Angular 9

Exercise D-SHOP

April 2020

John Coumbe

Project setup

- This simple SHOP application allows the user to add items to a basket.
- They can remove items from the basket.
- Review the help image in **help/help.png**.
- Rebuild the starter project.

```
npm install
ng serve -o
```

Shop heading

- File **data/shop.data.ts** defines shop details in an object.

```
const ShopDetails = {
    name:"Southwold Organics",
    street:"14 Dolphin Street",
    town:"Southwold",
    postCode:"IP18 4HZ"
}
```

- We can define a class property to hold this data. *Currently it does not have a Typescript type defined.*

```
    shopDetails;
```

- In the **OnInit** LifeCycle method, we use the spread operator to make a copy of the shop details object.

```
ngOnInit() {
  this.shopDetails = {...ShopDetails};
}
```

- We can display this object in the template using a **built-in JSON pipe**.

```
<h2>{{ shopDetails | json }}</h2>
```

- This method uses **ES6 destructuring and template literals** to format the shop details object into a single string.

```
getDetails(): string {
  let { name, street, town, postCode } = this.shopDetails;
  return `${name}, ${street} ${town} ${postCode}`;
}
```

- We can the method in the template.

```
    <h2>{{ getDetails() }}</h2>
```

Display the stock items

- File **data/shop.data.ts** defines stock items in an array of objects.
- We can copy this data into a class property. We need to ensure this is a deep copy, i.e. a new array containing new objects.

```
stockItems = [];

this.stockItems = StockItems.map( item => ({ ...item}))
```

- We can display the stock items with a JSON pipe.

```
  <section>{{ stockItems | json }} </section>
```

Iterate over stock items with ngFor

- This approach does not allow us to format the data, or allow the user to select individual items.
- We can iterate over the stock array using the Angular structural directive ***ngFor**.

```
<section *ngFor="let item of stockItems">
{{ item | json }}</section>
```

- This code creates a locally scoped variable "item".
- The directive is structural.
- It creates four instances of this section into the DOM.
- *Review the DevTools Elements tab to confirm this.*

Style the stock items

- Use the "item" class in app.component.css to style each item.
- Wrap these items in a FlexBox using class "produce"

```
<section class="produce">
    <section class="item" *ngFor="let item of stockItems">
    {{ item | json }}</section>
</section>
```

- Create separate paragraphs for each object property.
  {{ item.desc }}
  {{ item.size }}
  {{ item.quantity }}
  {{ item.price }}
- Style the price with a **currency pipe**. Note that this takes the pound sign as its argument.

  ```
  <p>{{ f.price | currency:"GBP" }}</p>
  ```

Buy Items

- When the user clicks on a stock-item it should be added to a basket array. Define an empty array.

  ```
  basket = [];
  ```

- Create a method which adds item-objects to the basket

  ```
  buyItem( item ) {
      this.basket.push( item );
      console.table( this.basket )
  }
  ```

  - Add a click event in the template to call this method.

    ```
    <section .... (click)="buyItem(item)">
    ```

- This code creates a **copy-by-reference problem**. It pushes the original objects from the stock array into the basket.
- Add a **copy** of the object using the ES6 spread operator.

  ```
      buyItem( item ) {
          this.basket.push( {...item} );
      }
  ```

Improve the buyItem method

- The object added to the basket array has the wrong quantity. Fix that using the spread operator.

  ```
  this.basket.push({ ...item, quantity: 1 });
  ```

- Reduce the quantity in the stock object.

  ```
  item.quantity = Math.max(item.quantity - 1, 0);
  ```

Disable items which are sold out.

- This introduces a new bug. A stock-item can still be selected when the quantity available is zero.
- We could add conditional logic in the buyItem method.

  ```
  if( item.quantity ) { .... }
  ```

- Alternatively, we can use an expression in the template.

  ```
  (click)="item.quantity && buyItem(item)"
  ```

- This works but visually the item still looks selectable.
- The **ngClass directive** conditionally applies CSS class "outstock" if the quantity is zero.

  ```
  [ngClass]="{ 'outstock' : !item.quantity }"
  ```

- Note the item opening-tag contains a lot of metadata. Consider formatting it over multiple lines.

