Exercise Manual for Course 2324

Building Web Applications With Angular

2324/MA/D.1/907/C.3

by Mike Way

Legend for Course Icons

Standard icons are used in the hands-on exercises to illustrate various phases of each exercise.

👣	**Major step**	⚠	**Warning**
1. ❑	**Action**	💡	**Hint**
✓	**Checkpoint**	🛑	**Stop**
?	**Question**	🏆	**Congratulations**
ℹ	**Information**	🎗	**Bonus**
🧩	**Solution/Answer**		

Objectives

In this Do Now exercise, you will

- **Create a new Angular application as the starting point for the exercises in the course**

1. ❑ Return to the Course Command Prompt.

2. ❑ Press **`<Ctrl><C>`** to terminate the previous command.

3. ❑ Type **`Y <Enter>`** when prompted.

4. ❑ Type **`cd ..\Exercises <Enter>`**

5. ❑ Type **`ng new FlySharp<Enter>`**

6. ❑ When prompted with "Would you like to add Angular routing?", type **`y <enter>.`**

7. ❑ When prompted "Which stylesheet format would you like to use?", press the **`<enter>`** key.

   This step takes a couple of minutes to complete. If it freezes completely, please tell your instructor.

8. ❑ Type **`cd FlySharp <Enter>`**

9. ❑ Type **`ng serve -o <Enter>`**

   You should see the application open in Chrome at the URL: ***`http://localhost:4200`*** *You should see a default web page with links to some Angular documentation.*

Congratulations! You have completed the exercise.

This is the end of the exercise.

Objectives

In this exercise, you will

- **Explore the code generated by Angular-CLI**
- **Run unit tests on the code**
- **Execute the application**

Overview

In the Do Now exercise during the lecture session, you created an Angular application called FlySharp. In this exercise, you will explore the structure of the application and see some of the other capabilities of Angular-CLI.

Exploring the starter Angular application

1. ❑ If you have not already done so, log on to your exercise machine. The user name is **`student`** and the password is **`pw`**

2. ❑ If you did not complete the Do Now exercise ("Creating an Angular Application") in the lecture session, please do so before proceeding. Ask your instructor for any assistance you may need.

3. ❑ If the command prompt from the Do Now is still running, switch to it and press `<Ctrl><C>`.

   Occasionally, the `<Ctrl><C>` command does not work. If this happens, click in the title bar of the command window and try again.

4. ❑ From the Windows desktop, start JetBrains WebStorm.

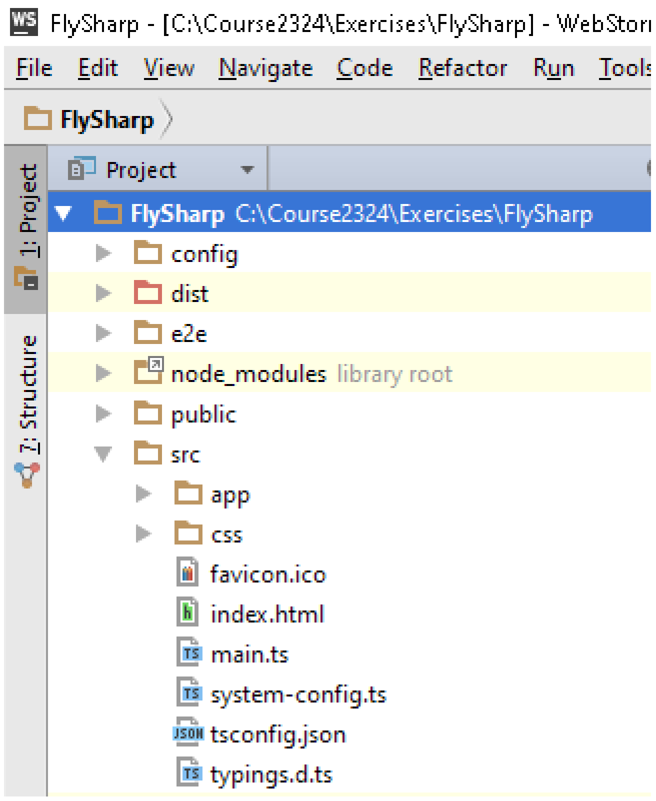
   We will refer to JetBrains WebStorm as WebStorm from here on.

5. ❑ In the "Welcome to WebStorm" window, select **Open**. Browse to `C:\Course2324\Exercises\FlySharp` and select **OK**.

   The first time WebStorm is opened on a new project, it takes a long time because it is indexing files. Please be patient.

6. ❑ Enable the Project view by selecting **View | Tool Windows | Project** from the WebStorm menu. (You can use the short-cut **`<Alt><1>`**)

7. ❑ In the Project view, expand FlySharp, then the **src** tree, then open `index.html`


FlySharp - [C:\Course2324\Exercises\FlySharp] - WebStor
File Edit View Navigate Code Refactor Run Tools
FlySharp
1: Project
7: Structure
Project
FlySharp C:\Course2324\Exercises\FlySharp
config
dist
e2e
node_modules library root
public
src
app
css
favicon.ico
index.html
main.ts
system-config.ts
tsconfig.json
typings.d.ts

8. ❑ Briefly examine the content of `index.html`.

The HTML page is part of the bootstrap of the Angular application. The key point is that there is an HTML element <`app-root`> where the initial component will load.

It may seem a little odd that there is no <`script`> block defined and this no JavaScript loaded. The <`script`> block is added dynamically by Angular-CLI when it builds the project.

9. ❑ Open `src\main.ts`. Examine the file.

Again, this is part of the bootstrap. You should see that it is loading the `AppModule` via the `bootstrapModule` call.

10. ❑ Across the top of the editor window, you may see a green bar and a grey bar each displaying a question.

    In the green bar (EditorConfig is overriding Code Style settings for this file), click **`OK`**. In the grey bar (Compile TypeScript to JavaScript), click **`No`**.

11. ❑ Expand the `src\app` directory.

    You should see six files:
    - *`app.component.ts`*
    - *`app.component.html`*
    - *`app.component.spec.ts`*
    - *`app.component.css`*
    - *`app.module.ts`*
    - *`app-routing.module.ts`*

12. ❑ Open `app.module.ts`.

    `AppComponent` is the starting `Component` for our application. How many times is `AppComponent` referenced in the `app.module.ts` file?

    ____

Check your work...

`AppComponent` is referenced 3 times!
- Once in the TypeScript import
- Once in the `declarations` section of the `@NgModule` metadata
- Once in the `bootstrap` section of the `@NgModule` metadata

It's the `bootstrap` entry that tells Angular to start by processing `AppComponent`.

13. ❑ Open `app.component.ts`.

Notice the three key sections:

- *An `import` statement that is importing the Angular core library*
- *The `@Component` decorator surrounding the metadata*
- *A class definition, which is where the code for the component will go*

14. ❑ Change the text of the `title` property to **`"Fly Sharp"`**

15. ❑ Press <`Ctrl`><S> to save all open files.

WebStorm actually saves files as soon as you click out of the active editor. The course development team just likes the reassurance of forcing a save by pressing <`Ctrl`><S>.

16. ❑ Open the WebStorm terminal by selecting **View | Tool Windows | Terminal** from the WebStorm menu.

17. ❑ In the terminal window, type **`ng serve -o`**, then press <`Enter`>.

This will start the development web server and compile any TypeScript files that are changed.

18. ❑ Wait for a browser to open.

After a few seconds, the browser should display your new message.

19. ❑ Switch back to the editor and open `app.component.html`

20. ❑ Replace the content of the file with the text below:

```
<h1> {{title}} </h1>
<p>best value airfares</p>
```

21. ❑ Save all files.

22. ❑ When you see the message `Compiled successfully.`, switch back to the browser.

You should see your change now displayed in the browser.

In the development mode, Angular keeps a Web socket connected to the server so it is notified if the code changes, and automatically reloads the application.

Running the unit tests

The `ng new` command generated a working unit test as part of the project creation.

23. ❑ In the WebStorm terminal window, press `<Ctrl><C>`, to stop the `ng serve` process.

24. ❑ In the WebStorm terminal window, run **`ng test`** `<Enter>`.

After about 30 seconds, you should see a browser that displays the Karma window.

Inspect the WebStorm Terminal window. Is there any red text indicating a test failure?
❑ *Yes* ❑ *No*

The error indicates a failing test, as the title string is not what was expected by the test. (You changed it a few minutes ago.)

25. ❑ Open `app.component.spec.ts`

26. ❑ In the test with the title `'should have as title 'FlySharp''`: change the text of the strings which are 'FlySharp`'` to '**`Fly Sharp'`** in the two places they appear.

27. ❑ If the test `'should render title in a h1 tag'` is failing: change `Welcome to FlySharp!` to **`Fly Sharp`** in the test.

28. ❑ Save the files.

The tests should now pass.

29. ❑ In the WebStorm Terminal window, press `<Ctrl><C>` and press **Y** to stop the `ng test` process.

Congratulations! You have completed the exercise.

If you have more time...

30. ❑ Investigate the other files that have been generated by Angular-CLI

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Create and compile some TypeScript code**
- **See the benefit of strong typing**
- **Enable automatic compilation**

Overview

In this exercise, you will write some TypeScript code to manipulate a collection of flights.

Creating a `TypeScript` class

1. ❑ In WebStorm, select **File | Open** and browse to `C:\Course2324\Exercises\Ex1.2`. Click **OK**.

2. ❑ When prompted, select **This Window**.

3. ❑ Open and inspect `index.html`

   It's a simple page to display the details of a flight and the total of all the flights in a shopping cart.

4. ❑ Right-click `index.html` and select **Open in Browse**r. Select your preferred browser.

   At this stage, there will be no flight data to see, just some placeholder text.

5. ❑ Switch back to the editor, and just after the closing body tag, add a **`script`** tag referencing **`scripts/flights.js`**

Hint...

```
<script src="scripts/flights.js"></script>
```

It's a `.js` file that is loaded by the browser, not the `.ts` file.

Creating some flight data

6. ❑ Open `scripts/flights.ts`

7. ❑ At the top of the file, create an array called **`MYFLIGHTS`** with two elements initialized using OLN.

   The values should be:

```
    { "flightNumber": "FS1298", "origin": "LAX",
"destination": "LHR", "price": 99.99 },
     { "flightNumber": "FS1201", "origin": "LAX",
"destination": "LHR", "price": 199.99 }
```

Hint...

```
let MYFLIGHTS = [
    { "flightNumber": "FS1298", "origin": "LAX",
"destination": "LHR", "price": 99.99 },
    { "flightNumber": "FS1201", "origin": "LAX",
"destination": "LHR", "price": 199.99 }
];
```

Creating a class

8. ❑ At the comment `TODO 2`, create a class called **`FlightInfo`**

Hint...

```
class FlightInfo {

}
```

9. ❑ In the class, create a method called **`getFlight`** that returns `MYFLIGHTS[0]`

Hint...

```
getFlight() {
    return MYFLIGHTS[0];
}
```

10. ❑ In the class, create a second method called **`getTotalPrice()`** that returns a `number`. In the body of this method, iterate through `MYFLIGHTS`, add up the total price, and return the total.

Hint...

```
        getTotalPrice(): number {
            let total = 0.0;
            for (let flight of MYFLIGHTS) {
                total += flight.price;
            }
            return total;
        }
```

11. ❑ Below the comment "`TODO get a flight`" (outside of the class definition):
    - Create a new **`FlightInfo`** object and assign it to the variable `flightInfo`
    - Call the **`getFlight()`** method on `flightInfo` and assign the result to a variable called **`theFlight`**

Hint...

```
class FlightInfo {
    // TODO get a flight
    getFlight() {
        return MYFLIGHTS[0];
    }

}
let flightInfo: FlightInfo = new FlightInfo();
let theFlight=flightInfo.getFlight();
```

12. ❑ Below the comment `"TODO examine code"`:
    - Briefly examine the four lines of code that display the flight data and the total price

13. ❑ Open the WebStorm terminal and change the directory to **`scripts`** with the following command:

    ```
    cd scripts
    ```

14. ❑ Run the TypeScript compiler with the command **`tsc -W flights.ts`**

    The `-W` flag causes the compiler to watch for changes to the `.ts` file and recompile if necessary.

15. ❑ Resolve any errors reported by the compiler, and re-save your file (`<Ctrl><S>` or click in another window).

16. ❑ In the Project view of WebStorm, locate the `flights.js` file (under `flights.ts`) and open it. Spend a few moments examining the code.

