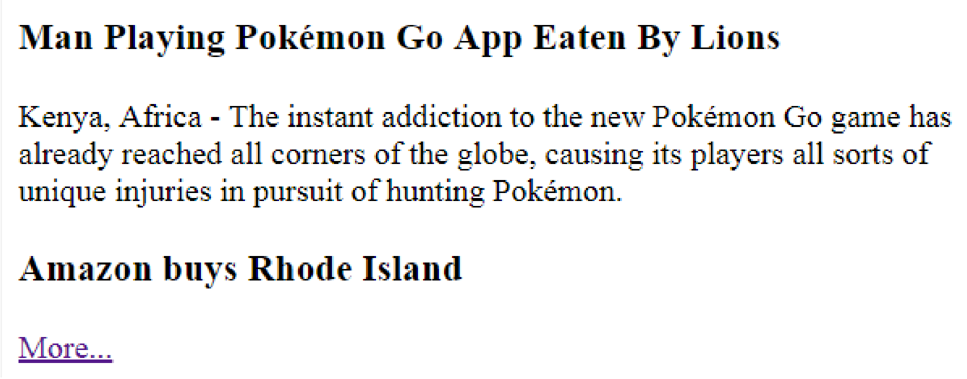
We are using one-way binding to update the hidden property.

__2. Similarly, change the template for the "More…" link for the other news items. Make sure to adjust the number used in the expression.

__3. Save changes.

__4. Check there are no issues recompiling the code and the web browser should refresh automatically.

__5. Verify that now when you expand a news item the corresponding "More…" link gets hidden.



Man Playing Pokémon Go App Eaten By Lions

Kenya, Africa - The instant addiction to the new Pokémon Go game has already reached all corners of the globe, causing its players all sorts of unique injuries in pursuit of hunting Pokémon.

Amazon buys Rhode Island

More...

__6. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__7. Close all open files.

Part 8 - Review

The NgClass directive allows us to conditionally set CSS classes to any DOM element. We used that feature to apply the "collapsed" class to any news item that is not currently selected. You can also update any property defined for an element in the DOM API using one-way binding. We used that to set the "hidden" property of the "More…" link.

Lab 8 - Using Structural Directives

Structural directives can add or remove DOM elements to a page. By contrast, an attribute directive manipulates an existing DOM element that you have manually coded into a template.

In a previous lab, we have created a page that shows a list of fake news items. We had hard coded the news items in the template. In real life, the data will most likely come from some kind of a web service. We will now modify the template to render the news items from a data source. We have not learned how to call a web service yet. So we will define the data source as an array in our TypeScript code.

This lab builds upon the previous lab "**Basic Angular Directives**".

Part 1 - Define News Items Data

__1. Open **C:\LabFiles\structuralDirectives\newsItems.txt**. It contains the code snippet for an array containing various news items.

__2. Copy the code.

__3. Open **directive-test/src/app/news-list/news-list.component.ts** that you had worked on in a previous lab.

__4. Paste the code inside the class as shown below in bold.

```
export class NewsListComponent implements OnInit {
    selectedNewsId:number

    newsItems = [
    { …
```

__5. Make sure that you have copied the entire contents of the array including the square bracket that ends the array "]"

__6. Save the file.

Part 2 - Render the List Using NgFor

NgFor is a built-in structural directive that can iterate over a collection and generate some DOM elements for each item. We will now use the NgFor directive to render items from a newsItems array we just copied.

__1. Open the **news-list.component.html** file.

__2. Delete the contents of the template.

__3. Add the following code to the template.

```
<div>
    <div *ngFor="let news of newsItems; let newsId = index">
        <h3>{{news.title}}</h3>
        <p [ngClass]="{collapsed: selectedNewsId != newsId}">
        {{news.body}}
        </p>
        <a href (click)="expandNews(newsId)"
            [hidden]="selectedNewsId == newsId">More...</a>
    </div>
</div>
```

Note these aspects:

- We are iterating over the **newsItems** array. Each item is saved in a local variable called **news**.
- The NgFor directive also exports various variables, one of them being **index**. We are copying that value in a local variable called **newsId**.
- We use the * with ngFor. This is very common for structural directives. These directives use templates of their own. Using the * is a short cut that saves typing the whole template.

__4. Save changes.

__5. Open a command prompt and navigate to the root folder of the project:

```
C:\LabWork\directive-test
```

__6. Run the following command to compile TypeScript and start the server:

```
npm start
```

__7. Open a browser at **http://localhost:4200/**

__8. Make sure that you can expand and collapse news items.

__9. Feel free to add your own stories to the **newsItems** array and test it out.

Part 3 - Hide Elements Using NgIf

Currently, we are hiding the "More…" link by setting the hidden DOM property. That is fine. But in some cases, you may wish to completely remove an element from the DOM tree. For example, if you need to hide a large and complicated element, it may be more efficient to simply remove it from the tree. To do that we need to use NgIf.

Generally speaking if you need to hide a large and complex element use NgIf. Otherwise use the hidden property. Both work just fine.

We will now hide the "More…" link using NgIf.

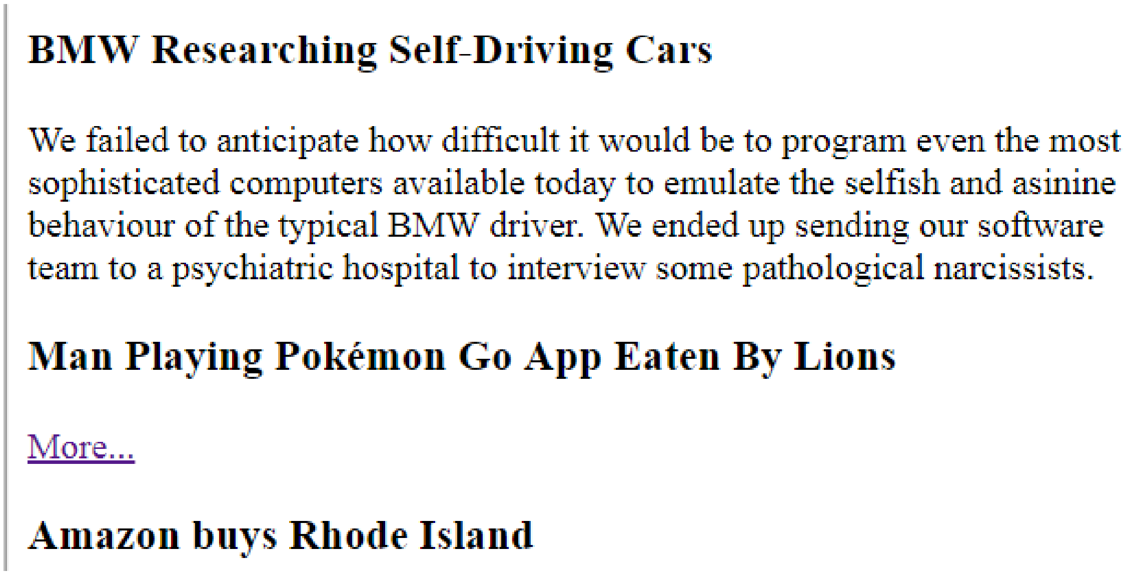
__1. In the **news-list.component.html** file, change the template for the "More…" link as shown in bold below. Note the expression changes slightly also.

```
<a href (click)="expandNews(newsId)"
  *ngIf="selectedNewsId != newsId">More...</a>
```

__2. Save changes.

__3. Check there are no issues recompiling the code and the web browser should refresh automatically.

__4. Verify that the page works same as before. Except now when we hide the "More…" link it gets completely removed from the DOM tree. You can verify that by inspecting the DOM tree using your browser's developer tools.



BMW Researching Self-Driving Cars

We failed to anticipate how difficult it would be to program even the most sophisticated computers available today to emulate the selfish and asinine behaviour of the typical BMW driver. We ended up sending our software team to a psychiatric hospital to interview some pathological narcissists.

Man Playing Pokémon Go App Eaten By Lions

More...

Amazon buys Rhode Island

__5. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__6. Close all open files.

Part 4 - Extra Credit

Right now, all news items show up as collapsed by default. What do you have to do to show the first news item as expanded when the application first loads?

Part 5 - Review

In this lab, we learned to use two structural directives – NgFor and NgIf.

Lab 9 - Custom Attribute Directive

In this lab, we will develop a directive, '**appCcLogo**', that shows the logo of a credit card company. The directive will take as input a credit card number. It will determine the type of the card vendor. It will then display the logo of the vendor. The directive will be used with an <img> tag like this.

```
<input type="text" [(ngModel)]="creditCardNumber"/>
<img appCcLogo [ccNumber]="creditCardNumber"/>
```

Part 1 - Get Started

__1. From the **C:\LabWork** folder run:

```
ng new custom-directive --defaults
```

__2. Switch to the **custom-directive** folder. Then run this to create the directive:

```
ng generate directive cc-logo
```

__3. Open **src\app\app.module.ts** and verify that the directive class got added to the declarations list.

```
@NgModule({
  declarations: [
    AppComponent,
    CcLogoDirective
  ],
...
})
```

__4. Add this import statement.

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

__5. Import FormsModule into the application module.

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

__6. Save changes.

__7. Copy these files from **C:\LabFiles\images** to the **src/assets** folder of the project.

- visa.png
- mastercard.png
- amex.png

Note: The build system will copy all files saved in the assets folder into the dist folder.

Part 2 - Develop the Directive

__1. Open **src/app/cc-logo.directive.ts**

__2. Import a few additional names.

```
import { Directive, HostBinding, Input } from '@angular/core';
```

__3. We need to add these member variables inside the class.

```
@HostBinding('src') imageSrc
@HostBinding('hidden') isHidden:boolean
@Input() ccNumber: string = ""
```

Explanation for these two variables are as follows:

- The @HostBinding("src") decorator binds the imageSrc member variable to the **src** attribute of an **<img>** tag.
- The @HostBinding("hidden") decorator binds the isHidden member variable to the **hidden** attribute of an **<img>** tag.
- The @Input directive lets a template supply a credit card number using the ccNumber attribute.

__4. Add a method that will determine the type of credit card. The logic is kept simple and may not satisfy all real life cases.

```
  getCCType() : string {
    if (this.ccNumber.startsWith("4")) {
      return "visa"
    } else if (this.ccNumber.startsWith("5")) {
      return "mastercard"
    } else if (this.ccNumber.startsWith("3")) {
      return "amex"
    }

    return undefined
  }
```

__5. Now we will add the ngOnChanges() lifecycle method. This will get called every time the bound variable ccNumber changes.

```
ngOnChanges() {
  const ccType = this.getCCType()

  this.isHidden = ccType == undefined
  this.imageSrc = `assets/${ccType}.png`
}
```

__6. Save the file.

Part 3 - Use the Custom Directive

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

__2. Add a member variable for the credit card number inside the class.

```
creditCardNumber:string = ""
```

__3. Save changes.

__4. Open **src/app/app-component.html**

__5. Delete all existing content. Then enter these lines.

```
<p>Please enter a credit card number:</p>
<p>
  <input type="text" [(ngModel)]="creditCardNumber"/>
  <img appCcLogo [ccNumber]="creditCardNumber"/>
</p>
```

Now the input field is bound to the creditCardNumber variable which in turn is bound to the ccNumber variable of the directive.

__6. Save changes.

__7. Open **app-component.css**

__8. Enter this style.

```
img {
    width: 40px;
    vertical-align: middle;
}
```

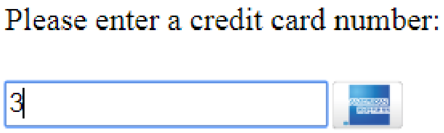
__9. Save changes.

Part 4 - Test

__1. From the command line run:

```
ng serve --open
```

__2. A browser will open, start typing a number starting with 3, 4 or 5. The logo should change.



Please enter a credit card number:

3

__3. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__4. Close all open files.

Part 5 - Review

In this lab, we learned how to write a custom directive.

Lab 10 - Template Driven Form

In this lab, we will explore basic form development.

Most common tasks with form development are:

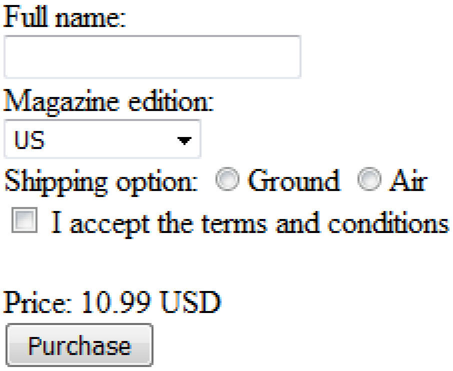
1. Pre-populating form controls with data. For example, populate a text box and select an item in a list box.
2. Obtain user's input.
3. Validate user's input.
4. Post user's input to the server.

Angular provides two different ways of doing this:

- Template driven – The form is entirely defined in HTML template. This is very similar to the way things were done in AngularJS 1.x.
- Model driven – Several aspects of the form are defined in component code. This makes your form unit testable.

This lab focuses on the template driven approach.

We will build a form that incorporates all the common HTML input controls.



Full name:

Magazine edition: US

Shipping option: ○ Ground ○ Air

□ I accept the terms and conditions

Price: 10.99 USD

Part 1 - Create a New Project

__1. Open a command prompt window.

__2. Go to the **C:\LabWork** folder.

__3. Run this command to create a new project called **form-test**.

```
ng new form-test --defaults
```

No Internet Access?

Extract C:\LabFiles\project-seed.zip to the C:\LabWork Directory.

This should create the C:\LabWork\project-seed folder.

Rename the C:\LabWork\project-seed to C:\LabWork\form-test.

Part 2 - Add NgModel Directive Support

Data binding for the form input elements heavily depends on the NgModel directive. This is not available unless our application module imports the FormsModule. We will do that now.

__1. Open **form-test\src\app\app.module.ts**

__2. Add this import statement.

```
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 3 - Create the Component

__1. From the **C:\LabWork\form-test** folder run this command.

```
ng generate component magazine
```

If you do not have Angular CLI installed globally then run:

```
npm run ng generate component magazine
```

Part 4 - Design the Model

It is generally a good idea to focus first on the data structure of the form first. The form design will be much easier after that.

__1. Open **form-test/src/app/magazine/magazine.component.ts**

__2. In the MagazineComponent class, add these fields that will act as the model for our form. Cross check the model with the screenshot of the form shown earlier.

```
export class MagazineComponent implements OnInit {
  fullName = ""
  editions = [
    {editionCode: 1, editionName: "US", price: "10.99 USD"},
    {editionCode: 2, editionName: "Canada", price: "14.99 CAD"},
    {editionCode: 3, editionName: "International", price: "23.99 USD"}
  ]
  selectedEdition = this.editions[0] //Choose US by default
  selectedShipping = ""
  acceptPolicy = false
...
```

__3. Save changes.

We will now start to design the form in the component's template.

Part 5 - Add the Text Box

We will now add the text for the "Full name" field. Text boxes are the easiest to work with, and we have already seen it in use in previous labs.

__1. Open **magazine.component.html**. Delete the current contents.

__2. Add the following HTML template to the component replacing the existing code.

```
Full name:<br/>
<input type="text" [(ngModel)]="fullName"/><br/>
```

Basically, we are setting up a two-way data binding between the fullName field and the text box.

__3. Save changes.

Part 6 - Add the Drop Down List

We will now add the dropdown (select element) for the magazine edition. Lists often show dynamic data. In this case, we need to show items from the "editions" field of the component.

__1. In the HTML template, below the text box add the following.

