Introduction to Angular with TypeScript Workshop

Product Manager App

In this workshop, you will create a simple Angular app for managing products. The functions of the app include: listing, adding, updating, and deleting products. The products are very simple with just an id, name, and a price.

The starter code is located at:

https://github.com/bellingson/ng-workshop-starter

Setup Instructions

Install a recent version of nodejs (7.0 or later)

```
https://nodejs.org/en/
```

Install the angular-cli and typescript.

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

Make sure you have @angular/cli version 1.5.4 or later.

```
ng -v
```

Clone the starter code:

```
git clone git@github.com:bellingson/ng-workshop-starter.git
```

If you have difficulties with pervious step, try:

```
git clone https://github.com/bellingson/ng-workshop-starter.git
```

Install stuffmart(API) npm dependencies:

```
cd ng-workshop-starter/stuffmart
npm install
npm start
```

Open browser and navigate to:

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

You should see the StuffMart application.

Install product-mgr npm dependencies:

```
cd ng-workshop-starter/product-mgr
npm install
ng serve
```

Open browser and navigate to:

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

You should see an "Welcome to app!" message.

Which IDE should I use?

Only a few IDEs have good Angular support at this time. IntelliJ Ultimate Edition is recommended; however, you may use any IDE. IntelliJ Ultimate has features such as, JavaScript navigation, JavaScript code completion, and auto imports. You can use a free trial to complete the workshop. Visual Studio Code also has good support for Angular.

About the Project

The starter project contains:

1. typescript - Intro to TypeScript Project
2. product-mgr - Angular project generated by Angular CLI
3. stuffmart - shopping application

Exercise #1 - TypeScript

Instructions

1. Open ng-workshop-starter/typescript project in your IDE.
2. create Person class in person.ts
3. add member variables in the class constructor and a describe method to the Person class
4. import Person class into main.ts
5. create an instance of Person in main.ts and call the describe method
6. compile and run the main.ts

Step-by-step

Open typescript project in your IDE.

Create Person class in the person.ts file

```
export class Person {

}
```

Add member variables for id, name, and age to the Person class in the class constructor. Use the appropriate data type. Add a describe method to the Person class that prints the person's name and age to the console.

```
export class Person {

    constructor(public id: number,
                public name: string,
                public age: number) {}

    describe() {
        console.log(`${this.name} is ${this.age} years old`);
    }

}
```

Import Person class into main.ts.

```
import { Person } from './person';
```

Create an instance of Person and call the describe method.

```
import { Person } from './person';

let p = new Person(1, 'Bob Jones', 40);
p.describe();
```

Compile and run the main.ts

```
cd ng-workshop-starter/typescript
tsc
node main
```

Should print "Bob Jones is 40 years old"

Exercise #2 - Getting Started with Angular CLI.

Open the project in your IDE. If using IntelliJ, select Open Project and choose the product-mgr.ipr file in the root of the product-mgr project.

```
cd ng-workshop-starter/product-mgr
```

Install the project's dependencies with npm:

```
npm install
```

Start the angular-cli test server.

```
ng serve
```

Open http://localhost:4200 in Chrome. The page should display "Welcome to app!".

Spend some time exploring the product-mgr app. Almost everything was generated by the Angular CLI. When you use the 'ng new myapp' command, it will generate a project that is nearly identical. I have only added bootstrap css to the index.html and added a src/app/product.data.ts file with some demo data.

The entry points into your application are:

1. index.html - the page hosting our angular app
2. main.ts - bootstraps the angular app
3. app.module.ts - all angular apps have a root module
4. app.component.ts - all angular apps have a root component

Exercise #3 - Import FormsModule and HttpClientModule

We will create a simple CRUD app that uses HTML forms and makes calls to a restful API. To implement these features, we will need to add Angular FormsModule and HttpCientModule to our app.module imports.

app.module.ts

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

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

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

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

Exercise #4 - Product List Component

Let's start building our app by creating a product list component.

The 'ng g component' command is used to generate new angular components. 'g' is short for 'generate'. By default the angular cli creates each new component in a new folder. We'll create a 'product' folder and add several related components there. When we use the 'ng g component' command, we'll include the path to the 'product' folder and the '--flat' flag to indicate that the cli should not create a new folder.

```
cd ng-workshop-starter/product-mgr
ng g component product/product-list --flat
```

Open app.module.ts file. Notice that ProductListComponent has been imported and added to the 'declarations' section of the '@NgModule'.