    The code is very similar to the TypeScript you created, but look carefully at the code that has been used to define the `FlightInfo` class. This has been translated to a JavaScript variable using the JavaScript prototype to define the methods of the class as functions.

17. ❑ Return to the html page in your browser and refresh it. You should see that the flight values are now in your page.

    What is the total price of the flights?

    ____

Exploring the benefits of static types

18. ❑ Switch back to `flights.ts`

19. ❑ In the declaration of `MYFLIGHTS`, change the price values from number to string by surrounding the price data with quotes:

    `"price": "99.99"`

    Make this change in both lines.

    This is an easy mistake to make!

20. ❑ Save the file, ignoring any errors from the TypeScript window—the JavaScript is still generated.

21. ❑ Refresh the page in the browser.

    What is the total price now?

    ____

    Adding two strings is very different than adding two numbers!

22. ❑ At `TODO 3`, define an `interface` called **`Flight`** that declare these types:

```
flightNumber: string
origin: string
destination: string
price: number
```

Hint...

```
interface Flight {
    flightNumber: string;
    origin: string;
    destination: string;
    price: number;
}
```

23. ❑ Modify the declaration of `MYFLIGHTS` to specify that the type is an array of `Flight`

Hint...

```
let MYFLIGHTS : Flight[] = [...
```

You should see that WebStorm editor is now complaining about the error! The transpiler output in the terminal window should also show an error.

24. ❑ Refresh the `index.html` page in the browser.

Nothing has changed!

The TypeScript transpiler still generates output even when errors are detected! You should check the transpiler output.

25. ❑ Remove the quotes you added around the price.

This should fix the transpilation error.

26. ❑ Refresh the browser.

The total should be calculated correctly.

27. ❑ You have seen how using types provides information in the transpiler output about possible errors. This can make for more robust code—but only if we check the transpiler output!

Congratulations! You have built and tested a class using TypeScript and strong types.

If you have more time...

28. ❑ Modify the `getFlight()` method to specify that the return type is a `Flight` object. While this is not needed for the code to run, it makes the return strictly typed.

Hint...

```
getFlight(): Flight {
   return MYFLIGHTS[0];
}
```

STOP

This is the end of the exercise.

Objectives

In this Do Now exercise, you will

- **Integrate multiple components**

1. ❑ In WebStorm, open the `DoNow21` project from `C:\Course2324\DoNows\DoNow21`

2. ❑ Open `src\app\app.component.html`

3. ❑ Add this element to the end of the file:
   `<app-forecast></app-forecast>`

4. ❑ Open `src\app\app.module.ts` and add **`ForecastComponent`** to the `declarations` section of the `@NgModule` metadata with the value **`ForecastComponent`**

5. ❑ Add an **`import`** of **`ForecastComponent`** from **`'./forecast/forecast.component'`** (Your IDE may have done this automatically for you).

6. ❑ Switch to the Terminal window of WebStorm.

7. ❑ Run **`ng serve`**

8. ❑ Open **`http://localhost:4200`** in your browser.

9. ❑ You should see the text "`There is Weather is expected in ...`" from the ForecastComponent.

Congratulations! You have completed the exercise.

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Create an Angular component**
- **Integrate the component with a parent component**
- **Add CSS styling using Bootstrap (in the bonus exercise).**

Overview

In this exercise, you will create an Angular component for the Home tab of our application and integrate with the application previously generated.

Creating a new component

1. ❑ Return to the project you previously created in `C:\Course2324\Exercises\FlySharp` in WebStorm.

In the next step, be sure to name the home directory all lowercase. Everything is case sensitive, including the folder names.

2. ❑ Create a new directory called **home** under the `src\app` directory (`C:\Course2324\Exercises\FlySharp\src\app\home`). You can do this from within the WebStorm Project view by right-clicking **app** and selecting **New | Directory**. Be sure to name the directory all lowercase `home`.

This will be where you create the new component.

3. ❑ In the `home` directory, create a new file called **`home.component.ts`** (right-click `home` and select **New | File**).

4. ❑ If the new file has not already opened, then open it in the editor.

5. ❑ If prompted by WebStorm to compile TypeScript to JavaScript, select **No**. This will be done by Angular CLI; we do not need WebStorm to also compile it.

6. ❑ In the file, add the following:
   - An `import` of `Component` from `@angular/core`
   - A **`@Component`** declaration—leave it empty for the moment
   - An exported class definition for a class called **`HomeComponent`**—leave the class empty for now.

Hint...

The existing `app.component.ts` file may be useful as a template for you.

7. ❑ In the `@Component` section, add the following:
   - **`selector: 'app-home'`**
   - **``template: `<h1>Special Offer of the month {{specialOffer}}</h1>` ``**

If the template extends over more than one line, be careful to use the back-tick (` )characters around the HTML of the template. Make sure the template property is `template` and not `templateUrl`

8. ❑ Inside the class declaration, declare a field called **`specialOffer`** and initialize it to **`"10% off all round-the-World flights"`**

Hint...

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-home',
  template: `<h1>Special Offer of the month
{{specialOffer}}</h1>`,
})
export class HomeComponent {
  specialOffer="10% off all round-the-World
flights";
}
```

9. ❑ Save all open files by selecting **File | Save All** or pressing <`Ctrl`><`S`>.

WebStorm should automatically save files, but on occasion, this has failed to trigger the transpilation process.

Adding your component to the existing application

10. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\app.component.html`

11. ❑ Replace the content of the file with this:

```
<h1>
  {{title}}
</h1>
```

The `{{title}}` displays the title property of the component. We will discuss how this works in the next section.

12. ❑ Below the `<h1>...</h1>` element, add a new tag: **`<app-home></app-home>`**

You do need the full form of this element, not `<app-home/>`

Hint...

```
<h1>
  {{title}}
</h1>
<app-home></app-home>
```

13. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\app.module.ts`

14. ❑ Add **`HomeComponent`** to the `declarations` section of the `@NgModule` metadata.

Hint...

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

15. ❑ Add an import for the `HomeComponent`

In WebStorm, select the `HomeComponent` and press <`Alt`><`Enter`> to add the import. (Your IDE may have added this import automatically)

Hint...

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

16. ❑ Save all open files.

17. ❑ In the WebStorm terminal, make sure that `ng serve` is running.

If you get an error indicating that a module cannot be found, run `npm install` to update the installed modules. Please tell your instructor.

18. ❑ Open `http://localhost:4200/` in a browser.

You should see the special-offer message displayed.

If you did not see the message, the two key things to check are that:

- **You added the `<app-home></app-home>` into `app.component.html`**
- **You added the directive to `app.components.ts`**

Congratulations! You have completed the exercise.

If you have more time, add Bootstrap CSS styling to the application.

19. ❑ In the WebStorm terminal window press **`<Ctrl><C>`** and press **`Y`** to stop the `ng serve` process and run the command **`cpAddIns Ex2.1`**

`cpAddIns` is a NodeJS script we have created to copy files into the exercise directory.

20. ❑ You should now see a `css` directory under the `src/assets` directory.

21. ❑ Add import statements into `src/styles.css` to load the CSS. They should be loaded in this order:
    1. **`bootstrap`**
    2. **`styles`**

CSS imports should be of this form:

```
@import './assets/css/bootstrap.css';
@import './assets/css/styles.css';
```

22. ❑ Add a CSS class of **`container`** to the `<app-root></app-root>` element in `index.html.`

23. ❑ Save and test your work.

✓ *You should see some small change to the appearance of the application. This will become much more significant as we add a header and tabs to the application later in the course.*

Congratulations! You have added Bootstrap CSS styling to the application.

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Incorporate model data with a template using interpolation**
- **Display a list of data with `*ngFor`**
- **Conditionally display data with `*ngIf`**

Overview

We have provided a list of flight data for you. You will use interpolation and other template commands to display this data in the `BuyFlightComponent`.

1. ❑ In WebStorm, make sure the `C:\Course2324\Exercises\FlySharp` project is open.

2. ❑ In the WebStorm terminal, press <`Ctrl`><C> then <Y> to stop `ng serve`

   This next step will copy the solution from Hands-On Exercise 2.1 (including any bonus steps) onto your project to provide a consistent starting point. Although this should be very similar to your code at the end of the last exercise, we strongly recommend that you do perform this step.

3. ❑ In the terminal, run the command **`exStart Ex2.2`**

4. ❑ In the terminal, run the command **`ng generate component BuyFlight`**

   This Angular-cli command creates a new component called `BuyFlightComponent`

5. ❑ Open the file `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.ts`.

   We have created some dummy flight data for you as a starter for the exercise. In the next step, you will copy it into the `BuyFlightComponent`

6. ❑ Run **`cpAddIns Ex2.2`** in the WebStorm terminal window.

   The `cpAddIns Ex2.2` command also copied in a new version of `app.component.html` that has the HTML structure to provide the application tabs.

7. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\flights.txt`. Copy the entire content of the file and paste it at the very end of `buy-flight.component.ts`

   If the `flights.txt` file is not visible, try refreshing the project.

8. ❑ In `buy-flight.component.ts`, inside the class definition, create a field called **`flights`** and assign it the value **`FLIGHTS`**

Hint...

```
flights = FLIGHTS;
```

9. ❑ Save all files.

10. ❑ Open the file `src\app\buy-flight\buy-flight.component.html`.

11. ❑ Delete all of the existing content.

12. ❑ Create an HTML table, with a single row and eight columns.

Hint...

```
<table>
<tbody>
  <tr >
      <td></td>
      <td></td>
      <td></td>
      <td></td>
      <td></td>
      <td></td>
      <td></td>
      <td></td>
  </tr>
</tbody>
</table>
```

13. ❑ Add an `*ngFor` directive to the `tr` element to make each flight in the `flights` field available as a variable called **`flight`**

Hint...

```
<tr *ngFor="let flight of flights">
```

14. ❑ Using the interpolation syntax, populate the `<td>` elements with code to display the flight details in the following order:

- `flightNumber`
- `origin`
- `destination`
- `departDay`
- `departTime`
- `arriveDay`
- `arriveTime`
- `price`

Hint...

```
<td>{{flight.flightNumber}}</td>
<td>{{flight.origin}}</td>
<td>{{flight.destination}}</td>
<td>{{flight.departDay}}</td>
<td>{{flight.departTime}}</td>
<td>{{flight.arriveDay}}</td>
<td>{{flight.arriveTime}}</td>
<td>{{flight.price}}</td>
```

15. ❑ Save all files.

Adding the new component to the application

16. ❑ In WebStorm, switch back to `buy-flight.component.ts`

What is the value of the selector?

____

Hint...

```
selector: 'app-buy-flight'
```

17. ❑ Open `app.component.html.`

18. ❑ Near the end of the file, just *before* the closing `</main>` tag, add an HTML element matching the selector for `BuyFlightComponent`

Hint...

```
<main class="container-fluid">
  <app-home></app-home>
  <app-buy-flight></app-buy-flight>
</main>   <!-- this line is already there, don't
type </main> -->
```

19. ❑ Open `app.module.ts` and verify that **`BuyFlightComponent`** has been added to the `declarations` array.

20. ❑ Save open files.

21. ❑ Run **`ng serve -o`** in the WebStorm terminal.

22. ❑ Switch to the browser and load `http://localhost:4200/`

You should see a list of flights.

The list is pretty ugly, but we will tidy that up in the bonus.

Conditional display of the flight list

*Eventually, we only want to display the list of flights when the `buyFlights` button is pressed. For now, we will display the list of flights based on whether a field in the component called **`showBuyFlights`** is set to true.*

23. ❑ In `buy-flight.component.ts`, add a field **`showBuyFlights`** to the class and initialize it to **`false`**

Hint...

```
showBuyFlights = false;
```

24. ❑ In `buy-flight.component.html`, add an **`*ngIf`** attribute with a value of **`showBuyFlights`** to the `<table>` element.

Hint...

```
<table *ngIf="showBuyFlights">
```

25. ❑ Save the file and switch back to the browser.

You should no longer see the list of flights.

26. ❑ In `buy-flight.component.ts`, change the value of `showBuyFlights` to be **`true`**

27. ❑ Save the file and switch back to the browser.

You should see the flights.

Toggling the display of the flight list

28. ❑ Change the value of `showBuyFlights` to **`false`** again. Save the file.

29. ❑ At the end of `buy-flight.component.html`, add an **<a>** tag with the content **`Toggle Flights.`** The `href` attribute should have the value '#'.

Hint...