Create a basket total

- Define a total property.

  ```
  total: number = 0;
  ```

- Add the price of each item bought to the total in buyItem.

  ```
  this.total += item.price;
  ```

- Display the total in the template with a currency pipe.

  ```
  <p>TOTAL {{ total | currency:"GBP":"£" }}</p>
  ```

- We can use the **ngIf directive** to conditionally only display the total if items have been selected.

  ```
  <p *ngIf="total">TOTAL ....</p>
  ```

Display basket

- We can adapt the ngFor code used for the stock-items to display the basket.
  <section class="item" *ngFor="let item of basket" >

{{ item.desc }}

{{ item.size }}

{{ item.quantity }}

{{ item.price | currency:"GBP":"£" }}

Empty basket

- Implement an empty basket button.
- It may help to refactor the code in ngOnInit as shown.

```
<p class="button" *ngIf="basket.length"
(click)="initShop()">Empty</p>

ngOnInit() {
  this.initShop();
}

initShop() {
  this.shopDetails = {...ShopDetails};
  this.basket = [];
  this.stockItems = StockItems.map( item => ({ ...item}));
  this.total = 0;
}
```

Project review

- The project is one large component. We should think about architecting/**composing components** together.
- We are not making use of Typescript types.

Define custom types in Typescript (TS)

- We can type the class properties defined in **app.component.ts**.
- We can define **inline-styles**. For example this code describes the shape of the shopDetails object.

```
shopDetails : { name:string, street:string,
town:string, postCode:string };
```

- A better solution is to define a custom type using an **Interface**
- Create a new file **types/custom.types.ts** and define the TS interface.

```
interface Item{
    desc:string;
    price:number;
    size:string;
  quantity:number;
  code:string;
}

export {Item}
```

- Import that type into the main component.

  ```
  import { Item } from "./types/custom.types";
  ```

- Use that type to create an array of Items.

  ```
  private stock : Item [] ;
  ```

- Use the type for the basket as well.

  ```
  private basket : Item [] = [] ;
  ```

Component composition

- Angular components are **composable**.
- Angular projects consist of a **hierarchy** of related components.
- We want to take advantage of Angulars component archictecture.
- We need to decide on component boundaries, i.e. where we divide a project into chunks.
- In this exercise, we create an Item component which is used in both the array of stock items, and the basket.
- The item component will emit a custom event. We can choose to react to that event differently in different contexts.

Create a component for each item.

- Use the Angular CLI to **generate** a new components for each component.
- The **app.module.ts** file will be updated to include this new component.

  ```
  ng generate component item --dry-run
  ng generate component item
  ```

- Iterate over this new component instance using ngFor in the main template.
- You are doing **composition**. The main template **contains** instances of the Item component

  ```
  <app-item *ngFor="let item of stockItems"></app-item>
  ```

- The item component can share the CSS styles of its parent.
- Change the decorator in item.component.ts

  ```
    styleUrls: ['../app.component.css']

    <section class="item">ITEM</section>
  ```

Inputs

- We need to pass a stock-item object into each item-component instance as an argument. The Angular **Input decorator** passes values into component instances.
- Define the input inside item.component.ts

  ```
  import { Input } from '@angular/core';
  @Input() stockItem;
  ```

- Using this input in the main template.

  ```
  <app-item [stockItem]="item" ..
  ```

- If we can code to the item.component.ts constructor, it displays "undefined".

  ```
    constructor() {
      console.log(this.stockItem);
    }
  ```

- This is a **component lifecycle** issue. If we move the code into the OnInit lifecycle method, Angular will have bound the Inputs by this point.

  ```
    ngOnInit() { console.log(this.fruit);}
  ```

- The item-object is logged to the browser console.

  ```
  {type: "Pears", price: 1.85, instock: true, discount: 0.4}
  ```

Item template

- Refactor code from the main template into the item template.

  ```
  <section class="item">
    <p>{{ stockItem.desc }}</p>
    <p>{{ stockItem.size }}</p>
    <p>{{ stockItem.quantity }}</p>
    <p>{{ stockItem.price | currency:"GBP":"£" }}</p>
  </section>
  ```

- We can add back in conditional styling

  ```
  [ngClass]="{ 'outstock' : !item.quantity }"
  ```

Buying items

- We can add back logic in the main template to buy an item.

  ```
  <app-item (click)="item.quantity && buyItem(item)" >
  ```

- Alternatively, we can move the logic inside the Item component. We can emit an Output which is listened for in the main template.
- Import and define an Output before the constructor.

  ```
      import {EventEmitter,Output} ..
      @Output() select = new EventEmitter();
  ```

- Emit this event when the user clicks on an Item.

  ```
  (click)="stockItem.quantity && select.emit()"
  ```

- Listen for the select event in the main template

  ```
  <app-item
    (select)="buyItem(item)"
    [stockItem]="item"
    *ngFor="let item of stockItems">
  </app-item>
  ```