```
Magazine edition:<br/>
<select [(ngModel)]="selectedEdition">
  <option *ngFor="let e of editions"
    [ngValue]="e">{{e.editionName}}</option>
</select><br/>
```

First look at the <option> tag. We are using the ngFor directive to loop through the "editions" collection. We are defining a local variable called "e" for each item in the collection. The value for each option is the corresponding edition object. This is set using ngValue.

The option tags will show the editionName property for the corresponding edition. This is done using the {{e.editionName}} expression.

Lastly, look at the <select> tag. It has a two-way binding with the selectedEdition field of the component class. When user selects an option the corresponding edition object will be assigned to the selectedEdition field.

__2. Save changes.

Part 7 - Add the Shipping Radio Buttons

__1. Below the </select> tag add:

```
Shipping option:
<input type="radio" name="selectedShipping"
    [(ngModel)]="selectedShipping" value="GROUND"/>Ground
<input type="radio" name="selectedShipping"
    [(ngModel)]="selectedShipping" value="AIR"/>Air
<br/>
```

The only thing worth noting here is that we use the name attribute for the radio button elements to group them together. This is standard HTML but just worth paying attention.

__2. Save changes.

Part 8 - Add the Check Box

__1. Below the radio buttons add:

```
<label><input type="checkbox" [(ngModel)]="acceptPolicy"/>
  I accept the terms and conditions</label><br/>
```

We are setting up a two-way data binding between this checkbox and the acceptPolicy field which is of boolean type.

__2. Save changes.

Part 9 - Display Price of the Selected Edition

We save the selected edition object in the selectedEdition field. This makes it trivial to show the price.

__1. Below the checkbox add:

```
<input type="checkbox" [(ngModel)]="acceptPolicy"/>
    I accept the terms and conditions<br/>
<br/>
Price: {{selectedEdition.price}}
<br/>
```

__2. Save changes.

Part 10 - Add the Submit Button

__1. Below the price display add:

```
Price: {{selectedEdition.price}}
<br/>
<button (click)="submitForm()">Purchase</button>
```

__2. Save changes.

Part 11 - Implement the Submission Logic

In real life, you will likely make an Ajax request to submit the form. Here we will simply display the JSON for the request.

__1. In the Magazine class, add the submitForm() method like this.

```
export class MagazineComponent implements OnInit {
...

   submitForm() {
       let requestData = {
           customerName: this.fullName,
           productCode: this.selectedEdition.editionCode,
           acceptPolicy: this.acceptPolicy,
           shipMode: this.selectedShipping
       }

       alert(JSON.stringify(requestData))
   }
...
```

__2. Save changes.

Part 12 - Use the Component

Now we will add the component to a page and try it out.

__1. Open **form-test/src/app/app.component.html**

__2. Change the template like this.

```
<app-magazine></app-magazine>
```

__3. Save changes.

Part 13 - Test Changes

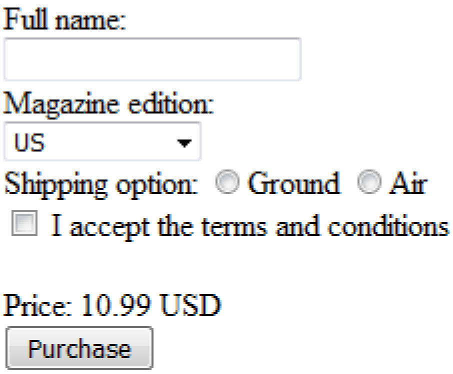
__1. Open a command prompt and navigate to the root folder of the project:

```
C:\LabWork\form-test
```

__2. Run the following command to compile TypeScript and start the server:

```
npm start
```

__3. Open a web browser to **http://localhost:4200/**



Full name:

Magazine edition: US

Shipping option: ○ Ground ○ Air

□ I accept the terms and conditions

Price: 10.99 USD

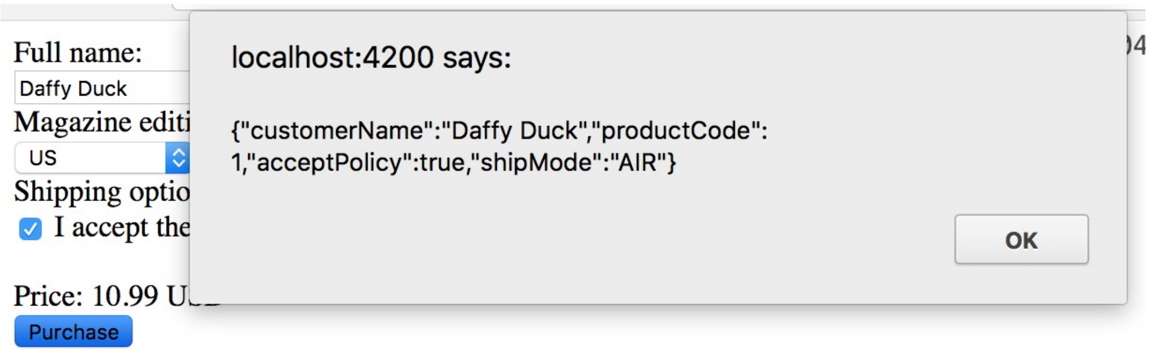
__4. Enter a name in the full name field.

__5. Change the magazine edition. Verify that the price display changes correctly.

__6. Select shipping.

__7. Check the box to accept terms.

__8. Click the **Purchase** button.



localhost:4200 says:

{"customerName":"Daffy Duck","productCode":
1,"acceptPolicy":true,"shipMode":"AIR"}

__9. Verify that the JSON has the correct user input. If the browser has an option to prevent further dialogs, make sure to NOT select the option. Click OK.

__10. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__11. Close all open files.

Part 14 - Review

In this lab, we implemented a simple form using Angular. This was basically an exercise in data binding for different types of controls.

Lab 11 - Validation of a Template Driven Form

Validation involves two key tasks:

- Showing visual feedback to the user if they have entered an invalid input.
- Disabling form submit unless all the form fields are valid.

In this lab we'll learn how to do this for a template driven form.

The lab builds upon the prior "**Template Driven Forms**" lab.

Part 1 - States of an Input Field

An input field can have these states:

- Valid or Invalid – Indicates if the value of the input is valid or invalid.
- Touched or untouched – If the input field ever received keyboard focus, then the status will be touched. Otherwise, it will be untouched.
- Pristine or dirty – If the user started to enter some data then the field will be marked as dirty. Otherwise, the field will be pristine.

Angular automatically assigns these CSS classes to an input based on its status:

- ng-valid or ng-invalid
- ng-touched or ng-untouched
- ng-pristine or ng-dirty

Let us observe the status of an input in action.

__1. Edit the following file (from the previous lab):

`form-test\src\app\magazine\magazine.component.html`

__2. Make the fullName input field mandatory by adding the 'required' attribute as shown in bold face below:

```
<input type="text" required
       [(ngModel)]="fullName" /><br/>
```

__3. Assign a template variable for the fullName input field as shown in bold below.

```
<input type="text" required
       [(ngModel)]="fullName" #theElement /><br/>
```

The template local variable "**theElement**" will point to the DOM element for the <input> tag. Using this variable, we can access all the CSS classes assigned by Angular.

__4. Below that line add this line to view all the CSS classes for the input field:

```
<input type="text" required
       [(ngModel)]="fullName" #theElement/><br/>
    Class names: {{theElement.className}}<br/>
    Magazine edition:<br/>
```

__5. Save changes.

__6. Open a command prompt and navigate to the root folder of the project:

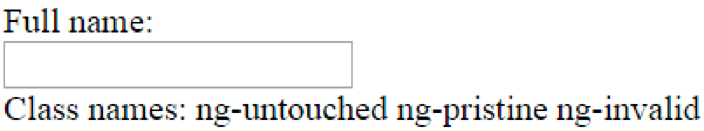
```
C:\LabWork\form-test
```

__7. Run the following command to compile TypeScript and run the server:

```
npm start
```

__8. Open a browser to **http://localhost:4200/**

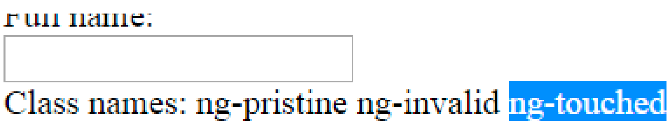
__9. Initially the class names will be like this. Depending on the browser, you may see the field with some error highlighting.



Full name:

Class names: ng-untouched ng-pristine ng-invalid

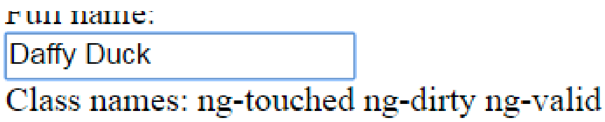
__10. Click into the text box and then click outside immediately. Do not type anything. Now the field will be marked as touched.



Full name:

Class names: ng-pristine ng-invalid ng-touched

__11. Click into the text box and start typing. Now the field will be both dirty and valid.

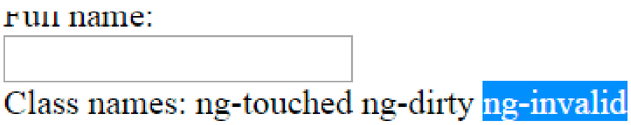


Full name:

Daffy Duck

Class names: ng-touched ng-dirty ng-valid

__12. Delete all the text. The field will still be dirty but it will be marked as invalid again.



Full name:

Class names: ng-touched ng-dirty ng-invalid

These states form the foundation of form validation in Angular. Everything we do here will depend on this concept.

Part 2 - Style Invalid Input Fields

When a form is first displayed many of the fields will have no value and may be invalid. You should not display these fields as invalid until the user actually starts to work on these fields. Which means we should display a field as invalid if both **ng-touched** and **ng-invalid** classes are applied to it.

__1. Open **magazine.component.css**

__2. Add this style rule.

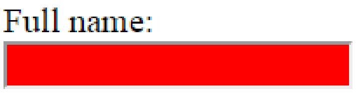
```
input.ng-touched.ng-invalid {
  background: red
}
```

__3. Save changes.

__4. Check there are no issues recompiling the code and the web browser should refresh automatically.

__5. Make sure that the full name field is shown using the usual white background.

__6. Click inside the text box and immediately click outside. Now the background should change to red.



Full name:

Part 3 - Model State

Just like Angular assigns certain CSS classes to the input fields, ngModel directive keeps track of the same states for the associated input field. For example, if an input field is invalid then the **valid** property of the associated ngModel directive will be false. For more advanced styling you may need to access the model's state. For example, if we are to show an error message for an input we will need to access the model's state. But first let's observe the model's state just like we did for the CSS classes. This will take a little bit of extra ground work.

__1. Open **magazine.component.html**

__2. Define a local variable for the ngModel directive for the full name text box as shown in bold face below.

```
<input type="text" required [(ngModel)]="fullName"
     #theElement #theModel="ngModel" /><br/>
```

This is the technique used to assign a directive object instance to a local variable within the template. Directives often expose useful properties and methods. We can access them using that variable. In this case, the **theModel** local variable will be of type **NgModel** class which is a directive. You can read about all the exposed properties and methods of the class here: https://angular.io/api/forms/NgModel.

You can also access the directive variable from within the component class. But that is outside the scope of this lab.

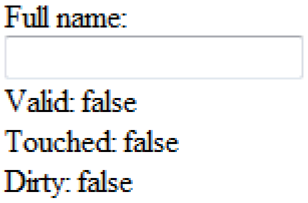
__3. Below that line, show the various states of the model like this, replacing display of the classes

```
    #theElement #theModel="ngModel"/><br/>
    Valid: {{theModel.valid}}<br/>
    Touched: {{theModel.touched}}<br/>
    Dirty: {{theModel.dirty}}<br/>
    Magazine edition:<br/>
```

__4. Save changes.

__5. Check there are no issues recompiling the code and the web browser should refresh automatically.

__6. You should see the initial state of the ngModel directive.



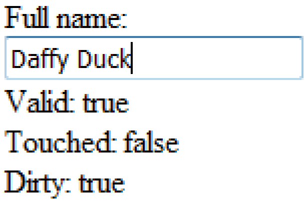
Full name:

Valid: false

Touched: false

Dirty: false

__7. Try playing with the text box. These states should change. The 'Touched' property will only change if you click outside the box after it has had focus.



Full name:

Daffy Duck

Valid: true

Touched: false

Dirty: true

You can also use the converse properties invalid, untouched and pristine.

Part 4 - Show Error Message

Now that we know how to access the status of a model associated with an input field, we can do all sorts of fancy error handling. For now we will display an error message for the full name text box.

__1. Add a <span> tag as shown in bold face below that displays an error message.

```
Full name:<br/>
<span style="color: red"
  [hidden]="theModel.valid || theModel.untouched">
Please enter your name<br/></span>
<input name="fullName" type="text" required [(ngModel)]="fullName"
        #theElement #theModel="ngModel"/><br/>
```

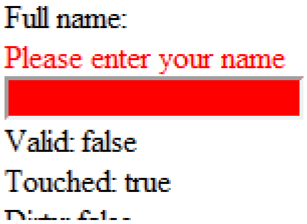
This basically uses one way binding to set the **hidden** property of the DOM element for the error message span.

__2. Save changes.

__3. Check there are no issues recompiling the code and the web browser should refresh automatically.

__4. Make sure that the error message is not displayed by default.

__5. Click the full name text box and immediately click outside. You should see the error message.



Full name:

Please enter your name

Valid: false

Touched: true

Dirty: false

__6. Start typing in the text box and the error message should go away.

Part 5 - Form Level Validation

So far we have done validation of individual input fields. To determine if all input fields are valid we need to do form level validation. A form is valid when all it's enclosed input fields are valid.

Just like ngModel, the ngForm directive also tracks the status of a form. The ngForm directive is associated with the <form> selector. We didn't have any need to use the <form> element so far. Let's add it now.

__1. Wrap the entire template HTML of MagazineComponent in a <form> tag.

```
<form>

...

</form>
```

A downside of wrapping input elements in a form is that Angular requires every element using ngModel directive to have a name.

__2. For the <input> tag assign a name.

```
<input name="fullName" type="text" .../>
```

__3. For the <select> element assign a name.

```
<select name="selectedEdition" …
```

__4. For the checkbox assign a name.

```
<input name="acceptPolicy" type="checkbox" ...
```

We need to declare a variable for the NgForm directive so that we can access its status.

__5. Modify the beginning of the <form> tag like this (see changes in **bold**):

```
<form #theForm="ngForm">
Full name:<br/>
```

Good to Know

The NgForm directive's selector is the "form" tag name. As a result, an instance of that directive is automatically attached to the <form> element. This differs from the NgModel directive whose selector is the ngModel attribute. We must use the ngModel attribute for an input element for an instance of that directive to be attached to the element.

__6. Disable the submit button if the form is invalid or if the policy has not been accepted.

```
<button [disabled]="theForm.invalid || !acceptPolicy"
    (click)="submitForm()">Purchase</button>
```

__7. Save changes.

__8. Check there are no issues recompiling the code and the web browser should refresh automatically.

__9. Make sure that the **Purchase** button is disabled by default.

__10. Start typing into the full name text box and check the box. The **Purchase** button should become enabled.

__11. Test the page.

__12. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__13. Close all open files.

Part 6 - Review

In this lab we learned how to validate user input. You should now be able to carry out these tasks:

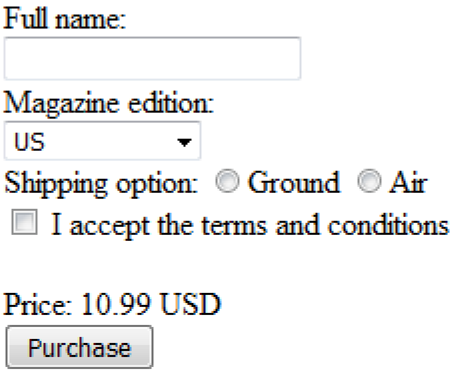
- Style invalid input fields.
- Show error message for individual input fields.
- Do form level validation.

Lab 12 - Reactive Forms

There are two ways to work with forms in Angular, template driven forms and reactive forms. Template driven forms rely on data binding to get or set form input values. Reactive forms explicitly call API methods to do the same. This makes reactive forms easier to unit test.

In this lab, we will explore form development using the reactive approach.

We will build a form that incorporates all the common HTML input controls. This will be the same form used in the template driven approach so you can see the similarities and differences between the two approaches. You will also add the same validation using the reactive approach.

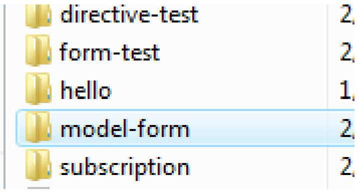

Full name:
Magazine edition:
US
Shipping option: ○ Ground ○ Air
□ I accept the terms and conditions
Price: 10.99 USD
Purchase

This lab builds on the "**Validation of a Template Driven Form**" Lab.

Part 1 - Copy Existing Project

__1. In the '**C:\LabWork**' folder, make a copy of the **form-test** folder.

__2. Rename the copied folder to **model-form**



Part 2 - Import ReactiveFormsModule Elements

The reactive form approach is defined by classes in the 'ReactiveFormsModule' instead of the 'FormsModule' of the template driven style. There are also more classes imported into the component code that are then used to define reactive form objects.

In this section you will alter the existing project to use the required reactive form modules.

__1. In the **model-form/src/app** folder, open the existing '**app.module.ts**' file in a text editor.

__2. Modify the two places the '**FormsModule**' is referenced into '**ReactiveFormsModule**' as shown below.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ReactiveFormsModule } from '@angular/forms';
import { Magazine } from './magazine/magazine.component';
...

  imports:      [ BrowserModule, ReactiveFormsModule ],
```

__3. Save and close the file after checking the changes above.

Part 3 - Define Form Elements in Component

We will now define the form and its input elements using the reactive forms API. A form is represented by the **FormGroup** class. Each input field is represented by the **FormControl** class.