Display List of Products

Open product-list.component.ts. Import the PRODUCTS from product.data.ts and assign it to a products member variable.

product-list.component.ts

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

import { PRODUCTS } from './product.data';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {

  products: Array<any> = PRODUCTS;

  constructor() { }

  ngOnInit() {
  }

}
```

Open app.component.html and modify as follows:

app.component.html

```
<h1>
  Product Manager
</h1>

<app-product-list></app-product-list>
```

Open product-list.component.html and modify as follows:

```
<table class="table table-striped">
    <tr>
        <th>Name</th>
        <th>Price</th>
    </tr>
    <tr *ngFor="let product of products">
        <td>{{product.name}}</td>
        <td>{{product.price | currency: 'USD':'symbol':'1.2' }}</td>
    </tr>
</table>
```

App should now look like this:

Exercise #5 - Product Add Component

Initially, you will create the product add form in the product-list.component. Later on, you will refactor it into it's own component.

Instructions

1. Create an interface named Product in product.model.ts with the fields: id, name, and price.
2. At the top of the product list, add a simple form with inputs for product name and price, and a button to submit the form
3. use Angular template driven forms to add products
4. refactor the app so that product add form has it's own component that emits new values to the parent product list component.

Step-by-step

Create the file product/product.model.ts with contents:

product.model.ts

```
export interface Product {
    id: number;
    name: string;
    price: number;
}
```

Open product-list.component.html. At the top of the file add this simple form:

product-list.component.html

```
    <form>
        <input type="text" name="name" required/>
        <input type="text" name="price" required/>
        <button>Add</button>
    </form>
```

Now, let's Angularize the form:

product-list.component.html

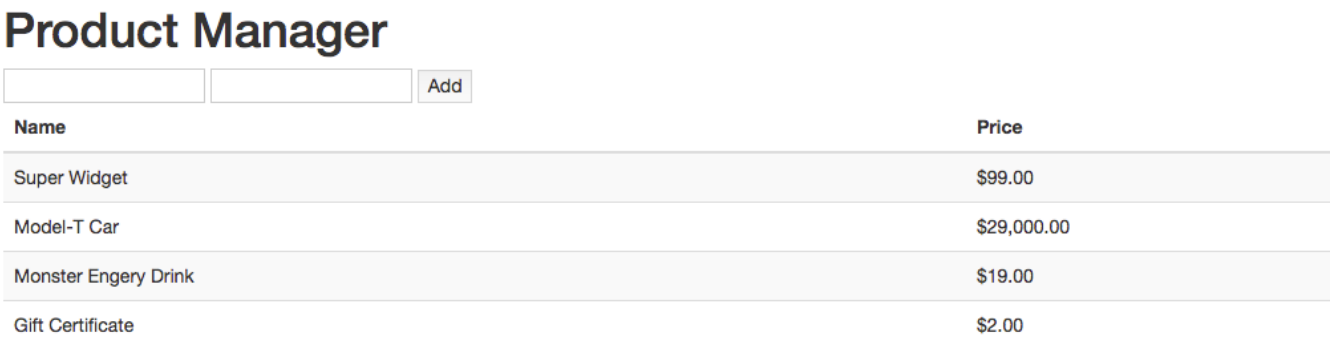
```
    <form #f="ngForm" (ngSubmit)="addProduct(f.value)">
        <input type="text" name="name" required ngModel />
        <input type="text" name="price" required ngModel />
        <button [disabled]="!f.valid">Add</button>
    </form>
```

Implement the addProduct method in the product-list.component.ts file

product-list.component.ts

```
    addProduct(value) {
        this.products.push(value);
    }
```

Your app should now look like this and you should be able to add products.



In a real app, a product would have more fields and components should typically do one thing. Let's refactor the product add form into it's own component.

```
ng g component product/product-add --flat
```

Copy the form into the product-add.component.html file. Replace form with product add directive in product-list.component.html

```
<app-product-add></app-product-add>
```

Implement product-add.component.ts:

product-add.component.ts

```
import { Component, Output, EventEmitter } from '@angular/core';
import {Product} from "./product.model";

@Component({
  selector: 'app-product-add',
  templateUrl: './product-add.component.html',
  styleUrls: ['./product-add.component.css']
})
export class ProductAddComponent {

  @Output() newProduct = new EventEmitter<Product>();

  addProduct(product: Product) {
      this.newProduct.emit(product);
  }

}
```

In product-list.component.html modify the app-add-product selector to connect the 'newProduct' event to the parent component 'addProduct' method.

```
<app-product-add (newProduct)="addProduct($event)"></app-product-add>
```

Your refactored app should continue to function.

Challenge Step

For the rest of the workshop, make your forms look better by adding CSS styles to the component CSS files. If you are familiar with Bootstrap, use Bootstrap styles.

Exercise #6 - clear product add form after submit

product-add.component.ts

```
import {NgForm} from "@angular/forms";
...

export class ProductAddComponent implements OnInit {

    @Output() newProduct = new EventEmitter<Product>();

    @ViewChild('f') form: NgForm;

  constructor() { }

  ngOnInit() {
  }

  addProduct(product: Product) {
      this.newProduct.emit(product);
      this.form.reset();
  }