```
<a href='#'>Toggle Flights</a>
```

30. ❑ Add this click handler to the <a> tag:

```
(click)="showBuyFlights = !showBuyFlights"
```

31. ❑ Don't try to use `showBuyFlights != showBuyFlights`. It's a test, not an assignment!

The <a> start tag should look like `<a (click)="showBuyFlights = !showBuyFlights" href='#'>`

32. ❑ Test your work.

The table should show and hide as you click the `Toggle Flights` link.

What we really wanted to do here was to get the flights list to toggle when the Buy Flights button in the Tab bar was clicked. The problem is that the Tab bar is in a different component, and we have not yet discussed how to achieve component communication. We'll come back to that later.

Congratulations! You have completed the exercise.

If you have more time...

The previous part had you modify a property of a class directly to control if the flights were shown. It is better to call a method that sets the property instead. In this bonus, you will modify the click handler to call a method on the class, which then changes the state of the variable controlling the display of the flights.

33. ❑ Modify the click handler on the <a> created earlier to look like:

```
(click)="onClickBuyFlights()"
```

34. ❑ Create a new method in the `BuyFlightComponent` class called `onClickBuyFlights()`:

```
onClickBuyFlights() {
}
```

35. ❑ Add the code in the method just created to manipulate the state of the `showBuyFlights` variable to toggle its value.

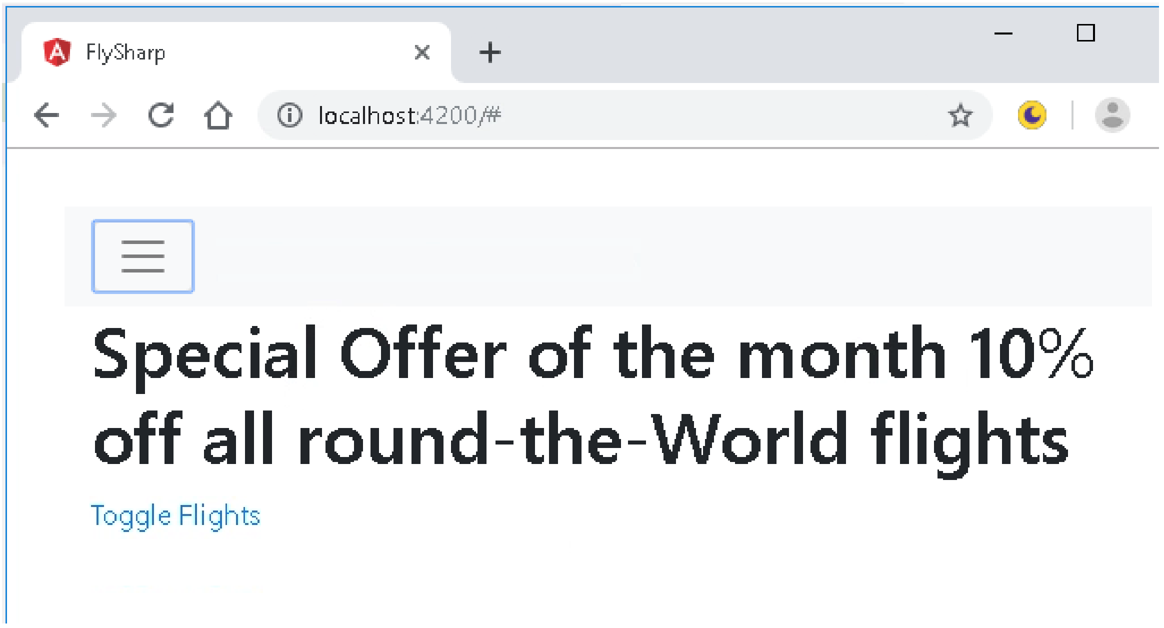
Hint, the method should look like...

```
onClickBuyFlights(){
   this.showBuyFlights = !this.showBuyFlights;
 }
```

36. ❑ Test the app in a browser. It should work just like before, but is now using a method to set the property.

If you have more time still...Enable the Responsive Menu Button

37. ❑ Reduce the width of the browser window until the menu tabs disappear and a menu button (burger bar) appears.



Special Offer of the month 10% off all round-the-World flights

Toggle Flights

38. ❑ Click on the menu button.

Was the menu displayed when you clicked the button?
❑ *Yes* ❑ *No*

To make the menu button work, we need to dynamically add a class to the <`div`> element containing the collapsed menu. We can do this conveniently using the `[ngClass]` binding.

39. ❑ In `app.component.html`, modify the <div> element near line 6 by adding an **`ngClass`** binding with the value **`"{ 'show': navbarOpen }"`**

The modified <div> element...

```
<div class="collapse navbar-collapse"
[ngClass]="{ 'show': navbarOpen }"
id="navbarSupportedContent">
```

40. ❑ This binding will add the class `'show'` to the `<div>` if `navbarOpen` is `true`.

41. ❑ Near line 2, add a **`click`** event handler binding which calls the method **`toggleNavbar()`** to the `<button>` element.

42. ❑ In `app.component.ts`:
    - Add a new field **`navbarOpen`** with an initial value of **`false`**
    - Add a method **`toggleNavbar()`** which toggles the value of `navbarOpen` between `true` and `false` when it is called

`toggleNavbar()` code:

```
toggleNavbar() {
  this.navbarOpen = !this.navbarOpen;
}
```

43. ❑ Test your work:
    - Save any open files
    - Run the application
    - If necessary, reduce the width of the browser so the menu button is displayed
    - Click the button to check that the menu is displayed
    - Click again to check that the menu is hidden

If you have even more time, improve the appearance of the table with some CSS

44. ❑ Add a table header row to the table with the column headings.

45. ❑ Improve the appearance of the flights table by adding these CSS classes to the `<table>` element:
    - `table`
    - `table-condensed`
    - `table-responsive`

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Define a service**
- **Use dependency injection to make a flight service available to the components**

Overview

Our application will eventually fetch flight data from a server-side application. In this exercise, you will create an initial version of the service class that will communicate with the server. For the moment, it will just return some mock data.

Creating a flight service

1. ❑ In WebStorm, make sure the project at `C:\course2324\Exercises\FlySharp` is open.

2. ❑ From the WebStorm terminal, press <`Ctrl`><`C`>, press <`Y`>, and then press <`Enter`> to stop `ng serve`.

3. ❑ From the WebStorm terminal window, run the command **`exStart Ex3.1`**

4. ❑ Select **Yes** if asked to reload the project

5. ❑ In the WebStorm terminal, run the command **`ng generate service --flat=false Flights`**

6. ❑ Examine `C:\Course2324\Exercises\FlySharp\src\app\flights` to see the generated code.

   You should see two files: `flights.service.ts`, which is the service code, and `flights.service.spec.ts`, which is the unit test.

7. ❑ Open `flights.service.ts` in the editor.

   What is the name of the generated class?

   ____

   What decorator has been inserted prior to the class declaration?

   ____

   The `providedIn: 'root'` property of `@Injectable` causes Angular to make the service available to the dependency injector everywhere in the application

8. ❑ From the WebStorm terminal, run **`cpAddIns Ex3.1`** to populate the `app\model` directory with some mock data and an interface to represent a `Flight`.

9. ❑ Open the file `C:\Course2324\Exercises\FlySharp\src\app\model \mock-flights.ts`

   We have created two arrays of flights to act as mock data.

10. ❑ Return to the `flight.service.ts` file.

11. ❑ Create a **`public`** method **`getFlights()`** that returns the mock data **`FLIGHTS`**. It should be declared to **`return Flight[]`**

Hint...

```
public getFlights() : Flight[]{
    return FLIGHTS;
 }
```

12. ❑ Use `<Alt><Enter>` to resolve any imports (click the items in red and press `<Alt><Enter>`).

Hint: The imports should now look like...

```
import {Flight} from "../model/flight";
import {FLIGHTS} from "../model/mock-flights";
```

13. ❑ Create a second public method called **`getMyFlights()`** that returns **`MYFLIGHTS.`** Use `<Alt><Enter>` to resolve any missing inputs.

Hint...

The method should look like:

```
public getMyFlights() : Flight[]{
    return MYFLIGHTS;
}
```

and the imports are now:

```
import {Flight} from "../model/flight";
import {FLIGHTS, MYFLIGHTS} from "../model/mock-
flights";
```

14. ❑ Save all files.

Modifying the `BuyFlights` component to load the flights from the service

15. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\buy-flight \buy-flight.component.ts`.

16. ❑ Edit the constructor to take `FlightsService` as an argument called **`flightsService`**. Mark the argument as **`private`**

Hint...

```
constructor(private flightsService :
FlightsService ){}
```

17. ❑ Add any required imports by pressing **`<Alt><Enter>`**.

18. ❑ Verify that the class declaration for `BuyFlightComponent` implements the **`onInit`** lifecycle interface.

19. ❑ Near line 12, change the initial value of `showBuyFlights` to **`true`**;

Hint...

```
export class BuyFlightComponent implements OnInit
{
```

20. ❑ Locate the empty `ngOnInit()` method. In the body of the `ngOnInit()` method, add code to call the **`getFlights()`** method on **`this.flightsService`**, and assign the results to **`this.flights`**

Hint...

```
ngOnInit() {
    this.flights =
this.flightsService.getFlights();
  }
```

21. ❑ Modify the declaration of the `flights` field so that it does not initialize the flights data. It should instead declare the field to be of type **`Flight[]`**

Hint...

```
flights: Flight[];
```

22. ❑ Use `<Alt><Enter>` to add the **`import`** of the **`Flight`** type:

    `import { Flight } from '../model/flight';`

23. ❑ Delete the older `FLIGHTS` array used earlier, which is defined at the end of the file.

24. ❑ Save all files.

25. ❑ In the WebStorm terminal, run **`ng serve -o`**

26. ❑ Open a browser and test your work.

    You should see a list of five flights that have been loaded from the service.

Congratulations! You have completed the exercise.

STOP

This is the end of the exercise.

Objectives

In this Do Now exercise, you will

- **Execute a test using Karma**

1. ❑ Open the **DoNow41** project at `C:\Course2324\DoNows\DoNow41` in WebStorm.

2. ❑ Open the WebStorm terminal window (<Alt><F12>).

3. ❑ Run **`ng test`**

   After about a minute, you should see a browser launch running Karma.

4. ❑ You should then see a message in the terminal window indicating that the tests have passed.

5. ❑ Open `app.component.ts` and change the value of the `title` property to **`broken`**

6. ❑ Save the file.

   After a few seconds, you should see a message indicating a test failure.

Congratulations! You have completed the exercise.

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Develop a Jasmine unit test for a component**
- **Inject a dependency into the object under test**
- **Mock a dependency (in the bonus)**

Overview

Unit testing is an important part of the development process, particularly as applications become more complex. In this exercise, you will create a Jasmine unit test for the `BuyFlightComponent`.

Enhancing the Jasmine test created by Angular-cli

1. ❑ In WebStorm, make sure the project at `C:\course2324\Exercises\FlySharp` is open.

2. ❑ From the WebStorm terminal window, run the command **`exStart Ex4.1`**

3. ❑ Press **Yes** if asked to reload the project.

Running the tests as they stand

4. ❑ In the WebStorm terminal window, run **`ng test`**

You should see Chrome being used to run the tests. After a minute or so, you should see a message indicating that 6 tests have passed with 0 failures. If you examine the output in the browser, you should see that tests have run against `AppComponent`, `BuyFlightComponent` and the `FlightsService`.

Break the tests!

The BuyFlightComponent depends on FlightsService being available in the dependency injector. This dependency is currently satisfied by the decorator for the FlightsService class:

```
@Injectable({
 providedIn: 'root'
})
```

Which causes the FlightService to always be available to the dependency injector. If one of the alternative injection strategies is used then we have to provide the dependency directly in the test.

5. ❑ Open `c:\course2324\Exercises\FlySharp\src\app\flights\flights.service.ts`. Remove the `providedIn` property from the `@Injectable` annotation.

The modified `@Injectable` decorator.

```
@Injectable()
```

6. ❑ Save the file.

This will trigger the test to run again. You should now see that some tests fail. If you examine the errors you will see something like this:

```
NullInjectorError: No provider for FlightsService!
```

Providing the dependency

7. ❑ Open `C:\course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.spec.ts`

8. ❑ Add a providers declaration to the TestBed configuration, specifying `FlightsService` as a provider.

Hint...

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ BuyFlightComponent ],
    providers: [FlightsService],
  })
  .compileComponents();
}));
```

9. ❑ Save the file.

   5 of the 6 tests should now pass. The failing test is for the `FlightsService`.

10. ❑ Open `c:\course2324\Exercises\FlySharp\src\app\flights\flights.service.spec.ts` and add a `providers` property for `FlightsService` to the TestBed as you did for `buy-flight.component.spec.ts`

Code to add the provider to the FlightsService test:

```
beforeEach(() => TestBed.configureTestingModule({
    providers: [FlightsService], }
));
```

All of the tests should now pass.