__1. In the **model-form/src/app/magazine** folder, open the existing '**magazine.component.ts**' file in a text editor.

__2. At the top of the component code, add the following line for new imports.

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

@Component({
    selector: "app-magazine",
```

__3. Add a new member variable to the MagazineComponent class like this.

```
magazineForm = new FormGroup({
  fullName: new FormControl(''),
  selectedEdition: new FormControl(this.editions[0]),
  selectedShipping: new FormControl(''),
  acceptPolicy: new FormControl(false)
})
```

Notice how we supply the initial value of the input fields as the first argument of the FormControl constructor. This works well when the initial value is known when the FormControl is constructed. If the initial value is available asynchronously at a later time we have to take another approach.

Part 4 - Handle Form Submission

A reactive form explicitly retrieves the input values instead of relying on automatic data binding. The process is actually quite easy.

__1. Modify the code of the '**submitForm**' as follows.

```
submitForm() {
    let requestData = this.magazineForm.value

    alert(JSON.stringify(requestData))
}
```

Basically the value property of FormGroup gives us all the input values in a single object.

__2. Save the file.

Part 5 - Modify Component Template

Right now, the template for the magazine component is still using template driven forms. There will be several changes needed to turn the HTML template into something appropriate for a reactive form. The good thing is the code will generally be much simpler in the template since more is done in the component class.

__1. Open '**magazine.component.html**' file in a text editor.

__2. Change the <form> element as shown in bold face.

```
<form [formGroup]="magazineForm" (ngSubmit)="submitForm()">
```

Note: One difference here is that we're linking the 'submitForm' method to the submit event of the entire form. In the component driven form it was linked to clicking a button. This was mainly because for some template driven form elements we didn't need the surrounding form tag. Here we are using it right away to link to the FormGroup model in the component.

__3. Change the full name text box <input> field to this.

```
<input formControlName="fullName" type="text" /><br/>
```

As you can see, the template starts to become a lot simpler for reactive forms.

__4. Change the following code for the <select> element as follows.

```
<select formControlName="selectedEdition" >
    <option *ngFor="let e of editions"
        [ngValue]="e">{{e.editionName}}</option>
</select><br/>
```

Note: In this code the <option> tag is exactly like it was with the template driven form. You still use the '*ngFor' directive to iterate over the options and '[ngValue]' for what each option value should be.

__5. Change code for the shipping option radio buttons.

```
<input type="radio" formControlName="selectedShipping"
    value="GROUND"/>Ground
<input type="radio" formControlName="selectedShipping"
    value="AIR"/>Air <br/>
```

__6. Change the code for the accept terms checkbox.

```
<label>
<input formControlName="acceptPolicy" type="checkbox" />
I accept the terms and conditions</label>
```

__7. Change the code to display the price.

```
Price: {{magazineForm.value.selectedEdition.price}}
```

__8. We no longer need to handle the click event for the submit button. Instead we need to set its type to submit. That way the ngSubmit event for the entire form will be fired when the button is clicked. Change the button like this.

```
<button type="submit">Purchase</button>
```

__9. Finally, we should delete some validation code that will no longer work. Delete:

```
<span style="color: red"
  [hidden]="theModel.valid || theModel.untouched">
Please enter your name<br/></span>
```

And:

```
Valid: {{theModel.valid}}<br/>
Touched: {{theModel.touched}}<br/>
Dirty: {{theModel.dirty}}<br/>
```

__10. Save the file.

Part 6 - Test Basic Form

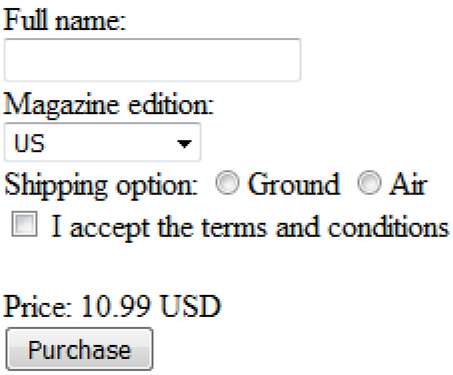
Now that you have the basic elements of the form implemented in the reactive style, you can test the form. This will verify it works before adding more complex things like Validation.

__1. Open a command prompt and go to the root folder of the project **C:\LabWork\model-form**.

__2. Run the following command.

```
npm start
```

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



Full name:

Magazine edition: US

Shipping option: ○ Ground ○ Air

□ I accept the terms and conditions

Price: 10.99 USD

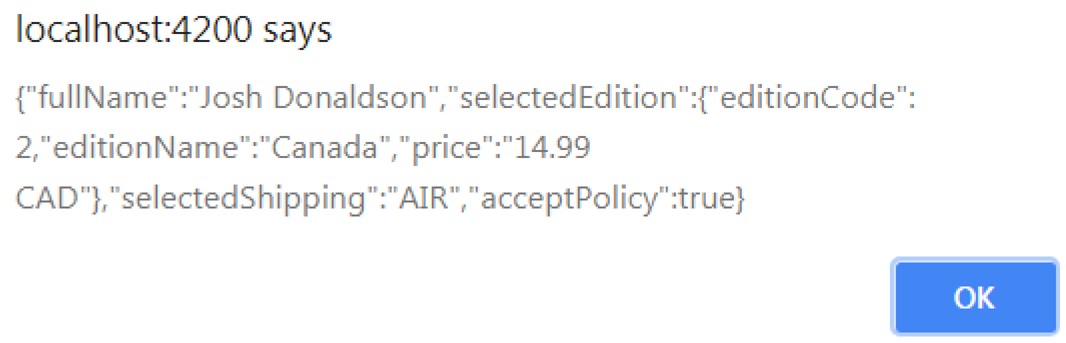
__4. Enter a name in the full name field.

__5. Change the magazine edition. Verify that the price display changes correctly.

__6. Select one of the shipping options.

__7. Check the box to accept terms.

__8. Click the **Purchase** button.



localhost:4200 says

{"fullName":"Josh Donaldson","selectedEdition":{"editionCode":2,"editionName":"Canada","price":"14.99 CAD"},"selectedShipping":"AIR","acceptPolicy":true}

__9. Notice that the property names of the object match the form control names.

__10. Switch the drop-down to a different magazine edition and check that the 'productCode' property in the resulting JSON is still set correctly.

__11. Leave the browser and page open for the future.

Part 7 - Add Form Validation

Just like with template driven forms, reactive forms can have validation logic. More of this logic is in the code of component although you can also use various form object properties in the HTML template of the component to do things like disable buttons of invalid forms or display error messages next to incorrect form fields.

__1. Open '**magazine.component.ts**'.

__2. Apply the required rule to the fullName FormControl in the magazineForm method as shown in bold face below.

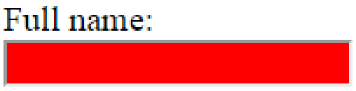
```
fullName: new FormControl('', [Validators.required]),
```

__3. Save the changes to the component.

__4. Check there are no issues recompiling the code and the web browser should refresh automatically.

__5. Make sure that the full name field is shown using the usual white background.

__6. Click inside the text box and immediately click outside. Now the background should change to red.



Full name:

Note: The styles are being applied because you left the CSS of the component the same and only changed the 'template' code.

__7. Fill in some kind of value for the name and verify the text box shows up in white again.

__8. Test that the rest of the form still functions as expected.

Part 8 - Use Form Validation Status Properties

Just like with template driven forms, reactive forms have various validation status properties.

__1. In the '**magazine.component.html**' file, add the following text and expressions to display various validation properties of the form.

```
Full name:<br/>
<input formControlName="fullName" type="text" /><br/>
Valid: {{magazineForm.controls.fullName.valid}}<br/>
Touched: {{magazineForm.controls.fullName.touched}}<br/>
Dirty: {{magazineForm.controls.fullName.dirty}}<br/>
Magazine edition:<br/>
<select formControlName="selectedEdition" >
```

__2. Between the label for the name textbox and the textbox itself, add the following '**<span>**' tag that will be hidden based on some attributes of the displayed form.

```
<form [formGroup]="magazineForm" (ngSubmit)="submitForm()">
    Full name:<br/>
    <span style="color: red"
        [hidden]="magazineForm.controls.fullName.valid ||
            magazineForm.controls.fullName.untouched">
    Please enter your name<br/></span>
    <input formControlName="fullName" type="text" /><br/>
    Valid: {{magazineForm.controls.fullName.valid}}<br/>
```

__3. Scroll down to the very bottom of the component template and add the following to the submit button to disable it if the form is not valid.

```
<button
  [disabled]="magazineForm.invalid || !magazineForm.value.acceptPolicy"
  type="submit">Purchase</button>
```

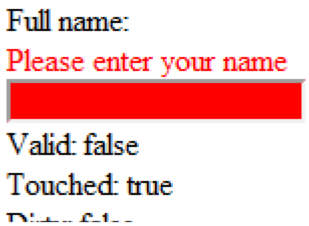
__4. Save the changes to the component.

__5. Check there are no issues recompiling the code and the web browser should refresh automatically.

__6. Make sure that the error message is not displayed by default.

__7. Check that the 'Purchase' submit button is disabled by default.

__8. Click the full name text box and immediately click outside. You should see the error message. Also check the form validation properties displayed by text are changing.



Full name:

Please enter your name

Valid: false

Touched: true

Dirty: false

__9. Start typing into the full name text box and click the checkbox. The **Purchase** button should become enabled and the error message should go away.

__10. Check that the rest of the function of the form still works as before.

__11. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__12. Close all open files.

Part 9 - Review

In this lab, you developed a reactive form. By implementing the same form as in the template driven form lab, you saw some of the differences between the two styles. With a model driven form there is more in the code of the component and a (sometimes small) reduction in the code of the component's HTML template.

Lab 13 - Service and Dependency Injection

In our ongoing fake news web site the news data is currently being served up by the NewsList component itself. It is a better idea to isolate non-GUI business logic in a service. This makes the code more reusable and testable.

In this lab we will develop a simple service that provides a list of news items. We will use the service from the NewsList component.

This lab builds on the previous "**Structural Directives**" lab.

Part 1 - Create the Service Class

In real life you will create a service using the **ng generate service** command. But we will do everything manually in this lab. This will help us learn the mechanics of creating a service.

__1. In the C**:/LabWork/directive-test/src/app/news-list** folder, create a file called **news.service.ts**

__2. Every service class needs to be decorated with @Injectable. So let's import that name first.

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

__3. Then add the shell of the class like this.

```
@Injectable({
    providedIn: 'root'
})
export class NewsService {

}
```

A service is a plain TypeScript class decorated with @Injectable. This is because a user of a service obtains a reference to a service class instance via dependency injection (DI).

By setting the providedIn property to 'root' we are making the service injectable throughout our application including in all feature modules if any.

__4. In the same file add a new class called News. This will represent each news item.

```
export class News {
    title: string
    body: string
}
```

__5. Open **directive-test/src/app/news-list/news-list.component.ts**

__6. Cut the entire declaration of the **newsItems** variable from the **NewsList** component class and paste it inside the **NewsService** class. Make sure to include the closing square bracket for the entire array content.

```
export class NewsService {
  newsItems = [
    array contents ...
  ]
}
```

__7. Explicitly state the data type of the newsItems variable.

```
newsItems : News[] = [...]
```

__8. Add this method to the **NewsService** class.

```
getNewsItems() : News[] {
    return this.newsItems
}
```

__9. Save <u>all</u> files.

Part 2 - Use the Service from a Component

__1. Open **directive-test/src/app/news-list/news-list.component.ts**

__2. Import the service class into namespace.

```
import { Component } from '@angular/core';
import { NewsService, News } from './news.service';
```

__3. In the class declare a variable that will hold the list of news items.

```
newsItems : News[] = []
```

__4. Modify the constructor to inject an instance of NewsService.

```
constructor(private newsSvc:NewsService) { }
```

__5. From the ngOnInit() method retrieve the list of news items.

```
ngOnInit() {
    this.newsItems = this.newsSvc.getNewsItems()
}
```

__6. Save changes.

__7. Open **news-list.component.html** and just review it. We do not need to change anything there.

Part 3 - Test

__1. Open a command prompt and go to the root folder of the project:

```
C:\LabWork\directive-test
```

__2. Run the following command to compile TypeScript and run the server:

```
npm start
```

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

The site should work same as before. Except now the data access logic has been moved to a service.

__4. In the command prompt, hit **'<CTRL>-C'** to terminate the batch job.

__5. Close all open files.

Part 4 - Review

In this lab we developed a service and used it from a component. To use a service from a component we had to first inject an instance of the service as a member variable of the component. Dependency Injection has a few advantages:

1. A component does not have to know how to construct a service instance. For example if the service S1 depends on other services (S2 and S3) the entire chain of dependency will be constructed on demand. The component does not need to supply the other dependencies (S2 and S3) in the constructor argument for S1. This makes coding easier and safer.
2. Instances can be reused. When a component injects a service instance all child components inject the same exact instance.
3. During testing you can provide a mock version of a service. Existing components or services that depend on that service do not have to be modified.

Lab 14 - HTTP Client Development

So far in our fake news web site we have hard coded the data. Now we will fetch the data from a web service. Angular provides the **HttpClient** service for this purpose. This services uses the RxJS library for asynchronous programming.

This lab builds on the prior "**Services and DI**" Lab.

Part 1 - Develop the Web Service

To save time we won't really write any code for the web service. We will simply drop a JSON file into the project that will be served by the embedded server. In real life you will probably use the platform of your choice (such as Java, .NET and NodeJS) to build the web service.

__1. Copy **C:\LabFiles\httpClient\news_data.json** and paste it inside the **C:\LabWork\directive-test\src\assets** folder

__2. Open a command prompt and go to the root folder of the project:

```
C:\LabWork\directive-test
```

__3. Run the following command to compile TypeScript and start the server:

```
npm start
```

If the server was already running then stop and restart it.

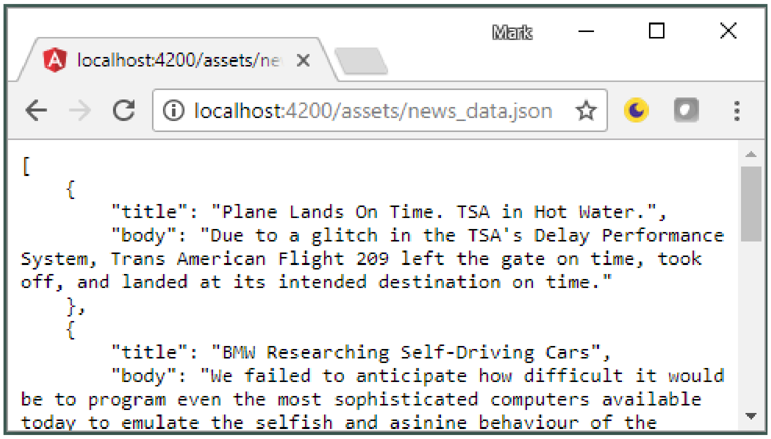
__4. Open the following URL for the application in your browser. The app should work the same way as before:

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

__5. Change the URL of the browser to:

```
http://localhost:4200/assets/news_data.json
```

__6. Depending on how your browser is set up the **news_data.json** file will either show in the browser or be downloaded. Verify that you see the file contents. If you get a 404 'File not Found' error the file is not in the correct location.