The basket

- Refactor the main template code to use the Item Component in the basket.

  ```
  <app-item
    [stockItem]="item"
    *ngFor="let item of basket">
  </app-item>
  ```

- We can listen for the same select event emitted by the Item Component.
- We can then call a different removeItem method.

  ```
      <app-item
      *ngFor="let s of basket"
      [item]="s"
      (select)="removeItem(s)">
  ```

```
    </app-item>

    removeItem( item ) {
  console.log(item)
}
```

- To remove items reliably, we should give each basket item a unique ID.
- This can be done when the item is created in buyItem.

```
this.basket.push({ ...item,
quantity: 1, id: Date.now() });
```

- This creates a type error. The basket objects have an ID field not defined in the Item interface.
- We can create a new interface to handle this.

```
interface BasketItem extends Item{
    id:number;
}
export {Item,BasketItem}
```

- Import this new type and use it.

```
import { Item, BasketItem } from "./types/custom.types";

private basket : BasketItem [] = [] ;
```

- The new removeItem method can now make use of this type.

```
removeItem( item ) {
    // Increase stock level for item removed.
    this.stock.find(
    i => i.desc === item.desc).quantity++;
    // Remove this item from the basket
    this.basket = this.basket.filter(
    i => i.id !== item.id);

}
```

Total

- The basket total does not update correctly after adding and removing items.
- Refactor this code to calculate the current value of the basket using reduce.

```
getTotal = () => this.basket.map(item => item.price).reduce((a, b)
=> a + b, 0);
```

{{ getTotal() | currency:"GBP":"£" }}

Dependency Injection

- **Dependency injection (DI)** allows us to separate our project into **loosely coupled** parts.
- The **connections** between the parts are clearly defined.
- Each part is testable in **isolation**.
- Mock data can be **injected** into components.
- This example injects a **service** into the main component using DI.

Create a data service

- Use the Angular-CLI to generate a new service.

```
ng generate service service/data --dry-run
ng generate service service/data
```

- This creates an empty service:

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

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

- Create a test method which can be called once this service is injected into a component.

```
  getName() {
    return "DataService"
  }
```

- Import this service into the shop component:

```
import { DataService } from "./service/data.service";
```

- Pass the service into the constructor.

```
constructor( private ds:DataService ) { .. }
```

- Call the getName method of the service.

```
console.log(this.ds.getName());
```

- Review the Injector Graph in the **Augury** DevTools.

The service reads a JSON file.

- We will extend the service to read data from a JSON file.
- Create a JSON file in the assets folder: **assets/fruitveg.json**. Note the JSON data needs to be stringified.
- Add the Angular HTTP module to app.module.ts

```
import {HttpClientModule} from '@angular/common/http';
imports: [BrowserModule,HttpClientModule]
```

- Import the Angular HTTP client into the service.

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

- Inject the HTTP client into the service constructor.

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

- Create a getData method that uses HTTP get() to read the JSON file.

```
  private path : string = "assets/fruitveg.json";

  getData() {
      return this.http.get( this.path );
  }
```

- The HTTP get method returns an **Observable**, an object for handling an asynchronous stream of data.
- We need to subscribe to this stream in order to read the data in the main component.

```
this.ds.getData( "assets/fruitveg.json")
.subscribe( data => this.stockItems = data )
```

- To avoid Typescript compile-time errors, we can define the return type of the getData method as an Observable stream of type Item-array.

```
    return this.http.get<Item[]>( this.path );
```

Mock Service

- Using services and dependency injection, it is easier to now test the code with a mock service.

- Add this mock service to the service folder.

  ```
  import { Injectable } from '@angular/core';
  import { HttpClient } from '@angular/common/http';
  import { Item } from "../types/custom.types";
  import { Observable,of } from 'rxjs';

  @Injectable()

  export class MockService {

      testData : Item[] = [ .... ]

      constructor( private http: HttpClient ) {}

      getData(path) : Observable<Item[]> {

          // The Observable OF function converts
          // testData into an Observable.
          return of(this.testData);
      }
  }
  ```

- Import the mock service into the main component.

  ```
  import { MockService } from "./service/mock.service";
  ```

- Add a providers property to the Component Decorator which redirects from the real service to the mock service.

  ```
  @Component({
      selector: 'app-root',
      templateUrl: './app.component.html',
      styleUrls: ['./app.component.css'],
      providers: [{ provide: DataService, useClass: MockService }]
  })
  ```

- *Observables will be covered in more detail later in the course.*