Modifying the `@Injectable` decorator has actually caused another issue which the unit tests do not detect (but the end-to-end tests would): we don't have a correctly defined `provider` for the `FlightsService`. You will now fix this issue:

11. ❑ Open `c:\course2324\Exercises\FlySharp\src\app\app.module.ts`. Add **`FlightsService`** to the `providers` array in `@NgModule`

Creating additional tests

12. ❑ Working in `c:\course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.spec.ts`

13. ❑ Immediately after the "describe" line (near line 8) and before the "it" test, add an additional variable declaration:
`let el: DebugElement;`

14. ❑ Duplicate the `it(...)` test and change the description to **`should default showBuyFlights to true`**

15. ❑ In the body of the new test, check that the value of `showBuyFlights` on the component under test is **`true`**

Hint...

```
it('should default showBuyFlights to true', () => {
  expect(component.showBuyFlights).toBeTruthy();
});
```

16. ❑ Save the files.

The tests should automatically run again. Make sure they all pass.

17. ❑ Create an additional test to check the behavior of `onClickBuyFlights()`. The test should:
    - Be named **`"should set showBuyFlights to false when onClickBuyFlights() is called"`**
    - Call **`onClickBuyFlights()`**
    - Expect that **`showBuyFlights`** will be `false`

Hint...

```
  it('should set showBuyFlights to false when
onClickBuyFlights() is called', () => {
    component.onClickBuyFlights();
    expect(component.showBuyFlights).toBeFalsy();
  });
```

18. ❑ Add another test to check the behavior when the **`onClickBuyFlights()`** method is called twice (it should set `showBuyFlights` to `true` again).

Hint...

```
  it('should set showBuyFlights to false when
onClickBuyFlights() is called', () => {
    component.onClickBuyFlights();
    component.onClickBuyFlights();
    expect(component.showBuyFlights).toBeTruthy();
  });
```

✓ *Check that the tests still work.*

19. ❑ Open `C:\course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.ts` and comment out the line **`this.showBuyFlights = !this.showBuyFlights;`** (near line 20).

20. ❑ Save open files.

✓ *After a few moments, the tests should run again and fail!*

21. ❑ Undo the previous change to fix the error.

✓ *Make sure the tests now pass.*

Verify the Component works correctly through DOM interactions

22. ❑ Add a new test with the description:

    `'should set showBuyFlights to false when the link is clicked'`

23. ❑ In the test, set the variable `el` to be a reference to the <a> element using this code:

    ```
    el = fixture.debugElement.query(By.css('a'));
    ```

    Make sure the import of `By` comes from `@angular/platform-browser`

24. ❑ Call `el.triggerEventHandler()` passing `'click'` and `null` as the arguments.

25. ❑ Use `expect()` to verify that `comp.showBuyFlights` is `falsy`.

Check your work...

```
  it('should set showBuyFlights to false when the
link is clicked', () => {
    el = fixture.debugElement.query(By.css('a'));
    el.triggerEventHandler('click', null);
    expect(component.showBuyFlights).toBeFalsy();
  });
```

26. ❑ Save the file.

    The tests should pass.

Congratulations! You have completed the exercise.

If you have more time, verify that the table is correctly hidden when the link is clicked

27. ❑ Create a test with the description: **`'should hide the flights table when the link is clicked'`**

28. ❑ Call `fixture.detectChanges();`

29. ❑ Using a CSS query of '`table`', get a reference to the `<table>` element.

Hint...

```
let tableEle =
fixture.debugElement.query(By.css('table'));
```

30. ❑ Verify that the `tableElement` is `truthy.`

31. ❑ Using the code from the previous test, trigger a `click` event on the <a> element.

32. ❑ Call `fixture.detectChanges();`

33. ❑ Query the `debugElement` element for a reference to the `<table>` element, as you did previously.

34. ❑ Check that the reference to the `<table>` element is `falsy`.

Check your work...

```
  it('should hide the flights table  when the link
is clicked', () => {
    fixture.detectChanges();
    let tableEle =
fixture.debugElement.query(By.css('table'));
    expect(tableEle).toBeTruthy();
    el = fixture.debugElement.query(By.css('a'));
    el.triggerEventHandler('click', null);
    fixture.detectChanges();
    tableEle =
fixture.debugElement.query(By.css('table'));
    expect(tableEle).toBeFalsy();
  });
```

When your work is saved, the test should run again and pass.

If you have more time, replace the `FlightsService` with a mock.

Our test is currently bound to the real implementation of the `FlightsService`. This means that as we evolve the `FlightsService` later in the course, the `BuyFlightComponent` spec will fail even though the component is still working. The solution is to replace `FlightsService` with a mock object.

35. ❑ Copy the `FlightsService` class from `C:\course2324\Exercises\FlySharp\src\app\services\flights.service.ts` and paste it into the test code BEFORE the `describe()` call. Fix any imports.

We can just use the original code here, as it is actually just a mock. Moving it into the test and calling it a mock class will isolate the `BuyFlightComponent` spec from changes to the real `FlightsService`.

36. ❑ Rename the class **`MockFlightsService`** and remove the `export`.

37. ❑ Create a new instance of `MockFlightService` called `mockFlightService`.

38. ❑ In the `providers` property of `TestBed.configureTestingModule()`, replace `FlightsService` with the `mockFlightService.` Take a look at the slide Providing the Mock Object for help.

39. ❑ Check that your tests work.

If you still have more time, write a test spec for the `FlightsService` by expanding the generated test.

There should be a separate test for each of the two methods, and it should validate that the correct number of flights is returned by each of the methods.

Use `toBe(...)` as the expect test. See `http://jasmine.github.io/2.4/introduction.html` for documentation.

This is the end of the exercise.

Objectives

In this exercise, you will

- **Resolve an issue with an existing E2E test**
- **Create a test to verify that the flights table is displayed**
- **Create a test to simulate user input**

Overview

In this exercise, you will write end-to-end tests using Protractor to check the behavior of the FlySharp application. Angular-cli generated a basic E2E test when we created the project, so we will start from that.

Running the existing E2E test

1. ❑ In WebStorm, make sure the project at `C:\course2324\Exercises\FlySharp` is open.

2. ❑ From the WebStorm terminal window, run the command **`exStart Ex4.2`**

3. ❑ In the WebStorm terminal, run **`ng e2e`** to run the end-to-end tests.

After a short period, you should see a message indicating that the test has passed. If you scroll up, you will see a message similar to below:

```
  fly-sharp App
    ? should display message saying "Special Offer of the
month 10% off all round-the-World flights
  Validate exercise 4.2 start
    ? should display message saying Special Offer of the
month 10% off all round-the-World flights
    ? should have an App-Home component
    ? should have a nav element
    ? should have an app-buy-flights element
    ? should have a Toggle Flights button
    ? should have a 0 flights displayed
    ? should have a 5 flights displayed
```

There are actually two sets of tests here, the first being the fly-sharp App test which you will enhance, the second being tests we use to ensure the quality of the course exercises. Feel free to investigate the exercise tests.

Running the E2E tests like this can be slow due to having to start the server each time. As an alternative, you can run **`ng serve`** *in an additional command prompt and then run the E2E test with* ***`ng e2e --dev-server-target=`***
NB: there really is nothing after the **=** *!*

Checking that the flights table is displayed

4. ❑ In `C:\course2324\Exercises\FlySharp\e2e\src\app.po.ts`, create a new method called **`getNumTableRows()`**

5. ❑ Add the code to return a count of the number of elements matching the CSS selector **`'table tbody tr'`**

Hint...

```
getNumTableRows() {
   return (element.all(by.css('table tbody tr'))).count();
 }
```

6. ❑ In `C:\course2324\Exercises\FlySharp\e2e\src\app.e2e-spec.ts`, duplicate the existing spec (`it(...)`).

7. ❑ Set the description of the test to **`"should show 5 rows in the table"`**

8. ❑ In the body of the test, after the call to `navigateTo()`, call the **`page.getNumTableRows()`** method and use **`expect().toEqual()`** to check that five rows were returned.

Hint...

```
it('should show 5 rows in the table', () => {
  page.navigateTo();
  expect(page.getNumTableRows()).toEqual(5);
});
```

9. ❑ Run **`ng e2e`** with the `--dev-server-target=` flag is appropriate again to test your work.

The tests should pass.

Testing the Toggle Flights link

10. ❑ In `C:\course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.html`, add an **`id`** attribute to the <a> tag with the value **`toggle`**

Adding an `id` to the element we want to test is the easiest way to be able to select it with CSS.

Hint...

```
<a (click)="onClickBuyFlights()" href='#' id="toggle">Toggle
Flights</a>
```

11. ❑ Back in `app.po.ts`, create a method called **`clickToggle()`**

12. ❑ In the method, select an element by the CSS selector `#toggle`, then call the **`click()`** method on the element.

Hint...

```
clickToggle() {
    element(by.css('#toggle')).click();
  }
```

13. ❑ Back in `app.e2e-spec.ts`, create a new test with a description of **`"should show 0 rows in the table when toggle is clicked"`**

14. ❑ In the test, after the call to `page.navigateTo()`:
    - Call **`page.clickToggle()`**
    - Use **`expect`** to check that the number of table rows is now 0

Hint...

```
it('should show 0 rows in the table when toggle is clicked', () => {
  page.navigateTo();
  page.clickToggle();
  expect(page.getNumTableRows()).toEqual(0);
});
```

15. ❑ Run **`ng e2e`** again to test your work.

    The tests should pass.

16. ❑ Verify that your tests are actually working by changing the number of expected rows in the last test to **`1`** and running the test again.

    The test should now fail.

17. ❑ Remove the error you just introduced.

18. ❑ Switch to the command prompt and use `<CTRL><C>` to stop the ng serve process.

Congratulations! You have completed the exercise.

If you have more time: enhance the tests

19. ❑ Add a test to check the number of columns in the table (count the number of `td` elements in a `tr`).

    *To do this, you first need to select a table row (see `getParagraphText()` as an example). Then call the all method with a selector of **`td`***

Detailed hint...

```
  getNumTableCols() {
    return (element(by.css('table tbody
tr')).all(by.css('td'))).count();
  }
```

20. ❑ Verify that the flight number of the 5th flight in the table is `FS2211` and that it's destination is `LHR`

A CSS query like this may help: ***`table tr:nth-child(5) td:nth-child(3)`***

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Implement an event binding**
- **Coordinate data between two components with a property binding**

Overview

You will create a new component that will display details of the currently selected flight. In a later exercise, you will add a form to this component to capture credit card details.

Create a new component to allow purchase of flights.

1. ❑ Stop any ng processes still running in a command prompt by switching to the command prompt and pressing <`Ctrl`><C>, then <Y>.

2. ❑ Return to the `C:\Course2324\Exercises\FlySharp` project in WebStorm.

3. ❑ From the WebStorm terminal window, run the command **`exStart Ex5.1`**

4. ❑ In the terminal, run the command **`ng generate component Payment`**

5. ❑ From the WebStorm terminal, run **`cpAddIns Ex5.1`** to copy a text file containing the HTML structure you will need for the next step.

6. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\payment\payment.component.html`

7. ❑ Replace the content of the page with the text from `C:\Course2324\Exercises\FlySharp\src\app\payment\payment.component.html.txt`.

8. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\payment\payment.component.ts`

9. ❑ Make a note of the value of the `selector`.

   ____

10. ❑ To the `PaymentComponent` class, add a field called **`selectedFlight`** of type **`Flight`**. Add an **`@Input()`** decorator to the new field.

Hint...

```
@Input() selectedFlight: Flight;
```

11. ❑ Switch back to `C:\Course2324\Exercises\FlySharp\src\app\payment\payment.component.html`.

12. ❑ For each of the `TODO` statements, replace all of the text within the `<div>` with an interpolation {{}} statement to display the appropriate value from `selectedFlight`. Use the safe navigation construct (`?.`) instead of `.` between `selectedFlight` and the property.

`selectedFlight` may be undefined. Using the `?.` prevents errors from being reported.

Hint...

To find the properties of `Flight`, just type **`selectedFlight?.`** and WebStorm will assist you. Or you can look in `C:\Course2324\Exercises\FlySharp\src\app\model\flight.ts`.

Below is an example of retrieving the flightNumber:
`{{selectedFlight?.flightNumber}}`

13. ❑ Save your work.