```
[
    {
        "title": "Plane Lands On Time. TSA in Hot Water.",
        "body": "Due to a glitch in the TSA's Delay Performance
System, Trans American Flight 209 left the gate on time, took
off, and landed at its intended destination on time."
    },
    {
        "title": "BMW Researching Self-Driving Cars",
        "body": "We failed to anticipate how difficult it would
be to program even the most sophisticated computers available
today to emulate the selfish and asinine behaviour of the
```

Part 2 - Inject the Http Service

Assuming that many of the services in an application will need the Http service it is easier to inject it at the root scope.

__1. Open **directive-test/src/app/app.module.ts**

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

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

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

This will import the HttpClient service and other services in the module that might be used in more advanced scenarios.

__3. Save changes.

Part 3 - Use the HttpClient Service

We will now use the Http service from NewsService to call a web service.

__1. Open **directive-test/src/app/news-list/news.service.ts**

__2. Add various import statements.

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

__3. Delete the entire **newsItems** array variable. We wont need it any more.

__4. Add a constructor in the class that will inject the Http service.

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

__5. Re-write the getNewsItems() method as follows. Make sure to change the signature as well as the return statement.

```
  getNewsItems(): Observable<News[]> {
      return this.httpClient.get<News[]>
          ('/assets/news_data.json')
  }
```

The get() method of Http service returns an Observable<News[]> object. A subscriber can listen for data to arrive in an observable.

__6. Save changes.

Part 4 - Render The Retrieved Data

We will now modify the NewsList component to extract data from the Observable.

__1. Open **directive-test/src/app/news-list/news-list.component.ts**

__2. Change the ngOnInit() method.

```
ngOnInit() {
    this.newsSvc.getNewsItems().subscribe(response => {
        this.newsItems = response;
    })
}
```

__3. Save changes.

Part 5 - Test

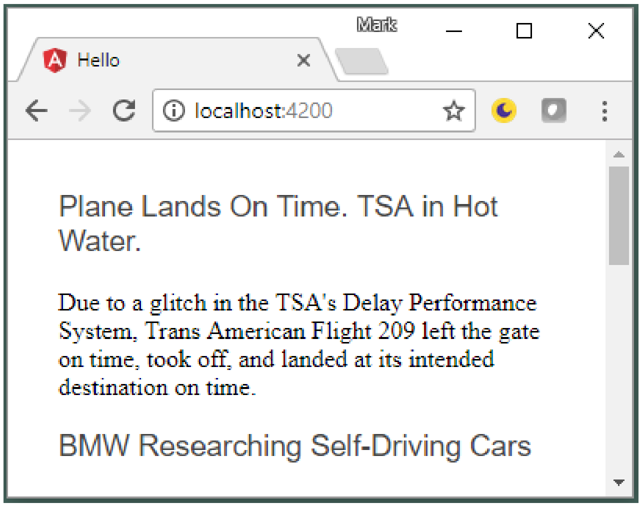
__1. Return to the command prompt and check there are no issues recompiling the code.

__2. Return to the browser. It likely has not refreshed because it was displaying the JSON data.

__3. Change the URL of the browser to:

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

__4. Make sure that you can use the app and the data coming from the json file is shown.



Plane Lands On Time. TSA in Hot Water.

Due to a glitch in the TSA's Delay Performance System, Trans American Flight 209 left the gate on time, took off, and landed at its intended destination on time.

BMW Researching Self-Driving Cars

Part 6 - Error Handling

So far in the subscribe method we provided a callback for the success case. Now we will provide a callback for the error case.

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

__2. In the subscribe method supply a second argument which will handle the error case. Be extra careful to have the correct commas, brackets and parenthesis.

```
ngOnInit() {
    this.newsSvc.getNewsItems().subscribe(response => {
        this.newsItems = response;
    },
    error => {
        alert("Sorry. There was a problem getting data.")
    })
}
```

__3. Save changes.

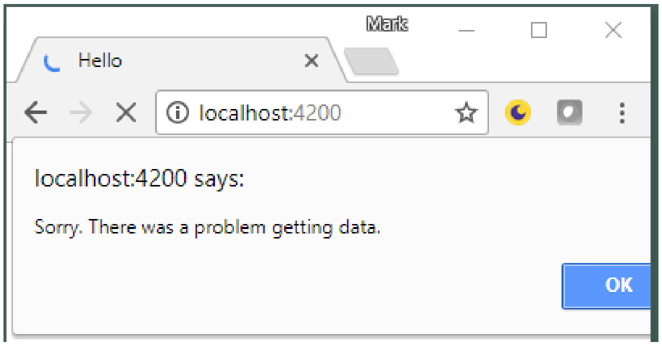
Part 7 - Test

__1. Check there are no issues recompiling the code and the web browser should refresh automatically.

__2. Make sure that the success path is still working.

__3. Rename the **\directive-test\src\assets\news_data.json** file to something else.

__4. Refresh the browser. You should see the error message.



localhost:4200 says:

Sorry. There was a problem getting data.

Troubleshoot: If you don't see the expected error then clean the browser's cache and try again or open the application's URL in in an incognito mode tab. You can also stop and then start the server.

__5. Click OK to get rid of the error popup.

__6. Rename the JSON file back to **news_data.json** and test again.

__7. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__8. Close all open files.

Part 8 - Review

In this lab we learned how to call a web service using the HttpClient service. The HttpClient service hands us an Observable. We can asynchronously obtain data from an Observable by subscribing to it.

Lab 15 - Using Pipes

Pipes are used to transform model data. Common uses are:

- To format data such as date and currency.
- Filter an array.

This lab builds on the prior "**Http Client**" lab.

Part 1 - Using a Built-in Pipe

Using a built-in pipe is super easy. We will now use the uppercase pipe to show the news title in uppercase.

__1. Open **directive-test/src/app/news-list/news-list.component.html**

__2. Change the way the title is rendered as follows.

```
<h3>{{news.title | uppercase}}</h3>
```

__3. Save changes.

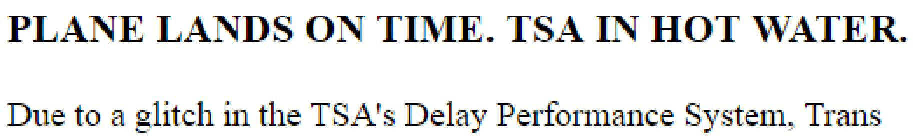
__4. Open a command prompt and go to the root folder of the project:

```
C:\LabWork\directive-test
```

__5. Run the following command.

```
npm start
```

__6. Make sure that the titles show up in uppercase.

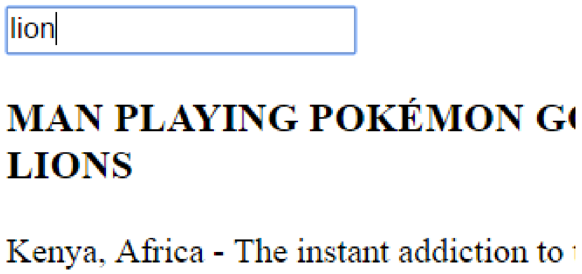


PLANE LANDS ON TIME. TSA IN HOT WATER.

Due to a glitch in the TSA's Delay Performance System, Trans

Part 2 - Write a Custom Pipe

We will now develop a pipe that will let users search for text in the news items.



lion

MAN PLAYING POKÉMON G
LIONS

Kenya, Africa - The instant addiction to

__1. In the **news-list** folder create a new file called **news-search.pipe.ts**

__2. In that file add the skeletal code for the pipe.

```
import { Pipe, PipeTransform } from '@angular/core';
import { News } from './news.service';

@Pipe({
    name: 'newsSearch'
})
export class NewsSearchPipe implements PipeTransform {

}
```

Note:

- The class has to be decorated with @Pipe and it must implement PipeTransform.
- The **name** attribute of the @Pipe decorator indicates how the pipe should be used in HTML template.

__3. Every pipe must implement the **transform** method. Add the method as follows.

```
  transform(sourceList: News[], searchText:string) : News[] {
    if (searchText === undefined || searchText.length == 0) {
      //No search term
      return sourceList
    }

    let reg = new RegExp(searchText, 'i')

    return sourceList.filter(news =>
      news.title.search(reg) >= 0 || news.body.search(reg) >= 0)
  }
```

The transform method takes as input the source data. It can optionally take one or more parameter values. In our case the pipe parameter is the search text. The method returns a transformed value.

__4. Save changes.

Part 3 - Use the Custom Pipe

We will now use the pipe from HTML template. First the pipe needs to be declared in the module.

__1. Open **directive-test/src/app/app.module.ts**

__2. Since we will use two-way binding with '[(ngModel)]', add the import of the Forms module.

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

__3. Import the pipe class into the namespace.

```
import { NewsSearchPipe } from './news-list/news-search.pipe';
```

__4. Add the 'FormsModule' to the @NgModule 'imports' property.

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

__5. Every pipe used must be declared in the **declarations** property of the @NgModule decorator. Add it like this shown in bold.

```
  declarations:  [ AppComponent, NewsListComponent, NewsSearchPipe ],
```

We will now deal with accepting the search text from the user.

__6. Save changes.

__7. Open **directive-test/src/app/news-list/news-list.component.ts**

__8. In the NewsListComponent class, add a member variable for the search text.

```
export class NewsListComponent implements OnInit {
     searchText:string
```

__9. Save the file.

__10. Open **directive-test/src/app/news-list/news-list.component.html**

__11. In the HTML template of the component, add a text box for the search text. This is shown in bold below.

```
<div>
    <input type="text" [(ngModel)]="searchText" placeholder="Search"/>
    <div *ngFor="let news of newsItems; let newsId = index">
```

__12. Now we will use the news search pipe. It will take the search text as parameter. Change the ngFor directive as shown in bold below.

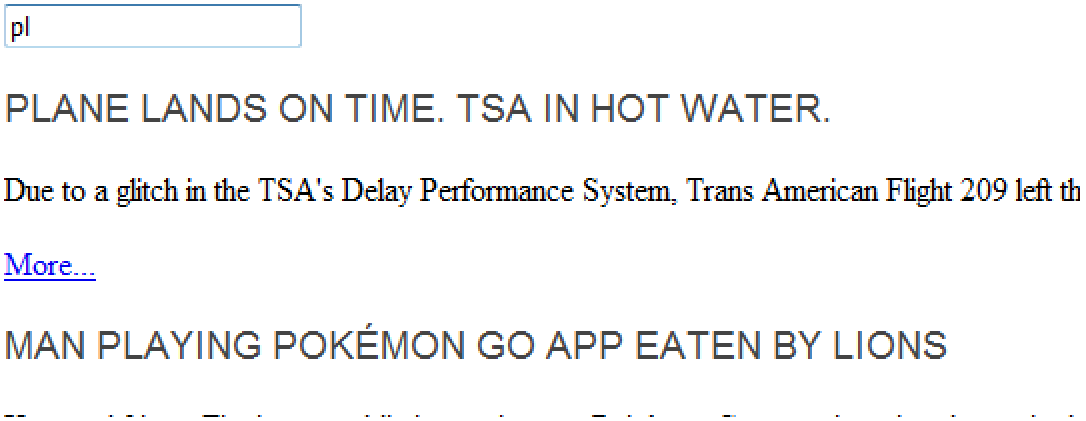
```
<div *ngFor="let news of (newsItems | newsSearch:searchText); let 
newsId = index">
```

__13. Save changes. Make sure all files were saved.

Part 4 - Test

__1. Check there are no issues recompiling the code and the web browser should refresh automatically.

__2. Enter some kind of search text. Verify that you see only news items that have matching text. The pipe will search the title and body so use something that is unique enough to filter some items out.



pl

PLANE LANDS ON TIME. TSA IN HOT WATER.

Due to a glitch in the TSA's Delay Performance System, Trans American Flight 209 left th

More...

MAN PLAYING POKÉMON GO APP EATEN BY LIONS

__3. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__4. Close all open files.

Part 5 - Review

In this lab we learned how to use a built-in pipe as well develop our own pipe.

Lab 16 - 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. Open a command prompt window.

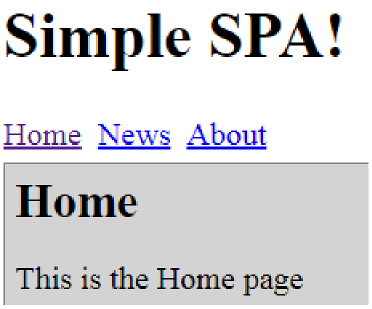
__2. Go to the **C:\LabWork** folder.

__3. Run this command to create a new project called **route-test**.

```
ng new route-test --routing --style css
```

> **No Internet Access?**
>
> Extract C:\LabFiles\project-seed.zip to the C:\LabWork Directory.
>
> This should create the C:\LabWork\project-seed folder.
>
> Rename the C:\LabWork\project-seed to 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. Change to the root project.

```
cd C:\LabWork\route-test
```

__2. Run these commands to create the components.

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

The shortcut **g** stands for generate and **c** for component.

__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 **route-test/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. Edit **\src\app\app.component.html** and replace all of its contents with the code below:

```
  <h1>Simple SPA!</h1>
  <a [routerLink]="['/']">Home</a>&nbsp;
  <a [routerLink]="['/news']">News</a>&nbsp;
  <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.

__2. Save changes.

__3. Edit **\src\styles.css**

__4. 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;
}
```

__5. Save changes.

Part 6 - Test

__1. Open a command prompt and go to the root folder of the project:

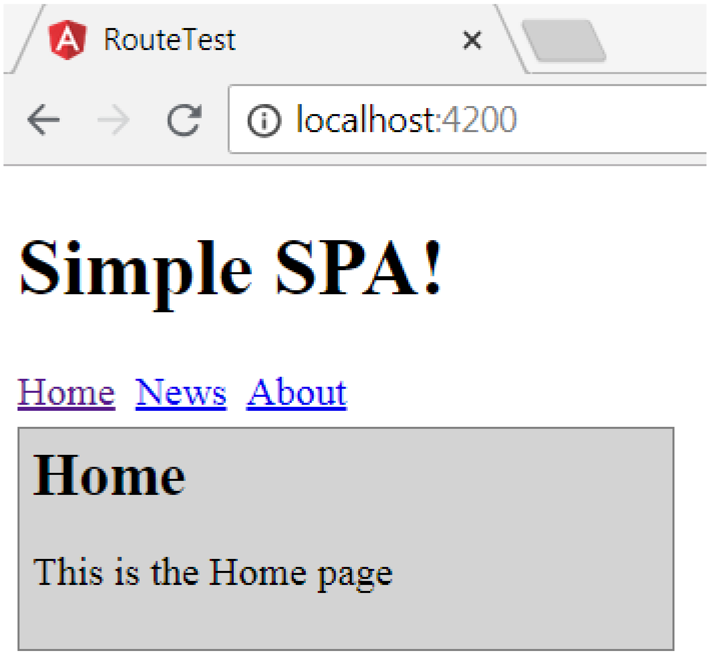
```
C:\LabWork\route-test
```

__2. Run the following command to compile TypeScript and start the server:

```
npm start
```

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

__4. Verify that the home page is shown by default.



Simple SPA!

Home News About

Home

This is the Home page

__5. 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.

__6. Verify that the back button navigation works as expected.

__7. Navigate to the About page.

__8. 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.

__9. In the command prompt, hit '**<CTRL>-C**' to terminate the embedded server.

__10. Close all open files.

Part 7 - Review

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

Lab 17 - Angular Communication with REST Services

The Angular HTTP client service provides the ability to interact with REST web services from Angular applications. It is a best practice to perform this interaction from a custom service instead of doing so from a component.

In this lab you will implement various methods in a custom service that will provide data to a completed component to display this data. Once the methods of the service are implemented, the features of the component displaying data will become active.

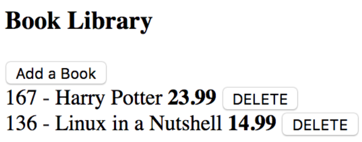
Part 1 - Business Requirements

In this lab we will create a book database management system. 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/***ISBN***	GET	Returns a specific book given its ISBN. Example: /books/123-456
/books/***ISBN***	DELETE	Deletes a specific book given its ISBN. Example: /books/123-456
/books/***ISBN***	PUT	Adds or updates a book.

Our application will need to develop these pages.

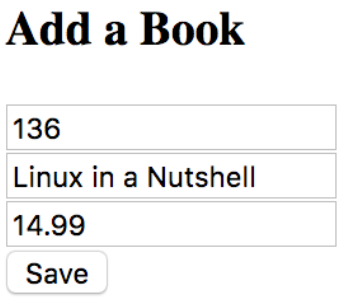


Book Library

167 - Harry Potter **23.99**

136 - Linux in a Nutshell **14.99**

Book List Page



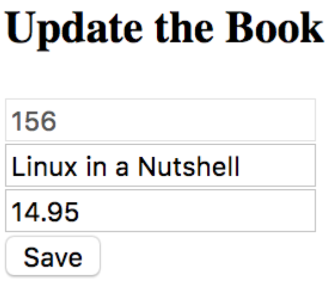
Add a Book

136

Linux in a Nutshell

14.99

Add Book Form



Update the Book

156

Linux in a Nutshell

14.95

Update Book Form

Part 2 - Get Started

We will now create a new Angular application.

__1. Open a command prompt window.

__2. Go to the **C:\LabWork** folder. Create the folder if it hasn't been created.

__3. Run this command to create a new project called **rest-client**.

```
ng new rest-client --routing --style css
```

No Internet Access?

Extract C:\LabFiles\project-seed.zip into the C:\LabWork directory.

This should create a C:\LabWork\project-seed folder.

Rename the C:\LabWork\project-seed to C:\LabWork\rest-client.

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. Open **rest-client\src\app\app.module.ts**

__2. Add these import statements.

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

__3. Import the modules as shown in bold face below.

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

This is Angular module import as opposed to the ES6 import in the previous step. This will automatically add all the services exported by HttpClientModule (such as HttpClient) to the providers list of our application module. This will let us inject HttpClient throughout our application.

__4. 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. Open a new command prompt window and from the root of the **rest-client** project run this command.

```
ng generate service data
```

Angular CLI not Installed Globally?

Run: npm run ng generate service data

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

__2. Open **C:\LabWork\rest-client\src\app\data.service.ts**

__3. Add these import statements.

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

Important: In prior versions of Angular 'Observable' was imported from the 'rxjs/Observable' or 'rxjs/Rx' package. Older tutorials online will show those packages. With Angular 6 and later versions you should import Observable from 'rxjs' package.

__4. Add the Book class like this.

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

__5. In the constructor of the DataService class inject the HttpClient service.

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

__6. Save changes.

__7. 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.

__8. Add the '**getBook**' method.

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

__9. 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>.

__10. 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)
}
```

__11. Save changes.

Part 5 - Generate the Components

__1. 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 except for the last one. All that will be left is this.

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

The routed components will be rendered there.

__7. 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. One way to fix this issue is 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\rest-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\rest-server** directory. Then execute the following command:

```
npm start
```

__2. To start the Angular application server open another command prompt and navigate to the project directory: **C:\LabWork\rest-client**. From that directory run the following command to compile Typescript and start the embedded server:

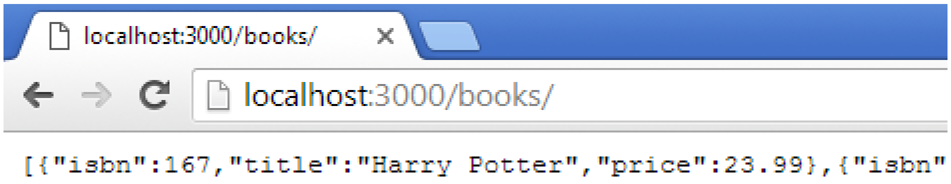
```
npm start
```

Now we see the benefit of starting the dev server using npm start rather than ng serve. We don't have to specify the proxy config file every time we start the server.

__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.



```
[{"isbn":167,"title":"Harry Potter","price":23.99},{"isbn"
```

__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:

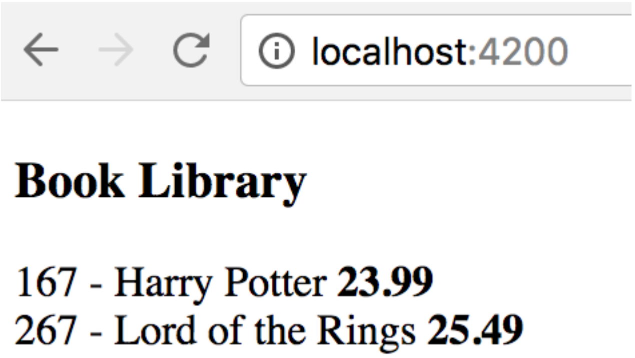
```
http://localhost:4200/books/
```

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>
```

__9. Save changes.

We now have to add a button the BookListComonent 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.

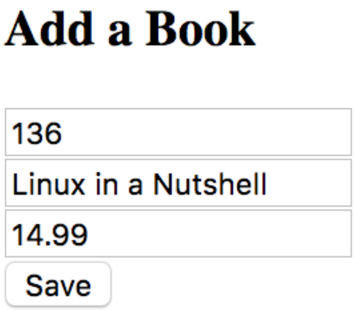
Part 12 - Test Changes

__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

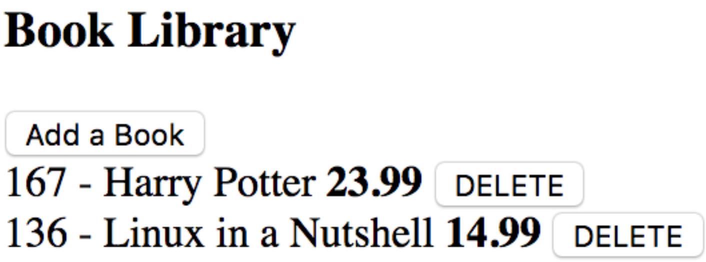
136

Linux in a Nutshell

14.99

__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

167 - Harry Potter **23.99**

136 - Linux in a Nutshell **14.99**

__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.params.subscribe(params => {
    let isbn = params['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.

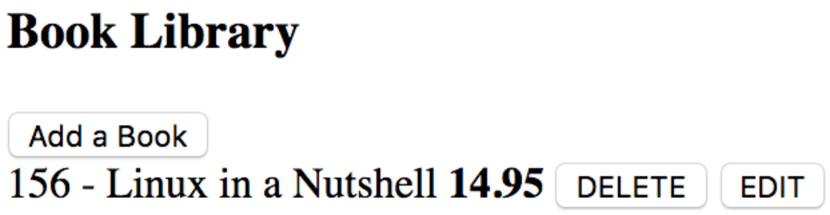
```
<button (click)="deleteBook(book)">DELETE</button>
<button [routerLink]="['/edit-book', book.isbn]">EDIT</button>
```

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.

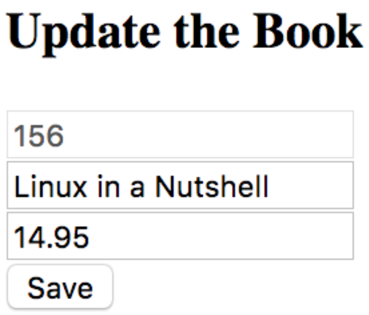


Book Library

156 - Linux in a Nutshell **14.95**

__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

__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 18 - HTTP Error Handling and Recovery

In this lab we will go through some of the strategies used in error handling. Namely:

- How to show error messages when a HTTP call fails.
- How to recover from error by returning cached data.
- How to attempt to recover from error by retrying HTTP calls.

We will continue to the book database application and add error handling 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. Open **rest-client/src/app/book-list/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 **app/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/${book.isbn}`)
}
```

__6. Save changes.

__7. The backend web service is available in the **C:\LabFiles\rest-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\rest-server** directory. Then execute the following command:

```
npm start
```

__8. To start the Angular application server open another command prompt and navigate to the project directory: **C:\LabWork\rest-client**. From that directory run the following command to compile Typescript and start the embedded server:

```
npm start
```

__9. Open a browser to:

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

__10. Try to delete a book. You should see the alert error message.

__11. Fix the error introduced in **data.service.ts**.

__12. Save changes.

Now, let's simulate a network communication error.

__13. Shutdown the Angular CLI server (not the REST server) by going to the command prompt and pressing 'CTRL-C'.

__14. Back in the browser try to delete a book. You will get the error alert once again.

Part 2 - 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 3 - 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 4 - Test Caching

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

```
npm start
```

__2. Open a browser and enter the URL:

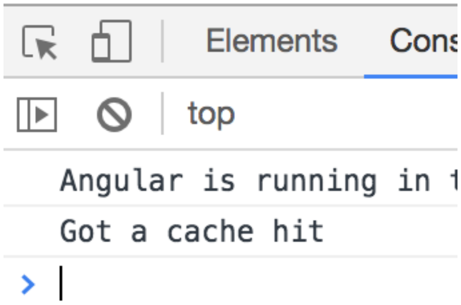
```
http://localhost:4200/
```

__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.



```
Angular is running in t
Got a cache hit
```

This verifies that our caching is working.

__7. Go back to the main page.

`http://localhost:4200/`

Part 5 - 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 6 - 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 7 - 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/bad-url/${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 8 - 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 9 - 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 10 - 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 19 - Using Angular Bootstrap

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

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

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 CSS

__1. From the **C:\LabWork\rest-client** folder run:

```
npm install bootstrap@4.3.1 --save
```

__2. Open **rest-client/src/styles.css**

__3. Add this line to include bootstrap CSS within styles.css.

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

__4. Save changes.

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. Open a command prompt and from **C:\LabFiles\rest-server**, start the server:

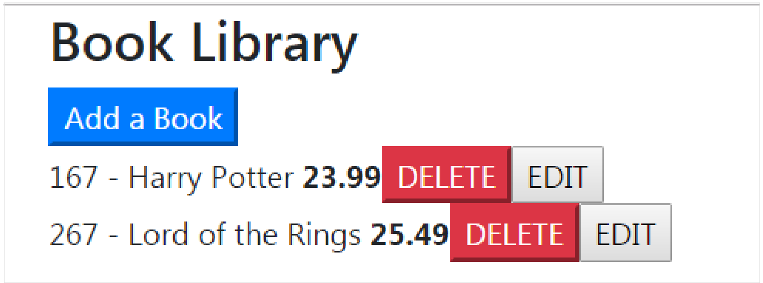
```
npm start
```

__2. Open a command prompt and from **C:\LabWork\rest-client**, run the dev server.

```
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.



Book Library

167 - Harry Potter **23.99**

267 - Lord of the Rings **25.49**

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" ...>
```

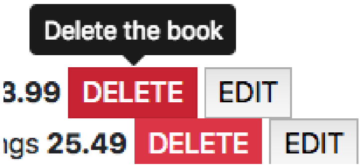
__4. Save changes.

__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 and Angular module called NgbModule. This is available from the NPM package called @ng-bootstrap/ng-bootstrap which we need to install first.

__1. Open a new command prompt windows and from the **C:\LabWork\rest-client** run:

```
npm install --save @ng-bootstrap/ng-bootstrap@5.0.0-rc.1
```

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

__2. Open **app.module.ts**

__3. Import the name.

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

__4. Then add the module to the imports list like this.

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

__5. Save changes.

__6. Open **book-list.component.html**

__7. 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>
```

__8. Save changes.

__9. 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.

```
private page = 1; //Current page. Starts with 1
private 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.

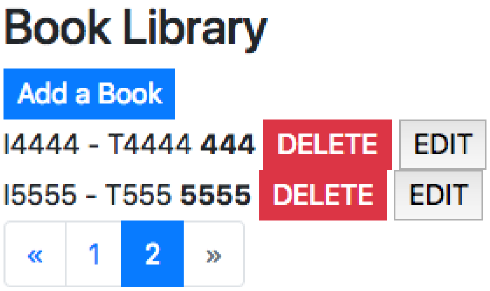
__5. Open **book-list.component.html**

__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.



Book Library

I4444 - T4444 444

I5555 - T555 5555

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 20 - Consuming Data from Web Sockets

Angular applications can be set up to exchange data in real time using web sockets. In this lab we will use a custom node.js based web socket server to deliver stock quotes to our Angular application. The application will use a code library (sockets.io-client) to open a connection to the server and initiate communications. Once the connection is open the server will push quotes to the client at regular intervals over the socket connection.

The lab consists of the following parts:

- Lab Setup
- Review the Quote Server
- Add an Angular Websocket Service
- Hook up the Quote Component and Websocket Service
- Test the Application

Part 1 - Get Started

We will now create a new Angular application.

__1. Open a command prompt window.

__2. Go to the **C:\LabWork** folder.

__3. Run this command to create a new project called **ws-client**.

```
ng new ws-client --defaults
```

No Internet Access?

Extract C:\LabFiles\project-seed.zip into the C:\LabWork directory.

This should create a C:\LabWork\project-seed folder.

Rename the C:\LabWork\project-seed to C:\LabWork\ws-client.

__4. Install the Socket.io client package.

```
cd ws-client

npm install --save socket.io-client@2.1.1
```

Part 2 - Start the Websocket Server

We will now start the server that will periodically emit updated stock quotes.

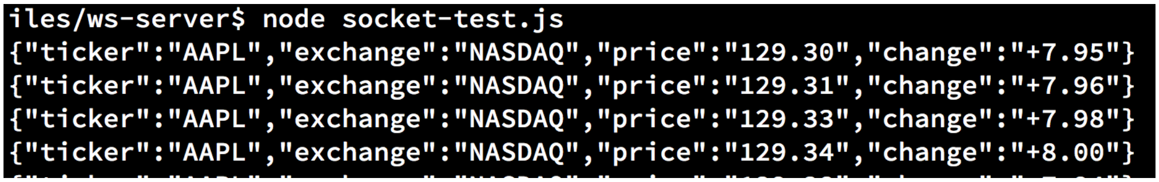
__1. From a new command prompt widow, go to **C:\LabFiles\ws-server**.

__2. Start the server by entering this command.

```
npm start
```

__3. Open a new command prompt window and enter these commands to make sure that the server is working.

```
cd C:\LabFiles\ws-server
node socket-test.js
```



```
iles/ws-server$ node socket-test.js
{"ticker":"AAPL","exchange":"NASDAQ","price":"129.30","change":"+7.95"}
{"ticker":"AAPL","exchange":"NASDAQ","price":"129.31","change":"+7.96"}
{"ticker":"AAPL","exchange":"NASDAQ","price":"129.33","change":"+7.98"}
{"ticker":"AAPL","exchange":"NASDAQ","price":"129.34","change":"+8.00"}
```

This will also show you the data structure of each message emitted by the server.

__4. Hit **Control+C** to end the test. But keep the server running.

__5. Next we need to setup the reverse proxy. Copy the file:

```
filename: proxy.conf.json
from: C:\LabFiles\ws-client\
to: C:\LabWork\ws-client\
```

__6. Edit the **C:\LabWork\ws-client\package.json** file.

__7. Replace the 'start' line in the scripts section with the following and save the file:

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

__8. Save changes.

We are now ready to start working on our Angular application.