Displaying the `payment` component and passing the value of the currently selected flight to the `payment` component

14. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.html`

15. ❑ Add an element matching the value of the selector for the `payment` component to the end of the file.

Hint...

```
<app-payment></app-payment>
```

16. ❑ If you still have `ng serve` running in the Course Command Prompt, switch to that window and press `<Ctrl><C>` to stop the process.

17. ❑ Test your work by running **`ng serve`** in the WebStorm terminal window and opening the application in a browser as usual.

You should see the payment screen with labels, but no data. We have not actually selected a flight yet.

Displaying the selected flight

18. ❑ Back in `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.ts`, add the following to the `BuyFlightComponent` class:
    - Add a field **`selectedFlight`** of type **`Flight`**
    - Add a method **`onFlightClick()`** that takes an argument of **`flight : Flight`**
    - In the body of the method, save the value of flight to **`this.selectedFlight`**

Hint...

```
onFlightClick(flight : Flight){
    this.selectedFlight = flight;
}
```

19. ❑ Working in `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.html`:
    - Add a **`<button>`** to the table as a new column immediately prior to the line outputting the flight number (`<td>{{flight.flightNumber}}</td>`)
    - Set the button text to **`Buy`**
    - Add a click handler to the button that calls **`onFlightClick()`**, passing the **`flight`** as a parameter
    - Add the class attribute **`class="btn btn-primary btn-sm"`** to use Bootstrap styles for the button

Hint...

```
<td><button (click)="onFlightClick(flight)"
class="btn btn-primary btn-sm">Buy</button></td>
```

You declared `flight` as the loop variable holding each flight in an earlier exercise.

20. ❑ To keep the table aligned, add an empty **`<th>`** element before `<th>Flight</th>` in the table header.

21. ❑ Locate the `<app-payment>` element at the bottom of the file. Add a binding to bind **`selectedFlight`** on the `payment` component to the `selectedFlight` property on the `buy-flights` component.

Hint...

```
<app-payment [selectedFlight]="selectedFlight"></
app-payment>
```

22. ❑ Switch to the browser to check your work.

When you click on the Buy button of a flight, the flight data should appear in the `payment` component at the bottom of the page.

One issue remaining is the application currently displays the `payment` component even if no flight was selected.

23. ❑ In the `payment.component.html` file, to the `<div>` element at the start of the file, add an **`*ngIf`** attribute to test if `selectedFlight` exists.

24. ❑ Test your work.

    The purchase form should only be displayed if there is data to populate it. For example, when the page it first loaded, it should not be displayed.

Congratulations! You have completed the exercise.

If you have more time, implement property `get/set` methods.

25. ❑ Modify the PaymentComponent to use `get/set` methods for `selectedFlight`.

    You will need to rename the `selectedFlight` field as `_selectedFlight`.

STOP

This is the end of the exercise.

Objectives

In this Do Now exercise, you will

- **Examine the behavior of pseudo key event handlers**

1. ❑ Open the project `C:\Course2324\DoNows\DoNow51`

2. ❑ Run **`ng serve`** in a terminal window (in the project directory as usual).

3. ❑ Open **`localhost:4200`** in a browser.

4. ❑ Type some text into the input control.

   ✓ *You should see the text output on the page.*

5. ❑ Open `C:\Course2324\DoNows\DoNow51\src\app\app.component.html`

6. ❑ Inside the start tag of the input element, add the text:
   `(keyup.enter)="user=userInput.value"`

7. ❑ Save your work and switch back to the browser

8. ❑ Type more text in the input control, then press **`<Enter>`**

   ? *What happened?*

   ____

Congratulations! You have completed the exercise.

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Add a filter to enable sorting of the flights**
- **Bind the filter to a key event**
- **Bind the filter to a pseudo key event (bonus)**
- **Make the flight filter reusable (bonus)**

Overview

Event binding allows both DOM events and application-specific events to be captured and dispatched to an appropriate event handler.

1. ❑ Return to the `C:\Course2324\Exercises\FlySharp` project in WebStorm.

2. ❑ From the WebStorm terminal, type **`<CTRL><C>`** then **`Y <Enter>`** to stop ng `serve`.

3. ❑ From the WebStorm terminal window, run the command **`exStart Ex5.2`**

4. ❑ Run the command **`ng generate component FlightFilter`**

5. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\flight-filter \flight-filter.component.html`.

   Replace the content of the <p> element with the text **`Filter flights by origin:`**

   At the end of the new text, add an `<input>` control.

6. ❑ Working in the new `<input>` start tag:
   - Create a template variable called `filter`.
   - Define an event handler for the `keyup.enter` event that will call `onFilterEnter()` on the flight-filter component.
   - Use the template variable to pass the current `value` of the input control to the `onFilterEnter()` method.

Hint...

```
<input #filter
(keyup.enter)="onFilterEnter(filter.value)">
```

7. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\flight-filter\flight-filter.component.ts.`

   Inside the class, define a new private field called `filterEmitter` and initialize it to a new `EventEmitter` object which outputs a `string`.
   Decorate the field with `@Output()`

Hint...

```
@Output()
filterEmitter = new EventEmitter<string>();
```

When you add the import for EventEmitter, make sure you select async.d.ts from the list. The import should come from `@angular/core` and not one of the other ones listed by WebStorm.

8. ❑ Create a method called `onFilterEnter()` that takes a `string` as an argument called `filterValue.`

   Inside this method, call the `emit()` method on `this.filterEmitter` and pass `filterValue` as the argument to `emit()`.

Hint...

```
onFilterEnter( filterValue : string){
  this.filterEmitter.emit(filterValue);
}
```

Integrating the filter component

9. ❑ Make a note of the selector value for the FlightFilterComponent:

   ____

10. ❑ Add an element matching this selector to the top of `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.html`

11. ❑ Add an event handler binding for the `filterEmitter` event that calls `onFilterChange($event)`

Hint...

```
<app-flight-filter (filterEmitter) = "onFilterChange($event)"></app-flight-
filter>
```

12. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.ts`, create a field called `originFilter` of type `string` and initialize it to `null`.

    Create a method `onFilterChange()` that is passed `filterValue` as a string and saves it in the field `this.originFilter`.

Hint...

```
originFilter : string = null;

onFilterChange(filterValue: string) {
  this.originFilter = filterValue;
}
```

We are going to create a custom getter for `flights` that returns either the entire `flights` array or a filtered version. To do this, we need to rename the `flights` property so it is not returned by the default `flights` getter.

13. ❑ Rename the `flights` field as `_flights` (two places in the code).

14. ❑ Create a `get` method for `flights` using the following code:

```
get flights(): Flight[] {
    /**
    * Version of the flight getter that implements a simple filter
    */
    if (this.originFilter != null) {
      return this._flights.map((flight) => {
        console.log(flight);
        let match = flight.origin.startsWith(this.originFilter);
        if (match) {
          return flight;
        }
        // the filter expression stops empty elements being returned (drops the
null elements)
      }).filter(x => !!x);
    } else {
      return this._flights;
    }
  }
```

The code returns an array of flights filtered by the value of `this.originFilter`.

15. ❑ Test your work:
    - Make sure `ng serve` is running
    - Switch to the browser and refresh the page
    - Enter **`NRT`** into the filter field, then press `<Enter>`

You should see only flights with an origin of `NRT`.

Congratulations! You have completed the exercise.

If you have more time...

16. ❑ Modify the keyup binding to be just `keyup` instead of `keyup.enter`

When you test your work, you should see that the filter responds to every keystroke.

If you still have more time, make the flight-filter component reusable.

17. ❑ Modify the `FlightFilterComponent` so that the label for the input field is supplied by a field of the component called `label` instead of being hard-coded to `origin` in the template.

Hint...

Use interpolation to display the value of label. At this stage, initialize the field where it is declared.

Replace origin with `{{label}}`

18. ❑ Add an `@Input()` decorator to the `label` field.

19. ❑ Modify `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.html` to use a property binding in the `<app-flight-filter>` start tag to set the value of `label` to `"Origin"`.

Hint...

`[label]="'Origin'"` note the quotes within the quotes!
Or, of course: you could use: `label='Origin'`

When you test your work, you should see the `Origin` label by the filter control.

If you fancy a major challenge...

20. ❑ Reuse the FlightFilterComponent to add additional filtering for destination.

21. ❑ You will need to:
    - Use an additional `<app-flight-filter>` in the buy-flights component with the `label` set to `Destination`.
    - Configure an additional event handler method to set the value of a new field `destinationFilter` in the buy flights component.
    - Modify the `get` method for `flights` to do the additional filtering.

Part of the fun with this bonus is figuring out the filter code. If you are really stuck, use this hint...

```
get flights(): Flight[] {
  if (this.originFilter != null || this.destinationFilter != null) {
    return this._flights.map((flight) => {
      let match = true;
      if(this.originFilter != null) {
        match = flight.origin.startsWith(this.originFilter);
      }
      if(!match){
        return null;
      }
      if (match && this.destinationFilter != null) {
        match = flight.destination.startsWith(this.destinationFilter);
        if (match) {
          return flight;
        } else {
          return null;
        }
      }
      if(match) return flight;       // the filter expression stops empty
elements being returned (drops the null elements)
    }).filter(x => !!x);
  } else {
    return this._flights;
  }
}
```

One last bonus...

22. ❑ The filters are not well laid out. Use Bootstrap CSS to improve the layout.

You will need to use the Bootstrap row and column classes on the FlightFilterComponent. Unless you are a Bootstrap wizard, use a search engine or look at the solution!

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Define a Component Router configuration**
- **Incorporate the Component Router into your application**
- **Activate the tabs in the user interface (UI)**

Overview

In this exercise, you will make the tabs across the top of the screen switch between various parts of the application by using the Component Router.

Generating new Components to represent the Account and MyFlights screens

1. ❑ Return to the `C:\Course2324\Exercises\FlySharp` project in WebStorm.

2. ❑ From the WebStorm terminal, press <`Ctrl`><`C`> to stop `ng serve`.

3. ❑ In the WebStorm terminal window, run the following commands:
   - **`exStart Ex6.1`**
   - **`ng generate component Account`**
   - **`ng generate component MyFlights`**

4. ❑ Examine the code that has been generated.

   You should see that two directories (`account` and `my-flights`) have been created, each with the usual set of files for a component.

Adding the router configuration to the app component

5. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\app-routing.module.ts`

In the file, add the following path component pairs to the existing `Routes` array to define routes for the application:

```
path: '', component: AppComponent
path: 'home', component: HomeComponent
path: 'buy',  component: BuyFlightComponent
path: 'myflights', component: MyFlightsComponent
path: 'account',  component: AccountComponent
```

Hint...

```
const routes: Routes = [
  {
    path: '',
    component: AppComponent
  },
  {
    path: 'home',
    component: HomeComponent
  },
  {
    path: 'buy',
    component: BuyFlightComponent
  },
  {
    path: 'myflights',
    component: MyFlightsComponent
  },
  {
    path: 'account',
    component: AccountComponent
  }
];
```

6. ❑ Add any imports as needed.

7. ❑ Examine the `@NgModule` metadata: you should see the Routes array being loaded into the router with `RouterModule.forRoute(routes)` and the `RouterModule` then being exported.

   This makes the routes available to any module which imports the `AppRoutingModule`.

8. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\app.module.ts` verify that `AppRoutingModule` is included in the `imports` section of the `@NgModule` metadata.

   The `AppRoutingModule` was added to the imports section when you created the project using `ng new FlySharp --routing`

9. ❑ In `C:\Course2324\Exercises\FlySharp\src\app \app.component.html`:
   - Locate the set of links (<a>) used for the tabs near line 17.
   - Remove the href attributes from all of the <a> tags.
   - Add a `routerLink` to each of the four <a> tags with paths matching the route paths configured in `app.routes.ts`.
     – You will not need to use the empty path! That is just to keep the router happy if no path is specified.
   - Prefix each of the paths with a `'/'` character.

Hint...

```
<a routerLink='/home'>Home</a>...
```

10. ❑ In the same file:
    - Completely remove the `<app-home>` and `<app-buy-flight>` elements.
    - Replace them with a `<router-outlet></router-outlet>` element.

11. ❑ Start `ng serve -o` and test your work.

The initial content may be blank, or you may initially see two sets of tabs. It is because we have defined the default route to be the app component.

However, once you click on one of the tabs you should see the content of that tab. Try clicking on each of the four tabs.

Setting an initial tab

12. ❑ In `app-routing.module.ts`, modify the route with a `path` of `''` to redirect to `/home`. Add an additional property: `pathMatch` with the value `full`.