Part 3 - Add an Angular Service

We will now add a service to our Angular application that will interface with the quote Websocket server. The socket.io-client package has a callback based API. Every time the server emits a message the API will call your callback. The challenge will be to convert this API to be RxJS Observable based. That will make it easier for our Angular application to get the quote messages.

__1. Open a command prompt windows and from **C:\LabWork\ws-client**, run this command to create the service.

```
ng g s quote
```

__2. Open **src/app/quote.service.ts** in your text editor.

We will take a look at the existing code and see how it works.

__3. Add these import statements at the top of the file:

```
import { Observable, Subscriber } from 'rxjs';
import * as io from 'socket.io-client';
```

__4. Add the Quote class that represents each message sent by the server.

```
export class Quote {
  ticker: string
  exchange: string
  price: string
  change: string
}
```

__5. Inside the QuoteService class add a member variable for the socket connection. The socket.io-client package doesn't come with any type information. So, we will use the any type.

```
socket: any;
```

__6. Add the getQuotes() method like this.

```
getQuotes() : Observable<Quote> {
  const observable = Observable.create(
    (observer: Subscriber<Quote>) => {
      this.socket = io('ws://localhost:4200');

      this.socket.emit('getquotes', 'start');

      this.socket.on('newquote', (data: string) => {
        observer.next(JSON.parse(data));
      });
    });

  return observable;
}
```

Basically, every time we get a new quote message we deliver the message to the subscriber.

__7. Add this method that will disconnect from the quote server.

```
disconnect() { this.socket.disconnect(); }
```

__8. Save the file.

Part 4 - Develop the Component

__1. Run this command to create the component.

```
ng g c quote
```

__2. Open the **src/app/quote/quote.component.ts** file in your text editor.

__3. Add this import statement.

```
import { QuoteService, Quote } from '../quote.service';
```

__4. Add a member variable that will have the latest quote message.

```
lastQuote:Quote
```

__5. Inject the QuoteService instance in the constructor.

```
constructor(private quoteSvc : QuoteService) { }
```

__6. From the ngOnInit() method start to subscribe for messages.

```
ngOnInit() {
  this.quoteSvc.getQuotes()
    .subscribe(quote => this.lastQuote = quote)
}
```

__7. Save the file.

__8. Open **quote.component.html**

__9. Change the template to be like this.

```
<h3>Latest Quote</h3>
<div *ngIf="lastQuote != undefined">
  <b>Symbol: </b> {{lastQuote.ticker}}<br/>
  <b>Price: </b> {{lastQuote.price}}<br/>
  <b>Change: </b> {{lastQuote.change}}<br/>
</div>

<div *ngIf="lastQuote == undefined">
  Connecting...
</div>
```

__10. Save changes.

__11. Open **app.component.html**

__12. Replace all content with this.

```
<app-quote></app-quote>
```

__13. Save changes.

Part 5 - Apply Polyfill

At the time of this writing there is a defect in socket.io-client that prevents it from working with Angular (https://github.com/socketio/socket.io-parser/issues/87). We will now fix it using polyfill.

__1. Open **src/polyfills.ts**

__2. At the very bottom add:

```
(window as any).global = window
```

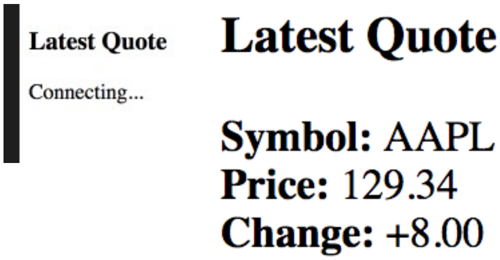
__3. Save changes.

Part 6 - Test the App

__1. From the **C:\LabWork\ws-client** folder run:

```
npm start
```

__2. Open **http://localhost:4200/** in the Chrome browser.



Latest Quote

Connecting...

Latest Quote

Symbol: AAPL
Price: 129.34
Change: +8.00

After a brief period, a quote messages should start to appear on screen.

__3. Close the browser window.

__4. Shut down both servers. (enter Ctrl-C in the server command prompts windows)

__5. Close all open files.

Part 7 - Review

In this lab we developed a client for a websocket server.

Lab 21 - 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. Open a command prompt window.

__2. Go to the **C:\LabWork** folder.

__3. Run this command to create a new project called **large-app**.

```
ng new large-app --routing --style css
```

Part 2 - Create the Main Module Components

__1. In the command prompt window go to the large-app folder.

```
cd large-app
```

__2. Run these commands.

```
ng g component home

ng g component catalog
```

Part 3 - Create the Feature Module

__1. 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.

```
  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 **large-app/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:
"./account/account.module#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 the path to the feature module path relative to the location of app-routing.module.ts.

__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. 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>
```

__3. Save changes and close the file.

Part 8 - Test

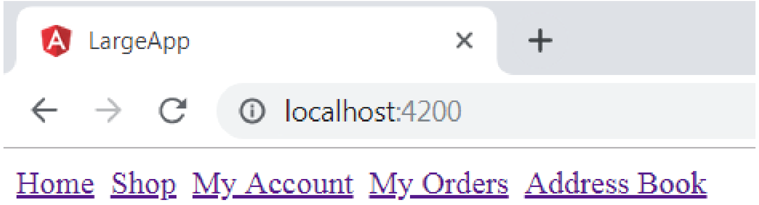
We will now use the application and verify that AccountModule is getting loaded lazily when user navigates to one of its components.

__1. From the command line run.

```
npm start
```

__2. Open a browser and go to **http://localhost:4200/**

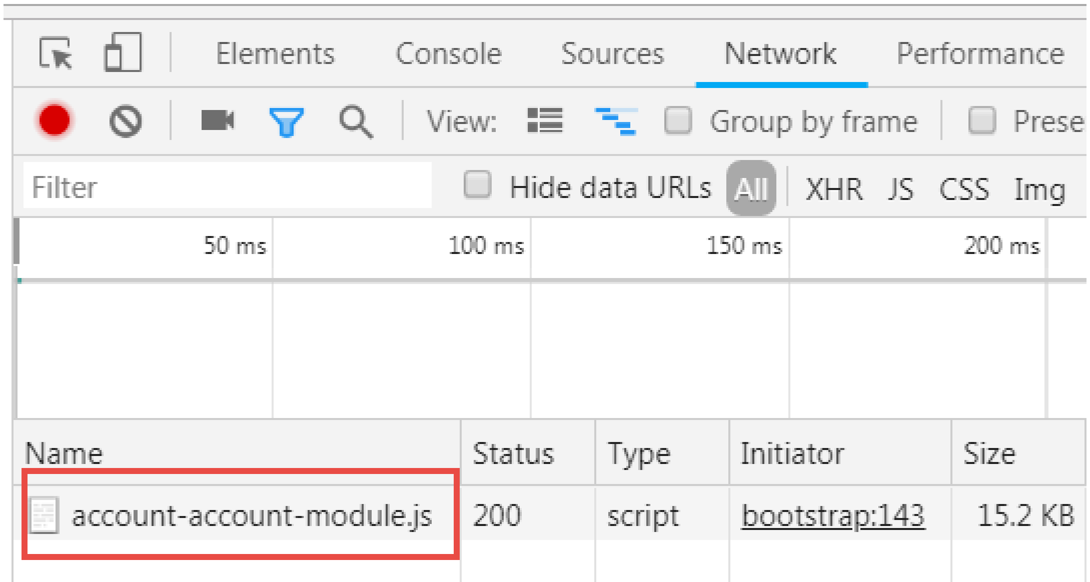
__3. You should see this.



__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.



Name	Status	Type	Initiator	Size
account-account-module.js	200	script	bootstrap:143	15.2 KB

__7. Test the other links.

__8. Hit Control+C to end the web server.

Part 9 - Run a Build

__1. From the command line run:

```
ng build
```

__2. Look at the **dist\large-app** folder. 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.

__3. Close all.

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. Two key aspects of this are:

- Routes for a feature module must be configured using RouterModule.forChild().
- A route for the feature model is configured like this:

```
{path: "account", loadChildren:
"app/account/account.module#AccountModule"}
```

Lab 22 - 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. Open a command prompt window.

__2. Go to the **C:\LabWork** folder.

__3. Run this command to create a new project called **advanced-routing**.

```
ng new advanced-routing --defaults
```

No Internet Access?

Extract C:\LabFiles\project-seed.zip into the C:\LabWork directory.

This should create a C:\LabWork\project-seed folder.

Rename the C:\LabWork\project-seed to C:\LabWork\advanced-routing.

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. Open a command prompt and go to the root folder of the project:

```
cd C:\LabWork\advanced-routing
```

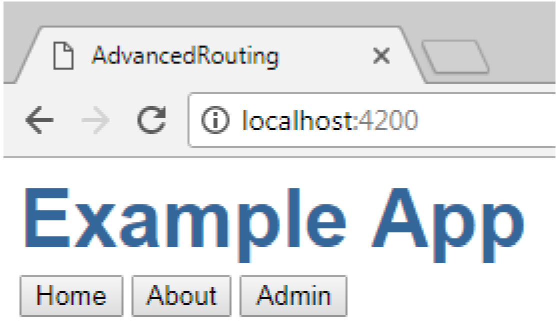
__4. Run the following command to compile TypeScript and start the server:

```
npm run start
```

__5. Open a browser window to the following address:

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

__6. The browser window should appear like this:



The lab setup is complete.

Part 3 - Defining a Default Route

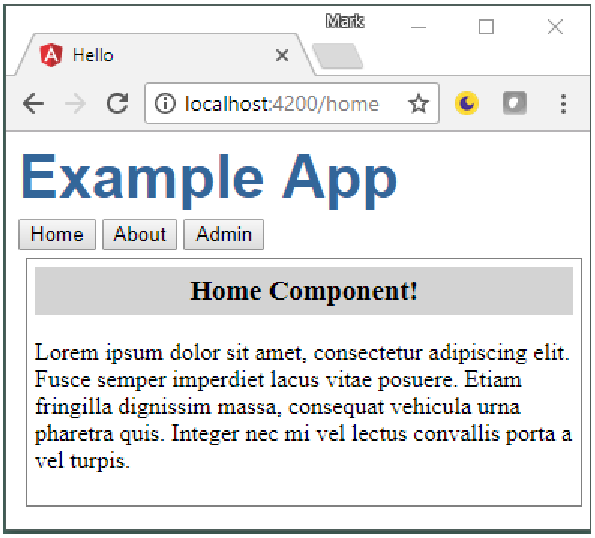
Description.

__1. Take a look at the browser window and take note of the following:

- The address bar shows: localhost:4200
- The title 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.



Example App

Home Component!

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce semper imperdiet lacus vitae posuere. Etiam fringilla dignissim massa, consequat vehicula urna pharetra quis. Integer nec mi vel lectus convallis porta a vel turpis.

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 the following file in your text editor (path is referenced from the project root):

\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 }
];
```

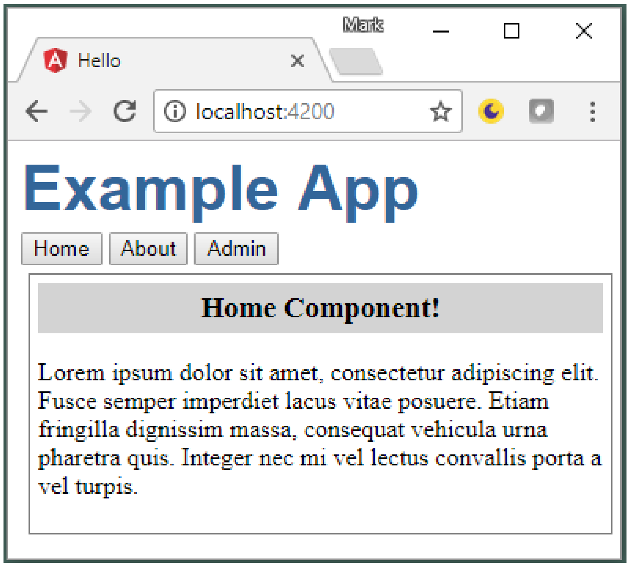
__5. Save the file.

__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.



Example App

Home Component!

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce semper imperdiet lacus vitae posuere. Etiam fringilla dignissim massa, consequat vehicula urna pharetra quis. Integer nec mi vel lectus convallis porta a vel turpis.

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:

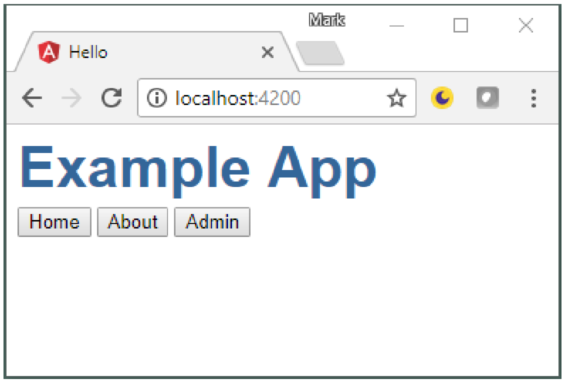
- "" (no path)
- "/home"
- "/about"
- "/admin"

__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. But before we can add it to the route array we need to do some setup for the component.

__4. Open the following file in your text editor:

\src\app\app.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 component to the declarations array:

```
declarations: [
  AppComponent,
  HomeComponent,
  AboutComponent,
  AdminComponent,
  RouteErrorComponent
]
```

__7. Save and close the app.module.ts file.

__8. Open the following file in your text editor:

\src\app\app-routing.module.ts

__9. Add the following import after the other imports at the top of the file:

```
import { RouteErrorComponent } from './error/route.error.component';
```

__10. 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' }
];
```

__11. Save and close **app-routing.module.ts**

__12. Close all open editors.

__13. Close the current browser window.

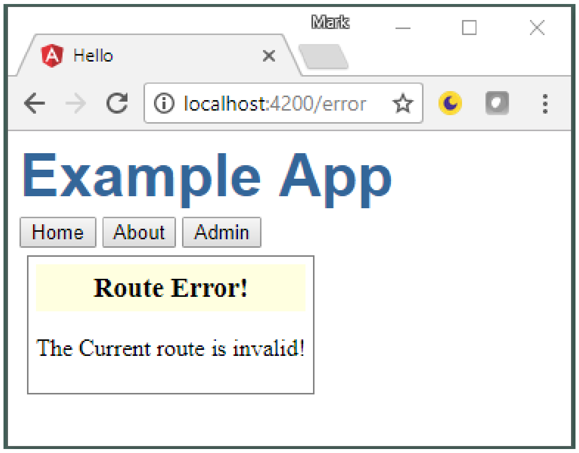
__14. Open a new browser window with the following url which contains an invalid path:

`http://localhost:4200/pathx`

__15. 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:



Route Error!

The Current route is invalid!

__16. 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 in your text editor:

```
\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 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" >
```

__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.

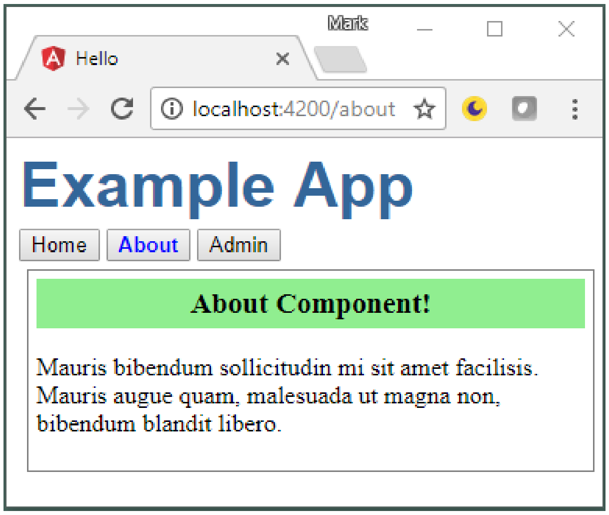
```
.active{
    color: blue;
    font-weight:bold;
}
```

__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.



About Component!

Mauris bibendum sollicitudin mi sit amet facilisis. Mauris augue quam, malesuada ut magna non, bibendum blandit libero.

__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. Copy the following file from **C:\LabFiles\advanced-routing** into the **\src\app** directory of your project:

```
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:

```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';
@Injectable()
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;
  }
}
```

__6. Open **\src\app\app.module.ts** in your text editor.

__7. Add the following import after the other imports at the top of the file:

```
import { AuthService } from './auth.service';
```

__8. Add AuthService to the providers array:

```
providers: [ AuthService  ],
```

__9. Save and close the file.

Task 3 - Add a login() function to the component

__10. Open **\src\app\app.component.ts** in your text editor.

__11. Add the following line after the other imports at the top of the file to import the AuthService:

```
import { AuthService } from './auth.service';
```

__12. Modify the constructor to inject the AuthService:

```
constructor( private auth: AuthService, private router: Router ){}
```

__13. Insert the following new method into the AppComponent class after the constructor:

```
  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);
    }
  }
```

__14. Save and close the file.

Task 4 - Add the guard to the route table

__15. Open **src\app\app-routing.module.ts** in your text editor.

__16. Add the following line after the other imports at the top of the file to import the AuthService:

```
import { AuthService } from './auth.service';
```

__17. 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] }
```

__18. Save and close the file.

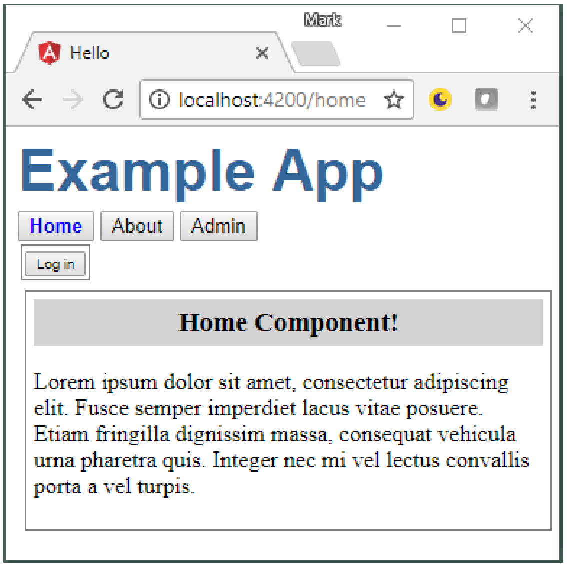
Task 5 - Test

__19. 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.

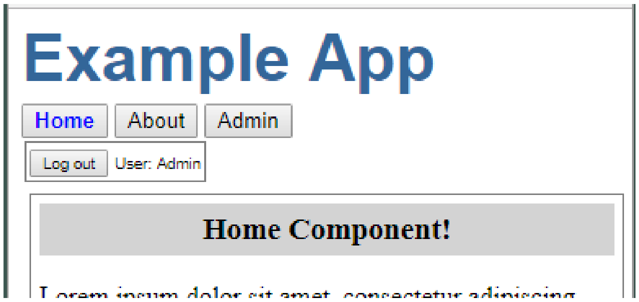
You should notice a "log in" button below the navigation buttons. Users need to click this button to log in as an administrator. Don't log in just yet. First we want to test how the app works when the user is Not logged in.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce semper imperdiet lacus vitae posuere. Etiam fringilla dignissim massa, consequat vehicula urna pharetra quis. Integer nec mi vel lectus convallis porta a vel turpis.

__20. Try clicking on the **Admin** button. Notice how the button is no longer active and no longer takes you to the admin screen.

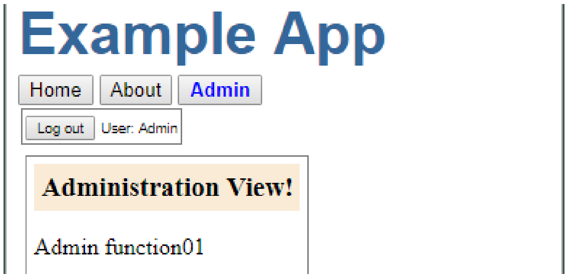
__21. Click on the "**Log in**" Button. The screen should change to indicate that you have logged in.



Home Component!

Lorem ipsum dolor sit amet, consectetur adipiscing

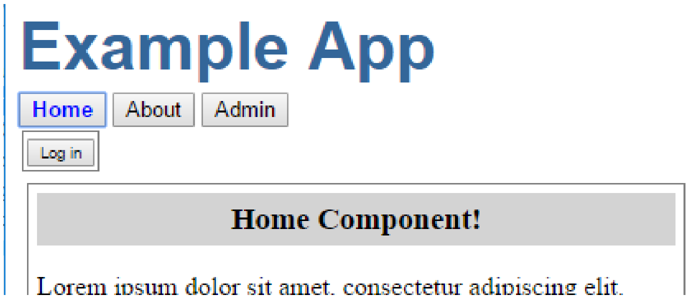
__22. Now try again to click on the **Admin** button. You should notice that the app now navigates as requested to show the Admin view.



Administration View!

Admin function01

__23. Click on the "**Log out**" Button. The application logs out the user and navigates away from the restricted page.



Home Component!

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

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. 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 the **\src\app\app.module.ts** file in your text editor.

__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** in your text editor.

__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.

Task 3 - Add a button to allow navigation to the Products component

__9. Open the **\src\app\app.component.html** file in your text editor.

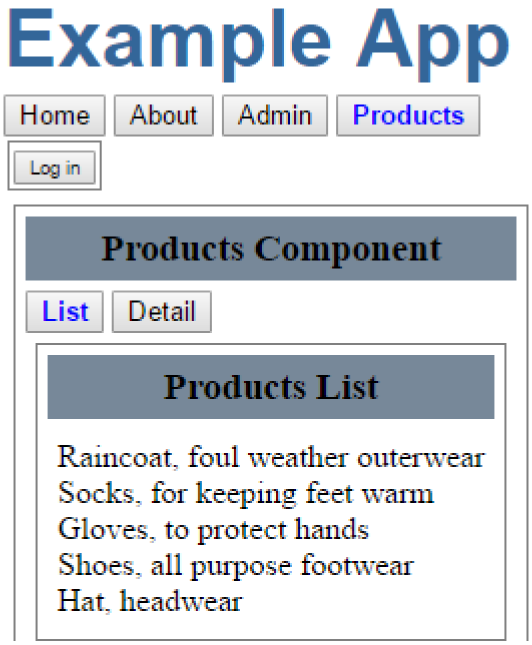
__10. Add the following products button element right after the admin button element:

```
    <input type=button [routerLink]="['/admin']"
        routerLinkActive="active" value="Admin" >
    <input type=button [routerLink]="['/products']"
        routerLinkActive="active" value="Products" >
</div>
```

__11. Save and close the file.

__12. Run the app in the browser and you will see a Products button in addition to the other navigation buttons.

__13. Click on the Products button, the ProductsComponent will appear. 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.



Raincoat, foul weather outerwear
Socks, for keeping feet warm
Gloves, to protect hands
Shoes, all purpose footwear
Hat, headwear

Task 4 - 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.

__14. 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' }
];
```

__15. 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.

__16. 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>
```

__17. 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.

__18. 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'.

__19. Close the browser.

__20. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__21. Close all open files.

Part 8 - 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 23 - Introduction to Unit Testing

Unit testing gives developers confidence that new changes have not broken parts of the application. Writing test scripts may take time away from actual development but it can be worth it.

In this lab we will develop a very simple application and write test scripts for it. The business logic will be very simple so that we can focus on the testing aspect.

Part 1 - Create the Project

__1. Open a command prompt window.

__2. Go to the **C:\LabWork\** folder.

__3. Run this command to create a project.

```
ng new basic-testing --defaults
```

A project created this way has support for testing. A minimal project created using --minimal does not have support for testing.

Part 2 - Create a Service

We will create a very simple service that will have a few synchronous and asynchronous methods. We will learn to unit test them later.

__1. From the **C:\LabWork\basic-testing** folder run.

```
ng g service simple
```

__2. Open **basic-testing/src/app/simple.service.ts**.

__3. Add this import statement.

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

__4. Add these methods to the service class.

```
sayHello(name:string) : string {
  return `Hello ${name}`
}

addNumbers(a:number, b:number) : Observable<number> {
  return of(a+b)
}
```

__5. Save changes.

We will now write test scripts for these methods.

Part 3 - Inject Service Instance

__1. Open **simple.service.spec.ts** file. Angular CLI had generated this file when the service class was created. We will write our test scripts here.

First thing we need to do is inject an instance of SimpleService class. There are several ways to do this. In fact the generated test script shows how to do this using the **inject()** function. We will use the simpler **TestBed.get()** method instead.

__2. Within the describe() arrow function, add a variable for the instance like this.

```
describe('SimpleService', () => {
  let service:SimpleService
...
}
```

__3. Write a new beforeEach() function to get an injected instance of DataService.

```
beforeEach(() => {
  service = TestBed.get(SimpleService)
});
```

Now that we have an instance of the service we can start calling its methods.

__4. Save changes.

Part 4 - Test Synchronous Method

We will first test the sayHello() method of SimpleService.

__1. Open **simple.service.spec.ts** file.

__2. Add this test script.

```
it("Should call sayHello", () => {
  let name = "Bob"
  let greeting = service.sayHello(name)

  expect(greeting).toBe(`Hello ${name}`)
});
```

__3. Save changes.

Part 5 - Remove AppComponent Tests

At the time of this writing there are some problems with the generated tests for AppComponent. We will just remove them.

__1. Open **app.component.spec.ts**.

__2. Delete all the **it()** function calls.

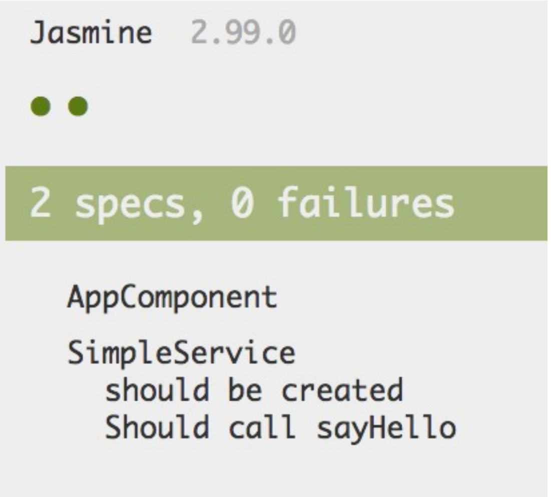
__3. Save changes and close the file.

Part 6 - Run the Test

__1. From the command line run:

```
npm test
```

This will build the application and run all tests. A new Chrome browser will window will open and show the test results.



```
Jasmine 2.99.0

● ●

2 specs, 0 failures

  AppComponent

  SimpleService
    should be created
    Should call sayHello
```

__2. Verify that the sayHello method has passed test.

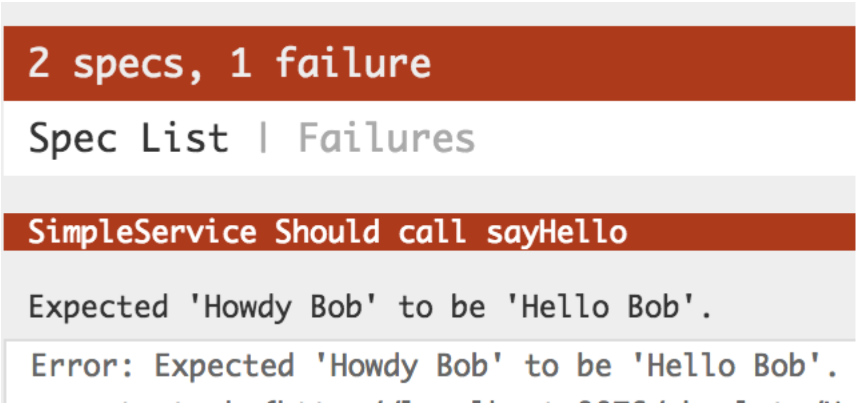
We will now deliberately introduce an error to see how a failed test looks like.

__3. Open **simple.service.ts** and introduce a defect in the sayHello() method like this.

```
sayHello(name:string) : string {
  return `Howdy ${name}`
}
```

__4. Save changes.

__5. The application should be rebuilt automatically and Karma should run your tests again.



```
2 specs, 1 failure
Spec List | Failures
SimpleService Should call sayHello
Expected 'Howdy Bob' to be 'Hello Bob'.
Error: Expected 'Howdy Bob' to be 'Hello Bob'.
```

__6. Verify that the test has failed.

__7. Fix the defect. Make sure that all tests pass after that.

Part 7 - Test Asynchronous Method

We will now test the **addNumbers()** method.

__1. Open **simple.service.spec.ts** file.

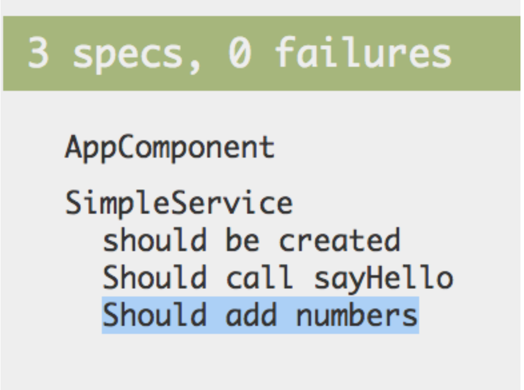
__2. Add this test script.

```
it("Should add numbers", (done) => {
  service.addNumbers(3, 4).subscribe((result) => {
    expect(result).toBe(7)
    done()
  })
});
```

The key thing here is the done() function. It is passed to the test script arrow function. We need to call it to indicate that a test has finished. Karma will wait until that point for a pass/fail result.

__3. Save changes.

__4. Make sure that the new test passes.

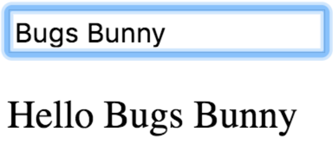


```
3 specs, 0 failures

  AppComponent

  SimpleService
    should be created
    Should call sayHello
    Should add numbers
```

__5. In the command prompt window, hit Control+C to stop the run.

Part 8 - Develop a Component

Our component will be very simple. It will have a text box for user to enter her name. The output from the SimpleService.sayHello() method will be shown in a <p> tag.



Bugs Bunny

Hello Bugs Bunny

__1. From the command line run this command to create the component.

```
ng g component greet
```

__2. Open **app/greet/greet.component.ts**.

__3. Add this import statement.

```
import { SimpleService } from '../simple.service';
```

__4. Inject the service instance from the constructor.

```
constructor(private service:SimpleService) { }
```

__5. Add a member variable to the GreetComponent which will be bound to the input text field.

```
userName:string = ""
```

__6. Finally add this method.

```
getGreeting() : string {
  return this.service.sayHello(this.userName)
}
```

__7. Save changes.

__8. Open the template file **greet.component.html**.

__9. Set the template to this.

```
<input type="text" [(ngModel)]="userName"/>
<p>{{getGreeting()}}</p>
```

__10. Save changes.

Part 9 - Test the Component

__1. Open **greet.component.spec.ts**.

Note that Angular CLI has already generated fairly useful code. Specifically, it obtains the GreetComponent and ComponentFixture prior to every test.

Our component depends on SimpleService. We must set it up as a provider in the test module. We also need to import the FormsModule so that we can use the ngModel directive in the template. We need to do these because the component will be tested as a part of the test module and not the application module.