Hint...

```
{
  path: '',
  redirectTo: '/home',
  pathMatch: 'full'
},
```

13. ❑ Test your work. Make sure you enter the URL as `http://localhost:4200/` so that there is not a path already set.

You should now see the special offers page as the initial page. The path in the browser url bar should end with `/home`.

Highlighting the current Tab

To make the tabbed application work correctly from a user perspective, we need to ensure that the selected tab is highlighted.

14. ❑ Working in `app.component.html`, locate the four `<li>` elements (near line 17) which make up the "tabs" of the application. To each `<li>`, add **`routerLinkActive="active"`**

15. ❑ Test your work: Make sure you enter the URL as `http://localhost:4200/` so that there is not a path already set.

You should see that each tab is highlighted as it is clicked.

16. ❑ Select each of the tabs in turn, then use the Back button.

It should take you through each tab you selected.

Congratulations! You have completed the exercise.

If you have more time, add a parameterized route.

In this bonus exercise, you will add a parameterized route so that it is possible to specify the value of the origin filter for the buy flights page as a parameter on the URL.

If a parameter is passed in the URL, it will look like `http://localhost:4200/buy/LHR` where `LHR` is the airport code to filter by.

17. ❑ The steps you need to complete to make this bonus work are:
    - Duplicate the existing 'buy' route in the route table
    - Make one of the routes match on just the route prefix
    - Modify the other route to take a parameter of `:origin`
    - Modify the `BuyFlightComponent` so that is subscribes to `ActivatedRoute` and retrieves the `origin` parameter
    - Assign the value of the `origin` parameter to `this.originFilter`

Test your work. Use a URL like `http://localhost:4200/buy/LHR` and check the flights are filtered correctly.

If you have even more time, fix the display of the Origin filter.

While your code should be filtering the flights correctly, there is an issue now in that the filter component does not reflect the filter. This bonus fixes that issue.

18. ❑ The steps you need are:
    - Add an `@Input()` called **`initialValue`**to the `FlightFilterComponent`
    - Modify `buy-flight.component.html` by adding **`[initialValue]='originFilter'`** to the `<app-flight-filter>` for `Origin`
    - Add **`[value]="initialValue"`** to the `<input>` element in `flight-filter.component.html`

STOP

This is the end of the exercise.

Objectives

In this exercise you will

- **Create a Feature Module containing the AccountsComponent**
- **Use the router to lazy load the module**

Overview

Feature modules allow Angular functionality to be packaged in such a way as to enhance reusability and potentially improve performance.
In this exercise, you will package the existing `AccountComponent` into a new module and then configure the component router to lazy-load the module.

Creating a Feature Module

1. ❑ In the WebStorm terminal window, run the following commands:

```
exStart Ex6.3
ng generate module accounts --routing
```

2. ❑ Open `C:\course2324\Exercises\FlySharp\src\app\accounts\accounts.module.ts`

   Add **`AccountComponent`** into the declarations property of the `@NgModule` metadata.

   This makes `AccountComponent` part of the `AccountsModule`.

3. ❑ Open `C:\course2324\Exercises\FlySharp\src\app\app.module.ts.` Remove `AccountComponent` from the `declarations` section (and remove the associated typescript import).

4. ❑ Add **`AccountsModule`** to the `imports` section of `@NgModule.` Add the typescript import (if your IDE does not add it automatically).

5. ❑ Run the app with `ng serve` in the normal way.

   The application should work normally. The `AccountsComponent` should correctly display when the Accounts tab is selected.

Converting Module to use Lazy Loading

6. ❑ In `C:\course2324\Exercises\FlySharp\src\app\app-routing.module.ts` change the definition for the path of `'account'` by removing the component property and replacing it with a **`loadChildren`** property with the value: **`loadChildren: () => import('./accounts/accounts.module').then(mod => mod.AccountsModule)`**

This instructs the component router to load the module at the specified path when the account path is activated.

Account path definition should look like this:

```
{
   path: 'account',
   loadChildren: () => import('./accounts/
accounts.module').then(mod => mod.AccountsModule)
}
```

7. ❑ Open `C:\course2324\Exercises\FlySharp\src\app\accounts\accounts-routing.module.ts` and add a route specification to the routes array to load the component `AccountComponent` when a path of `''` is specified.

The new route in the accounts routing module:

```
const routes: Routes = [ {
  path: '',
  component: AccountComponent
}];
```

8. ❑ In `app.module.ts` remove `AccountsModule` from the imports in `@NgModule`

9. ❑ If you fail to remove `AccountsModule` from the imports, the module will still be statically loaded.

10. ❑ Save your work, switch to the browser.

You should see the application working correctly. Make sure you test the Account tab.

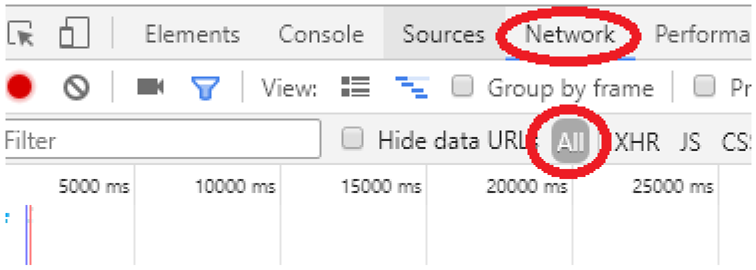
*If your application is not working correctly, use the **Router Tree** view in **Augry** to check that you have the correct route definition. In particular, make sure that `account` is marked as `Lazy`.*

Congratulations! You have completed the exercise.

If you have more time: prove that the `AccountsModule` is lazy loaded.

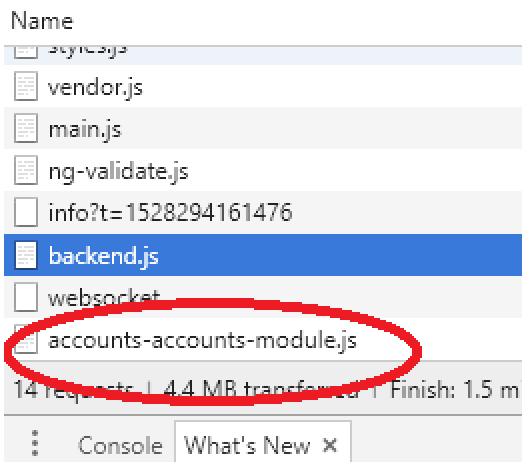
11. ❑ Close the Chrome browser you are using to test the application then open **localhost:4200** in the normal way.

12. ❑ Start Developer Tools and switch to the Network tab. Make sure that the **All** category is selected.



13. ❑ Click on the Account tab.

You should see a file with a name like `accounts-accounts-module.js` loaded when you clicked the Account tab.



Name
styles.js
vendor.js
main.js
ng-validate.js
info?t=1528294161476
backend.js
websocket
accounts-accounts-module.js
14 requests | 4.4 MB transferred | Finish: 1.5 m

You have proved the module is being lazily loaded.

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Create a custom pipe to perform currency conversion**
- **Incorporate the custom pipe into the flights table**

Overview

Custom pipes are used to perform conversions and formatting within the Angular 2 templates. In this exercise you will create a custom pipe to convert currencies.

Creating a custom pipe

1. ❑ Return to the `C:\Course2324\Exercises\FlySharp` project in WebStorm.

2. ❑ Create a new directory under `src\app` called `currency`

3. ❑ From the WebStorm terminal, type **`<CTRL><C>`** then **`Y <Enter>`** to stop ng `serve`.

4. ❑ In the WebStorm terminal window, run the following commands:
   - **`exStart Ex6.3`**
   - **`ng generate pipe currencyConversion --flat=false`**

5. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\currency-conversion\currency-conversion.pipe.ts`:
   - Add a field to the class with a name of RATE and a value of `0.8`.
   - Modify the **`transform`** method so that the type of the value parameter is a number.
   - Add the code to return the value multiplied by the rate prefixed with `"USD "`

Hint...

```
return "USD " + (value * this.RATE);
```

Integrating the currency converter pipe with the buy flights page

6. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\app.module.ts`, verify that **`CurrencyConversionPipe`** has been added to the declarations.

7. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.html`:
   - Locate the interpolation statement that is outputting the price of the flight (near line 23)
   - Modify the statement to pipe the price through the `currencyConversion` pipe

Hint...

```
{{flight.price | currencyConversion}}
```

8. ❑ Test your work by running `ng serve`, then view the **Buy Flights** page.

You should see the price column of the buy flights page outputting a value in USD.

The number of decimal places being output by the pipe is too great.

9. ❑ Use the `toFixed( n )` method to truncate the output to two decimal places.

Hint...

```
return "USD " + (value * this.RATE).toFixed(2);
```

Congratulations! You have completed the exercise.

If you have more time, parameterize the pipe to allow the rate to be passed in.

10. ❑ Modify the transform method of the pipe to expect the rate as the second parameter of type number.

11. ❑ Multiply the value by the new rate.

12. ❑ Update the buy flights page to pass the rate in to the `currencyConversion` pipe.

13. ❑ Test your work.

If you have even more time, upgrade the `BuyFlightComponent` so that the conversion rate is set by a field of the class.

14. ❑ Add a field called **`conversionRate`** to `BuyFlightComponent` and initialize it to **`4.0`**

15. ❑ Modify `buy-flight.component.html` so that the conversion rate is fetched from the `conversionRate` property.

Make the conversion rate adjustable.

Ideally, the conversion rate should be set based on a currency selection. To do that, we really need to use a form (which we will cover soon). For this bonus, create a field on the buy flight page to allow the rate to be entered.

16. ❑ It's up to you how you do this, but one way would be to make use of the code we saw when we looked at pseudo key bindings in the previous chapter.

STOP

This is the end of the exercise.

Objectives

In this Do Now exercise, you will

- **Bind a form control to the underlying model using the `[(ngModel)]` binding**

1. ❑ Open the `C:\Course2324\DoNows\DoNow71` project.

2. ❑ Open `src/app/preferences-form/preferences-form.component.html`

3. ❑ Add **`<pre>{{model | json}}</pre>`** to the end of the page just before the closing `<div>`

   `json` is a pipe that converts output to JSON.

4. ❑ Add an **`[(ngModel)]`** binding to the `<input>` at line 6 binding to **`model.name`**

5. ❑ Examine the other input controls—they are already bound to the model.

6. ❑ Run **`ng serve`**

7. ❑ Open **`http://localhost:4200`** in a browser.

8. ❑ Edit the data in the form.

   You should see the model output change at the bottom of the form.

Congratulations! You have completed the exercise.

STOP

This is the end of the exercise.

Objectives

In this Do Now exercise, you will

- **Use `ngControl` and CSS to provide visual feedback to the user**

View the classes applied by `ngControl`

1. ❑ Return to `src/app/preferences-form/preferences-form.component.html`

2. ❑ Add a template local variable called **`#controlState`** to the first `<input>`

3. ❑ Below the <div> element enclosing the `<input>`, add **`<br>{{controlState.className}}`**

4. ❑ Switch to the browser and examine the list of classes below the Name field.

5. ❑ Click into the field, then click outside the field. Then click inside the field again.

   You should see the classes change from `ng-untouched` to `ng-touched`.

6. ❑ Modify the text in the entry field.

   You should see the classes change from `ng-pristine` to `ng-dirty`.

7. ❑ Delete the text.

   You should see the classes change from `ng-valid` to `ng-invalid`.

Enable CSS Feedback

8. ❑ Open `src/app/preferences-form/preferences-form.component.css`

9. ❑ Remove the block comment from around the content at the top of the file.

10. ❑ Switch to the browser.

    ✓ *You should see a red border at the left edge of the input fields.*

11. ❑ Enter some text in the field.

    ✓ *You should see a green border at the left edge of the input fields.*

Congratulations! You have completed the exercise.

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Define a form using `ngFormModel`**
- **Add validation**
- **Implement form submission**

Overview

You have probably noticed that our Payment component has nowhere to input any payment details! In this exercise, you will resolve that by building a form and binding it to the underlying data model with `ngFormModel`.

Enabling support for Forms

To support form processing, the `FormsModule` needs to be imported into our root module.

1. ❑ Open `C:\course2324\Exercises\FlySharp\src\app\app.module.ts` and add **`FormsModule`** to the `imports` section of the `NgModule` meta data.

2. ❑ Resolve any issues with imports.

Defining a form

3. ❑ Return to the `C:\Course2324\Exercises\FlySharp` project in WebStorm.

4. ❑ From the WebStorm terminal, type **`<CTRL><C>`** then **`Y <Enter>`** to stop `ng serve`.

5. ❑ In the WebStorm terminal, window run **`exStart Ex7.1`**.