__2. Add these import statements.

```
  import { SimpleService } from '../simple.service';
  import { FormsModule } from '@angular/forms';
```

__3. Add SimpleService to the list of providers of the test module. Also import FormsModule. This is shown in bold face below.

```
TestBed.configureTestingModule({
  declarations: [ GreetComponent ],
  imports: [FormsModule],
  providers: [SimpleService]
})
```

__4. Save changes.

__5. Write this test script.

```
it('Should get greeting', () => {
  component.userName = "Daffy Duck"

  expect(component.getGreeting()).toBe("Hello Daffy Duck")
});
```

__6. Save changes.

Part 10 - Run Test

__1. If the test runner is already running then end it by hitting Control+C. It doesn't always pick up newly created files. We will restart it.

__2. Start the test runner by entering this command.

```
npm test
```

__3. Verify that all GreetComponent tests pass.

> Note how we managed to test the component even before it is used in the application. This is actually by design. You can throughly test each component or service prior to actually using it in the application.
>
> The down side of this is that we have to configure every test module with the necessary providers and imports.

Part 11 - Test Generated DOM

A large part of component testing will be to verify that the DOM is generated correctly after a component state change. We will learn to do that now.

__1. Open **greet.component.spec.ts**.

__2. Add this test script.

```
it('Handle state change', () => {
  component.userName = "Bob"

  fixture.detectChanges();

  expect(fixture.nativeElement.querySelector('p').textContent)
    .toBe('Hello Bob');
});
```

Note a few things here:

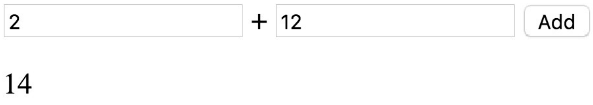
- The fixture.detectChanges() call is necessary so that Angular can react to a change in the component's internal state (modification of the userName field). By default, during testing, Angular does not automatically detect state changes.
- The querySelector('p').textContent call is pure DOM API. We can do this because fixture.nativeElement returns the DOM Element object for the component's root.

__3. Save changes.

__4. Verify that the "Handle state change" test passes.

Part 12 - Simulate User Interaction

We will now build a form that allows user to to enter two numbers. When the Add button is clicked we will call the SimpleService.addNumbers() method and display the result.



2 + 12

14

Testing this will require us to simulate a click on the Add button. First let's implement the add form feature in the component.

__1. Open **app/greet/greet.component.ts**.

__2. Add these member variables.

```
numberA = 0
numberB = 0
addResult = 0
```

__3. Add this method.

```
add() {
  this.service.addNumbers(this.numberA, this.numberB)
    .subscribe((result) => this.addResult = result)
}
```

__4. Save changes.

__5. Open the template file **greet.component.html**.

__6. Add this to the bottom of the template.

```
<input type="number" [(ngModel)]="numberA"/> +
<input type="number" [(ngModel)]="numberB"/>
<button (click)="add()">Add</button>
<div>{{addResult}}</div>
```

__7. Save changes.

Now we can test the component.

__8. Open **greet.component.spec.ts**.

__9. Add these import statements.

```
import { DebugElement } from '@angular/core';
import { By } from '@angular/platform-browser';
```

__10. Add this test script.

```
it('Adds numbers', () => {
  component.numberA = 10
  component.numberB = 20

  //Simulate button click
  let button:DebugElement =
    fixture.debugElement.query(By.css("button"))

  button.triggerEventHandler("click", null)

  fixture.detectChanges();

  expect(fixture.nativeElement.querySelector('div').textContent)
    .toBe('30');
});
```

__11. Save changes.

__12. Verify that the "Add numbers" test passes.

Part 13 - Put it All Together

Now that all our building blocks are unit tested we can assemble them in the application.

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

__2. Add this import statement.

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

__3. Add SimpleService to the list of providers. Also import FormsModule. This is shown in bold face below.

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

__4. Save changes.

__5. Open **app.component.html**.

__6. Change the template to be like this.

```
<app-greet></app-greet>
```

__7. Save changes.

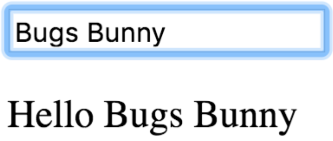
Part 14 - Run the Application

__1. Hit CTRL+C to stop the runner.

__2. Run the dev server.

```
npm start
```

__3. Open a browser and enter **http://localhost:4200**. Make sure you can use the application as intended.



Bugs Bunny

Hello Bugs Bunny

__4. Shut down the dev server.

__5. Close all open editors.

Part 15 - Review

You made it! There are obviously a LOT of things that may need to be considered when testing Angular components. By now you should have a good idea of the basics involved and of several common situations encountered in testing. There is more information on testing Angular components in the documentation on the Angular web site that can be used to supplement what you have learned here.

Lab 24 - 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. From **C:\LabWork\rest-client** start the development server.

```
npm start
```

__2. From **C:\LabFiles\rest-server** folder start the backend web service server.

```
npm start
```

__3. Open a browser and enter the URL: **http://localhost:4200** and make sure you can use the app normally.

__4. Open the browser developer tool (F12) and the Console pane. We will use the console a lot.

Part 2 - Simulate Typescript Compile Errors

__1. Open **C:/LabWork/rest-client/src/app/edit-book/edit-book.component.ts**.

__2. In the EditBookComponent class, add a variable with obvious compilation error.

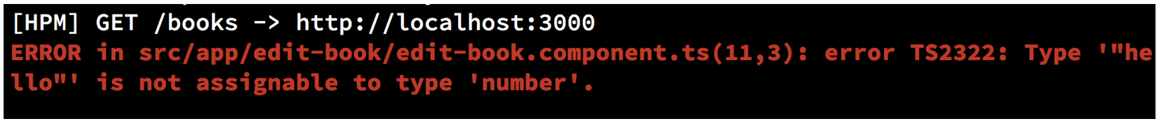
```
badVar:number = "hello";
```

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:



```
[HPM] GET /books -> http://localhost:3000
ERROR in src/app/edit-book/edit-book.component.ts(11,3): error TS2322: Type '"he
llo"' is not assignable to type 'number'.
```

__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

Syntax errors in the template are not reported by the dev server. They are discovered at runtime and reported in the browser console.

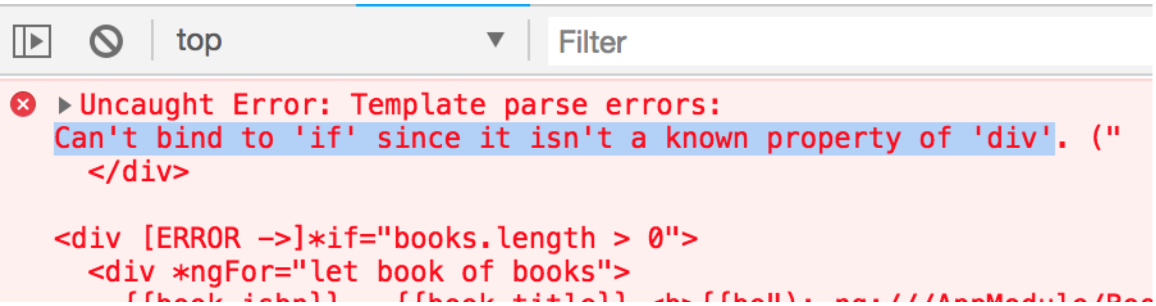
__1. Open **book-list/book-list.component.html**.

__2. Change the first ***ngIf** to ***if** (line 6).

__3. Save.

__4. Notice that the dev server compiles the code just fine.

__5. Back in the browser you will see a blank page. The console should show the actual error.



```
Uncaught Error: Template parse errors:
Can't bind to 'if' since it isn't a known property of 'div'. ("
  </div>

<div [ERROR ->]*if="books.length > 0">
  <div *ngFor="let book of books">
    [[book.isbn]] - [[book.title]] <b>[[hell"): ng:///AppModule/Boo
```

Important

During development always watch the dev server output as well as the browser console. Problems can be reported in either of these places.

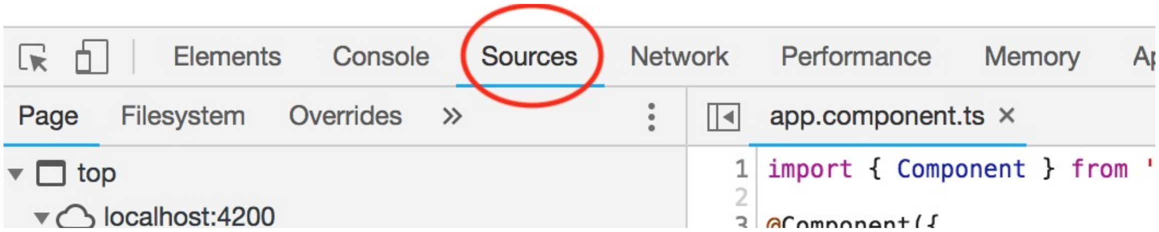
__6. Fix the problem (change **if** back to **ngIf**). Make sure that the app is working properly.

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.

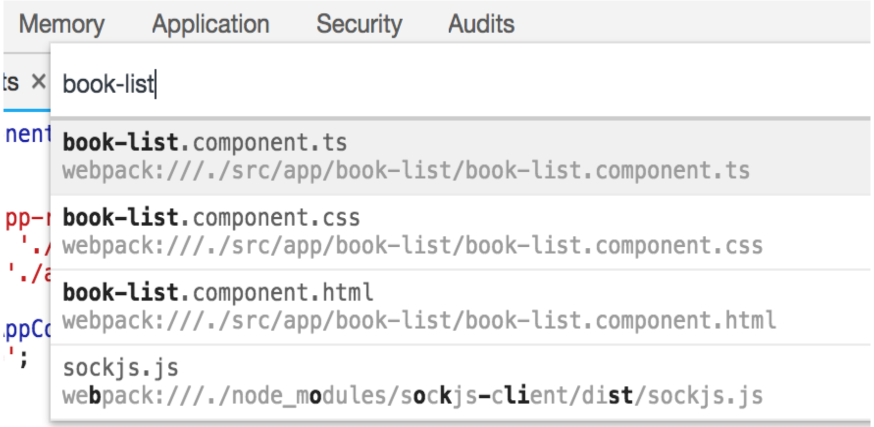
__1. Open **book-list/book-list.component.ts** in an editor. We will put a break point in the deleteBook() method.

__2. In the browser's developer tool, click the **Sources** tab.



```
import { Component } from '

@Component({
```

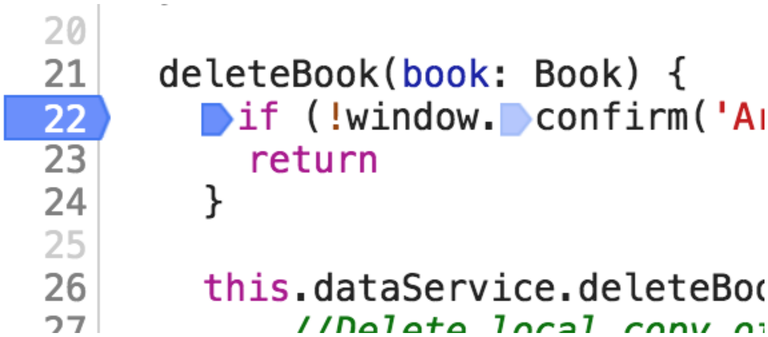
__3. To search for a file hit Control+P (Cmd+P in Mac). Start typing **book-list**.



```
book-list.component.ts
webpack:///./src/app/book-list/book-list.component.ts
book-list.component.css
webpack:///./src/app/book-list/book-list.component.css
book-list.component.html
webpack:///./src/app/book-list/book-list.component.html
sockjs.js
webpack:///./node_modules/sockjs-client/dist/sockjs.js
```

__4. Pick **book-list.component.ts**.

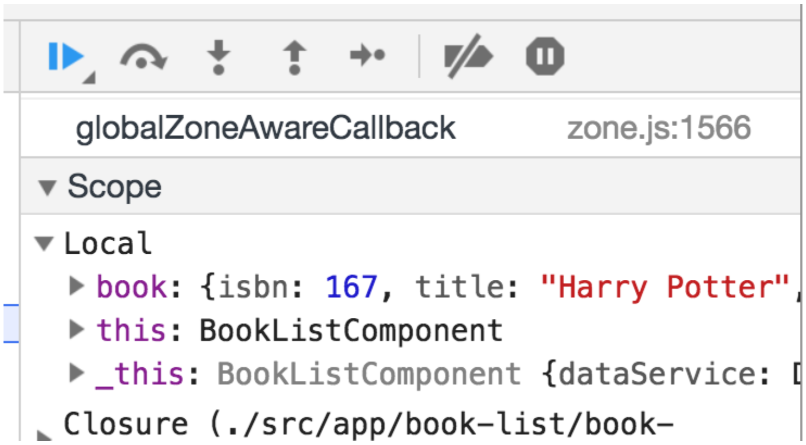
__5. Put a break point in the first line of deleteBook() method. (line numbers in your code may differ from those in the screenshot below)



```
deleteBook(book: Book) {
  if (!window.confirm('A
    return
  }

  this.dataService.deleteBo
      //Delete local copy o
```

__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.



```
globalZoneAwareCallback          zone.js:1566
▼ Scope
▼ Local
  ▶ book: {isbn: 167, title: "Harry Potter",
  ▶ this: BookListComponent
  ▶ _this: BookListComponent {dataService: D
▶ Closure (./src/app/book-list/book-
```

__8. Hit the resume button to continue. Click OK or Cancel.

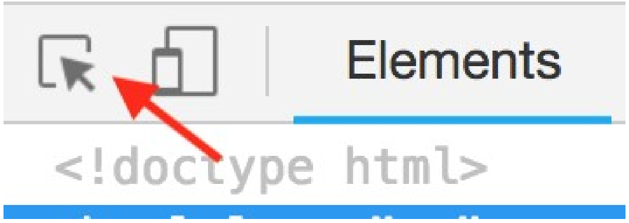
Part 5 - Use ng.probe

This is an easy way to inspect the internal state of a component without using a debugger. If you have many components on a page you can precisely select the one you wish to inspect.

__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. Click the Select button.



```
<!doctype html>
```

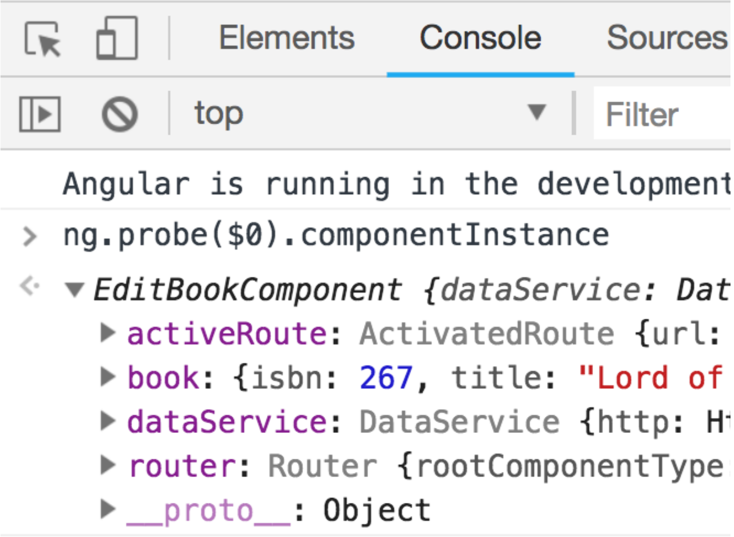
__4. Then click on one of the form input boxes. You can click anywhere within the boundary of the component.

__5. Switch back the **Console** tab in the developer tool.

__6. Type this and hit enter:

```
ng.probe($0).componentInstance
```

__7. You should now be able to expand the instance of EditBookComponent and look at its internal state (such as the **book** variable).



```
Angular is running in the development
> ng.probe($0).componentInstance
<· ▼ EditBookComponent {dataService: Dat
    ▶ activeRoute: ActivatedRoute {url:
    ▶ book: {isbn: 267, title: "Lord of
    ▶ dataService: DataService {http: H
    ▶ router: Router {rootComponentType
    ▶ __proto__: Object
```

__8. In both command prompt windows, hit Control+C.

__9. Close all.

Part 6 - Review

In the lab we learned a few common techniques to investigate errors in Angular applications.