6. ❑ From the WebStorm terminal, run **`cpAddIns Ex7.1`**, which copies a `payment.ts` file with a `Payment` class defined in it and some template HTML in `payment.component.html.txt`.

7. ❑ Open `C:\course2324\Exercises\FlySharp\src\app\app.module.ts` and add **`FormsModule`** into the `imports` section of `@NgModule`. The typescript import is `import { FormsModule } from '@angular/forms';`

8. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\payment\payment.component.html` and copy the HTML form definition from `C:\Course2324\AddIns\Ex7.1\src\app\payment\payment.component.html.txt` to the end of the `payment.component.html` file.

   It's a basic HTML form with some Bootstrap classes to make it look nice. Nothing related to Angular at the moment.

9. ❑ Working in the same file:
   - Add `#paymentForm="ngForm"` to the `<form>` tag
   - Add a data binding to each `<input>`, `<textArea>`, and `<select>` tag using the banana-in-a-box syntax to bind model properties to ngModel.
   - The model properties should have the `form model.XXXX` where `XXXX` is the `id` of the form control. Here's the first one to get you started: `[(ngModel)]="model.name"`

   Nothing mandates that the model properties match the id values of the form elements; it is good practice, though.

The form should look like this...

```
<form #paymentForm="ngForm">
  <div class="form-group">
    <label for="name">Name</label>
    <input [(ngModel)]="model.name" type="text" class="form-control" id="name"
name="name" placeholder="Name">
  </div>
  <div class="form-group">
    <label for="address">Address</label>
    <textarea [(ngModel)]="model.address" class="form-control" id="address"
name="address" placeholder="Address" rows="4"></textarea>
  </div>
  <div class="form-group">
    <label for="email">Email</label>
    <input [(ngModel)]="model.email" type="email" class="form-control"
id="email" name="email" placeholder="Email">
  </div>
  <div class="form-group">
    <label for="cardNum">Card Number</label>
    <input [(ngModel)]="model.cardNum" type="text" class="form-control"
id="cardNum" name="cardNum" placeholder="Card Number">
  </div>
  <div class="form-group">
    <label for="cardType">Card Type</label>
    <select [(ngModel)]="model.cardType" type="text" class="form-control"
id="cardType" name="cardType" placeholder="Card Type">
      <option>VISA</option>
      <option>AMEX</option>
    </select>
  </div>
  <div class="form-group">
    <label for="expDate">Card Expiry</label>
    <input [(ngModel)]="model.expDate" type="month" class="form-control"
id="expDate" name="expDate">
  </div>
  <button type="submit" class="btn btn-default">Buy Flight</button>
</form>
```

10. ❑ At the bottom of the HTML, add `{{jsonModel}}` for debugging purposes.

11. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\payment\payment.component.ts`:
    - Add the following field to the `PaymentComponent` class to hold the model data: **`model: Payment = new Payment();`**
    - Add the following method, which will convert the current model into JSON so it can be shown in the HTML page for debugging:

```
get jsonModel() {
return JSON.stringify(this.model);
}
```

12. ❑ Run `ng serve` and test you work by selecting the BuyFlights tab in the browser then clicking the **Buy** button next to a flight.

You should see that as you enter data into the form, it is printed back to the page by the `{{jsonModel}}` interpolation. This is just for debugging.

Adding validation

13. ❑ To each of the `input`, `textArea`, and `select` fields, add the `required` attribute.

Hint...

```
<input type="text" [(ngModel)]="model.name" class="form-
control" id="name" placeholder="Name"  required>
```

Integrating the validation with Angular

14. ❑ Locate the submit button at the end of the form and add this binding: **`[disabled]="!paymentForm.form.valid"`**

15. ❑ Test your work.

The submit button should be disabled unless all the required fields have been completed.

Handling form submission

16. ❑ In the form tag, add a binding to `ngSubmit` to call `onSubmit()` in the `PaymentComponent` class when the form is submitted.

Hint...

```
(ngSubmit)="onSubmit()"
```

17. ❑ In `C:\course2324\Exercises\FlySharp\src\app\payment\payment.component.ts`, add an `onSubmit()` method to the `PaymentComponent.` In the `onSubmit()` method add an alert to display the value of the `jsonModel` property of the form.

Hint...

```
onSubmit(): void {
    alert(this.jsonModel);
  }
```

18. ❑ Test your work.

When you submit the form, you should see an alert with the form data.

You may notice that the Submit button is enabled even if the email address is not valid. We need enhanced validation to fix this, which we will see in the next exercise.

Congratulations! You have completed the exercise.

If you have more time: Add some visual feedback.

19. ❑ Add additional CSS classes to `src\assets\css\styles.css` to provide visual feedback as to the touched/untouched and pristine/dirty states. To avoid confusing with the additional styles, you could color the background of the controls or perhaps the right border.

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Modify the Payment form to use `FormBuilder`**
- **Improve user feedback**
- **Provide enhanced validation**

Overview

The `FormBuilder` class enables sophisticated validation to be built into forms.

Defining a `FormGroup` to represent the form data

1. ❑ Return to the `C:\Course2324\Exercises\FlySharp` project in WebStorm.

2. ❑ From the WebStorm terminal, press <`Ctrl`><`C`> to stop `ng serve`.

3. ❑ From the WebStorm terminal window, run the command **`exStart Ex7.2`**

4. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\app.module.ts`, add `ReactiveFormsModule` to the `imports` property.

5. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\payment\payment.component.ts`:
   - Modify the constructor to take an argument `private formBuilder : FormBuilder` (which will be injected)
   - Add a field to the class called `payForm` with a type of `FormGroup`

   Make sure the imports for `FormBuilder` and `FormGroup` both come from `@angular/forms`.

6. ❑ Create a new **`private`** method called **`buildForm()`**. Inside the method: use the `FormBuilder` to create a control group and assign it to **`this.payForm`**

Hint...

```
private buildForm(){
  this.payForm= this.formBuilder.group({});
}
```

7. ❑ Inside the `{}` brackets in the call to `group()`, create a control for each of our form fields using OLN notation. Each field should have the `Validators.required` validation associated with it.

Hint...

```
this.payForm= this.formBuilder.group({
    'name': ['', Validators.required]
});
```

The finished FormGroup (no peeking unless you really need it!)...

```
this.payForm= this.formBuilder.group({
 'name': ['', Validators.required],
 'address': ['', Validators.required],
 'email': ['', Validators.required],
 'cardNum': ['', Validators.required],
 'cardType': ['', Validators.required],
 'expDate': ['', Validators.required],
});
```

8. ❑ Call the **`buildForm()`** method from the `ngOnInit()` method.

9. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\payment\payment.component.html`:
   - Delete the `#paymentForm="ngForm"` attribute from the form start tag
   - Add the binding `[formGroup]="payForm"` to the form start tag

The form element should look like...

```
<form  [formGroup]="payForm" (ngSubmit)="onSubmit()">...</form>
```

This causes the FormGroup we have created with FormBuilder to be associated with the form instead of one being generated automatically.

10. ❑ Modify the `[disabled]` attribute of the submit button so that it tests the `valid` property of the new `ControlGroup`.

Hint...

```
[disabled]="!payForm.valid"
```

11. ❑ Remove all the `required` attributes from the input fields as they are no longer needed.

    In WebStorm, ***`<CTRL><R>`*** *launches the search / replace dialogue.*

12. ❑ Remove all of the `ngModel` bindings.

13. ❑ Replace all the `name="..."` attributes in all form controls (`<input>`, `<textarea>`, and `<select>`) elements with `formControlName="..."`

14. ❑ At the end of the form remove the `{{jsonModel}}` and add the following code to output the current value of the form group:
    `<p>Form value: {{ payForm.value | json }}</p>`

15. ❑ Restart `ng serve` and test your work.

    When you switch to the Buy Flights tab, you should see the form and it should allow you to enter data. You should see the form data output at the bottom of the page.

Populating the form with domain data

Template driven forms use `ngModel` to automatically keep the underlying model data synchronized with the form. When using reactive forms this task must be performed in our own code.

16. ❑ Copy the method below into the `PaymentComponent` class.

```
private buildSampleModel(){
  this.model.name="A Customer";
  this.model.address="Customer Address";
  this.model.email="a.customer@ltree.com";
  this.model.cardNum="1234123412341234";
  this.model.cardType="VISA";
  this.model.expDate=new Date();
}
```

17. ❑ Add a call to **`buildSampleModel()`** into the constructor for `PaymentComponent`.

This initializes the model with data, simulating data which has perhaps been retrieved from a web service in a real application.

18. ❑ In the `ngOnInit()` method, after the call to `buildForm()`: add the code below to populate the `FormGroup` from the `model`.

```
this.payForm.setValue(this.model);
```

19. ❑ Save the file and view the app in the browser.

When you view the payment form, it should be populated with the sample data.

Capturing Form Data in a Domain Object

20. ❑ Add a new method to `PaymentComponent` called **`preparePaymentForSave()`** which returns a type of `Payment.`

The method:

```
private preparePaymentForSave() : Payment {
}
```

21. ❏ In the `preparePaymentForSave()` method:
    - Retrieve the **`value`** property of `payForm` and assign it to a const called **`formData`**.
    - Create a new object of type `Payment`.
    - Initialize all of the properties of the `Payment` object with their corresponding values from `formData`
      - e.g., **`{name: formData.name, address:formData.address`**}
    - Return the `Payment` object from the method

The `preparePaymentForSave()` method:

```
private preparePaymentForSave() : Payment {
const formData = this.payForm.value;

const payment: Payment = {
  name: formData.name,
  address: formData.address,
  email: formData.email,
  cardNum: formData.cardNum,
  cardType: formData.cardType,
  expDate: formData.expDate
 }
return payment;
}
```

22. ❏ In the `onSubmit()` method:
    - Remove the existing code which shows `jsonModel` in an `alert()`
    - Call **`this.preparePaymentForSave()`**
    - Stringify the returned data with **`JSON.stringify()`**
    - Display the JSON data in an **`alert()`**

The `onSubmit()` method:

```
onSubmit(): void {

alert(JSON.stringify(this.preparePaymentForSave()));
}
```

Enhancing feedback

Let's add some feedback to the email field.

23. ❑ Back in `payment.component.html`:
    - Add a `<div>` element immediately below the `<input>` control for email.
    - Set the content of the `<div>` to be **`"You must provide a valid email address"`**
    - Use **`*ngIf`** to generate the `<div>` when the email form control is not valid.

Hint...

```
<div *ngIf="!payForm.controls.email.valid">You
must provide a valid email address</div>
```

24. ❑ Test your work.

You should see the message about email validation displayed if you delete the data in the email field.

Enhancing validation

25. ❑ Working in `C:\Course2324\Exercises\FlySharp\src\app\payment\payment.component.ts`:

    Use `Validators.compose` to create a validator for the name field that uses both the required validation and a min-length validation to force name to be at least five characters long.

Hint...

```
'name': ['',
Validators.compose([Validators.required,Validators.minLength(5)])],
```

26. ❑ Test your work.

Congratulations! You have completed the exercise.

If you have more time...

With the existing validation, the form control is only checking whether the email address is present. In this bonus, you will add some Angular validation to check the email address using pattern validation.

27. ❑ Add additional validation for the email field using pattern validation with the pattern:

```
"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
```

Search the Angular API docs for Validators to find the pattern validator.

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Integrate a REST service into the application**
- **Subscribe to Observables to consume the data from the service**

Overview

The existing `FlightsService` is loading dummy data into the application. In this exercise, you will add an HTTP client to the `FlightsService` to load the flight data from a Web service running at `http://localhost:8080/flightserver/flights`.

Converting the `FlightsService` to fetch data from the Web service

1. ❑ Return to the `C:\Course2324\Exercises\FlySharp` project in WebStorm.

2. ❑ From the WebStorm terminal, type **`<CTRL><C>`** then **`Y <Enter>`** to stop ng `serve`.

3. ❑ From the WebStorm terminal window, run the command **`exStart Ex8.1`**

4. ❑ Open `C:\course2324\Exercises\FlySharp\src\app\app.module.ts`, add `HttpClientModule` to the array of `imports` within the `@NgModule` meta-data.

5. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\flights\flights.service.ts`:
   - Remove the import of `FLIGHTS`
   - Add the parameter **`private http: HttpClient`** to the `constructor`

6. ❑ Inside the body of `getFlights(),` perform the following steps:
   - Delete return `FLIGHTS`
   - Create a constant **`URL`** with the value **`"http://localhost:8080/flightserver/flights"`**

Hint...

```
const url = "http://localhost:8080/flightserver/
flights";
```

7. ❑ At the end of the `getFlights()` method, call the `get(url)` method on the `http` field of the class.
Use the generic syntax `get`**`<Flight[]>`**`(url)` to specify that the method will return an `Observable` containing an array of `Flight[]`

The method should look like:

```
public getFlights(): Observable<Flight[]> {
  const url = 'http://localhost:8080/flightserver/flights';
  return this.http.get<Flight[]>(url);
}
```

Add Error Handling

Errors may be created either by the server-side application or with the HttpClient code. They are best handled within the service code and a simple error returned to the calling code.

8. ❑ Create a new method called **`handleError`**. The method should take a single argument of type **`HttpErrorResponse`** and return **`Observable<never>`**.

`Observable<never>` is used to indicate that the Observable will never hold data, just an error.

9. ❑ Check the type of the method argument. If it is an instance of ErrorEvent then write a message to the console with the content of **`error.error.message`**
If it is not an ErrorEvent then write the value of **`error.status`** and **`error.error`** to the console.

10. ❑ At the end of the method: return `throwError` using the string `'Server error - is the REST server running?'` as the argument to the constructor.

*Completed **`handleError()`** method.*

```
 private handleError (error: HttpErrorResponse ) :
Observable<never> {
   if(error.error instanceof ErrorEvent){
     // Client error
     console.error('Http communication error:',
error.error.message )
   } else {
     // Server error
     console.error(`Server error: ${error.status}.
Message body: ${error.message}`)
   }
   return throwError( 'Server error - is the REST
server running?');
 }
```

11. ❑ In the `getFlights()` method, chain a **`pipe()`** call to the existing **`http.get()`** statement.
Pass a reference to **`catchError(this.handleError)`** as the argument to `pipe()`.

The pipe() code.

```
return
 this.http.get<Flight[]>(url).pipe(catchError(this.handleError));
```

Modify the `BuyFlightComponent` to use the newly modified flight service.

12. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.ts`:
    - Change the initial state of `showBuyFlights to` **false**.
    - Add a new field called `errorMessage` of type `string`.

The `getFlights()` method in the flights service now returns an `Observable` to which you must subscribe to get the flights data.

13. ❑ Locate the `ngOnInit()` method.
    - Remove the assignment of `this._flights =` from the code in `ngOnInit`.
    - Append a call to `subscribe()` to the `flightsService.getFlights()` method.

Hint...

```
        this.flightsService.getFlights().subscribe()
```

14. ❑ Make the first argument to `subscribe()` an arrow function that assigns the data from **`subscribe() to this._flights`**. It should also set `this.showBuyFlights` to **`true`** to indicated the flights have been loaded.

    Make the second argument an arrow function that assigns the error data to the field **`errorMessage`**.

Hint...

```
    this.flightsService.getFlights().subscribe(
      (flights : Flight[])=>{this._flights = flights;
this.showBuyFlights = true},
      (error : any)=>this.errorMessage = error);
  }
```

15. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.html`, remove the `Toggle Flights` link.

    It takes a few moments to load the flights from the table. Users may begin clicking on the link while waiting for the page to load, which could be confusing!

16. ❑ Immediately before the start of the table, add an <h2> element with the content `{{errorMessage}}` to display the error if there is one.

    Set the `class` attribute to `bg-danger text-danger`.

Hint...

```
<h2 class="bg-danger text-
danger">{{errorMessage}}</h2>
```

17. ❑ Make the display of the <h2> conditional on there being an `errorMessage`.

Hint...

```
<h2 *ngIf="errorMessage" class="bg-danger text-
danger">{{errorMessage}}</h2>
```

18. ❑ Test your work.

Congratulations! You have completed the exercise.

If you have more time: Verify that the error handling works.

19. ❑ From the Windows services control panel, stop the **Flights Service** service.

20. ❑ Refresh the application in the browser, then go to the Buy Flights tab.

    You should see an error message.

21. ❑ Restart the **Flights Service** service.

If you have more time: Provide user feedback.

It could take several seconds to load the list of flights. The application should provide feedback that this is happening.

22. ❑ In the `FlightsService`, modify the `url` (near line 13) to be **`"http://localhost:8080/flightserver/allflights"`**

Using the URL will return over 4000 flights: plenty of time to see a loading message!

23. ❑ In `C:\course2324\Exercises\FlySharp\src\app\buy-flight\buy-flight.component.html`, add an <h2> tag with the message **`"Loading Flight Data"`** to the top of the template. This should be displayed only if `showBuyFlights` is `false`.

24. ❑ At the bottom of the page, modify the `<app-payment>` tag so that the payment component is displayed only if there is a selected flight.

If you have even more time: Retrieve the flights "page at a time"

Warning: this bonus is fairly challenging. We suggest you test your work frequently as you progress!

Instead of loading all 4000+ flights in a single hit, a more normal approach would be to load a small number and provide buttons to allow the user to request the next set of flights.

To support this, the web service can be invoked using `Post` and passing a JSON object containing properties of:

- *`start`—the index to return flights from*
- *`num`—the number of flights to return.*

As this is a bonus step, we will suggest the steps to perform from a high-level only. You can always check out the solution if you are stuck.

25. ❑ Add a new method to the `FlightsService` called
`getChunkOfFlights( start : number, num : number)`

In the method request, the flights from the URL: `"http://localhost:8080/flightserver/flights" using POST.` The body of the `POST` should be a JSON object with the start and num properties e.g., `{"start" : 60, "num" : 10}`. The JSON object should be populated with the data which is passed as method arguments to the new method.

26. ❑ An easy way of encoding the data as JSON is to create a object literal with properties of **`start`** and **`num`**. This can be done using the TypeScript syntax for initializing properties with the same name as their containing variables:

```
let data = {start, num};
```

27. ❑ Provide a `Headers` object, setting **`Content-Type`** to **`application/json`** as the 3rd argument to the `Post` method.

The new method and supporting class should look like this:

```
private headers = new HttpHeaders({'Content-Type':
 'application/json'});

  public getChunkOfFlights( start: number, num:
 number): Observable<Flight[]> {
    const url = 'http://localhost:8080/flightserver/
flights';
    const data = {start, num};
    const resultObservable =
 this.http.post<Flight[]>(url, JSON.stringify(data),
                                          {headers:
 this.headers}).pipe(catchError(this.handleError));
    return resultObservable;
  }
```

28. ❑ Modify the `ngOnInit()` method of `BuyFlightComponent` to call your new method. Initially hard-coding the start and length values to 0 and 20.

Like this:

```
ngOnInit() {
  this.activatedRoute.params.subscribe(params => {
    if(typeof params['origin'] !== 'undefined' ) {
      this.originFilter = params['origin'];
    }
  });

this.flightsService.getChunkOfFlights(0,20).subscribe(
    (flights : Flight[])=>{this._flights =
flights; this.showBuyFlights = true},
    (error : any)=>this.errorMessage = error);
}
```

Check your work: it should load 20 flights.

29. ❑ Add a new property to `BuyFlightComponent` called **`nextFlightIndex`** with an initial value of **`20`**.

30. ❑ Add a new method to the `BuyFlightComponent` called **`onNext()`**. In this method call `getChunkOfFlights()`. Use **`this.nextFlightIndex`** as the first argument and **`20`** as the second. Don't forget that you will need to subscribe to the result then assign it to **`this._flights`**. Use the code in `ngOnInit()` as a template.
In the method, you should also increment **this.nextFlightIndex** by 20.

The `onNext()` method:

```
onNext(){

 this.flightsService.getChunkOfFlights(this.nextFlightIndex
+= 20,20).subscribe(
    (flights : Flight[])=>{this._flights =
flights; this.showBuyFlights = true},
    (error : any)=>this.errorMessage = error);

}
```

31. ❑ Add a **`Next`** button to the template for the **`BuyFlightComponent`**. Call **`onNext()`** when the button is clicked.

32. ❑ Add a **`Previous`** button and the supporting code.

33. ❑ Add error checking to prevent the Next/Previous buttons trying to load non-existent flights. There is a method on the web service at the URL: `http://localhost:8080/flightserver/numflights` which returns the total number of flights.

STOP

This is the end of the exercise.

Objectives

In this exercise, you will

- **Create an attribute directive**
- **Add the directive into the template of a Component**

Overview

In this exercise, you will create an attribute directive that displays a clock. The clock can then be added to any template element by simply specifying the appropriate attribute.

Creating the attribute directive

1. ❑ Return to the `C:\Course2324\Exercises\FlySharp` project in WebStorm.

2. ❑ From the WebStorm terminal, type **`<CTRL><C>`** then **`Y <Enter>`** to stop ng `serve`.

3. ❑ In the WebStorm terminal window, run the following commands:
   - **`exStart Ex9.1`**
   - **`ng generate directive time`**

4. ❑ Open `C:\Course2324\Exercises\FlySharp\src\app\time.directive.ts` and make a note of the selector for the directive.

   ____

5. ❑ Modify the constructor to take **`el : ElementRef`**

   Mark it private so that `el` becomes a member of the class.

6. ❑ Add the following two methods to the end of the `Time` class. Most of this code creates and styles the clock, and is not directly related to Angular 2:

```
private showTime(el: ElementRef){
  let myDate =  new Date();
  el.nativeElement.innerHTML =
myDate.toLocaleTimeString("en-US");
}

ngOnInit(){
    this.el.nativeElement.style.fontSize = '2em';
    this.el.nativeElement.style.marginTop = '0.2em';
    this.el.nativeElement.style.float = 'right';
}
```

7. ❑ In the `TimeDirective` class constructor:
   - Call `this.showTime()` passing the appropriate parameter
   - Call `setInterval()` passing an arrow function that will call `showTime(...)`
   - Set the repeat duration to **`1000 mS`**

Hint...

```
setInterval(() => {
 this.showTime(el);
 }, 1000);
```

Fixing the unit test compilation issue

Adding the argument to the constructor for `TimeDirective` class causes the unit test to fail.

8. ❑ For the purposes of this exercise, rather than injecting the argument, just open `time.directive.spec.ts` and comment out the code.

There is an example of a simple test for this directive in the solutions directory.

Integrating the directive with the application

9. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\app.module.ts`, verify that `TimeDirective` is in the `declarations` array.

10. ❑ In `C:\Course2324\Exercises\FlySharp\src\app\app.component.html`:
    - Immediately after the end of the `</ul>` tag (near line 13), add a `<span></span>` element.
    - Add a class attribute specifying **`label label-primary`**.
    - Add an **`appTime`** attribute to the `<span>` to trigger the directive.

Hint...

```
<span class="label label-primary" appTime></span>
```

11. ❑ Test your work.

Congratulations! You have completed the exercise.

If you have more time...

12. ❑ Try adding the **`appTime`** attribute to other elements.

Still more time? Modify the `appTime` directive to take a color as an input.

*Modify the directive so you can use it like this **`appTime="red"`**. Such that the font of the clock will be the color specified.*

13. ❑ You will need to:
    - Add a field to the directive to hold the font color
    - Decorate the field with an **`@Input`** decorator
    - If your field is not called `apptime` then you will need to specify an alias in the `@Input()` decorator
    - Set **`style.color`** to the supplied color in the `ngOnInit()` method.

STOP

This is the end of the exercise.

Objectives

In this Do Now exercise, you will

- **Add animation to a Component**

1. ❑ Open the `C:\course2324\DoNows\DoNow91` project.

2. ❑ Open `src/app/preferences-form/preferences-form.component.ts` and examine the code.

   What is the name of the trigger?

   ____

3. ❑ Apply the trigger to the `<form>`, near the top of the file. Bind the **`trigger`** to the **`onOff`** property.

4. ❑ Run **`ng serve`**

5. ❑ Open `http://localhost:4200/`

6. ❑ Experiment with the **Hide** and **Show** buttons.

Congratulations! You have completed the exercise.

STOP

This is the end of the exercise.

Objectives

In this Do Now exercise, you will

- **Publish the FlySharp application to a web server**

1. ❑ Open a command prompt and change directory to `C:\course2324\FlySharpSolution`

2. ❑ Run the command **`ng build --prod --aot`**

3. ❑ Open **`C:\course2324\FlySharpSolution\dist`** with Windows Explorer.

   These are the files generated by the build process.

4. ❑ Copy all of the files and directories from the `dist` directory to `C:\inetpub\wwwroot`

5. ❑ Open **`http://localhost/`** in a browser.

   ✓ *You should see the working application. It is being served by the IIS server.*

Congratulations! You have completed the exercise.

STOP

This is the end of the exercise.