}
```

Exercise #7 - Configure Angular CLI to Product API Proxy

Instructions

1. start the stuffmart application
2. create product-mgr/proxy.config.json file
3. add --proxy-config param to package.json
4. restart product-mgr

Start the stuffmart application

```
cd stuffmart
npm start
```

Checkout the product API data. http://localhost:3000/ and http://localhost:3000/api/admin/product. The application has an in-memory store of product data that will NOT persist between restarts.

Create product-mgr/proxy.config.json

```
{
  "/api/*": {
    "target": "http://localhost:3000",
    "secure": false,
    "logLevel": "debug"
  }
}
```

Modify start command in product-mgr/package.json

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

Restart the product-mgr, but now use "npm start" command

```
npm start
```

Exercise #8 - Product Service

Instructions

1. generate the product service
2. import HttpClient and inject it in your product service constructor
3. import rxjs and implement the query method to retrieve a list products from the product REST service
4. add ProductService to providers in app.module.ts
5. inject ProductService into product-list.component.ts constructor
6. implement the fetchProducts method in product-list.component.ts
7. implement ProductService methods for: get, add, update, and delete
8. update product-list.component.ts addProduct method to use the ProductService

Step-by-step

Generate the product service.

```
    cd product-mgr
    ng g service product/product
```

Import Http and inject it in your prodcut service constructor.

product.service.ts

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

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

@Injectable()
export class ProductService {

  constructor(private http: HttpClient) { }

}
```

Import rxjs and implement the query method to retrieve a list products from the product REST service

product.service.ts

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

import { Observable } from 'rxjs/Observable';

import {Product} from "./product.model";

@Injectable()
export class ProductService {

    baseUrl = '/api/admin/product';

  constructor(private http: HttpClient) { }

  query() : Observable<Array<Product>> {
    return this.http.get<Array<Product>>(this.baseUrl);
  }

}
```

Add ProductService to providers in app.module.ts

app.module.ts

```
...
import {ProductService} from "./product/product.service";

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

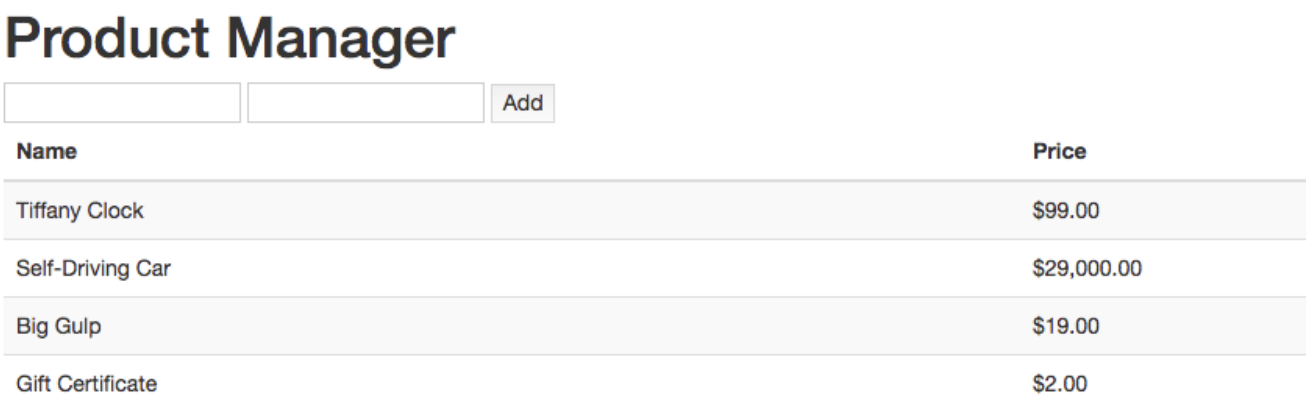
Inject ProductService into product-list.component.ts constructor

```
...
import {ProductService} from "./product.service";
...

constructor(private productService: ProductService) { }
```

Implement the fetchProducts method in product-list.component.ts. Remove references to test data PRODUCTS.

```
...
products: Array<Product>;
...
ngOnInit() {
  this.fetchProducts();
}

fetchProducts() {
  this.productService.query().subscribe(products => {
     this.products = products;
  });
}
```

Your app should now be retrieving products from the product REST service. The products should match those displayed in the front-end app.



Implement ProductService methods for: get, add, update, and delete

```
...
  get(id: number) : Observable<Product> {
      return this.http.get<Product>(this.baseUrl + '/' + id);
  }

  add(product: Product) : Observable<any> {
      return this.http.post(this.baseUrl, product);
  }

  update(product: Product) : Observable<any> {
      return this.http.put(this.baseUrl + '/' + product.id, product);
  }

  delete(product: Product) : Observable<any> {
      return this.http.delete(this.baseUrl + '/' + product.id);
  }
```

Update product-list.component.ts addProduct method to use the ProductService

product-list.component.ts

```
  addProduct(product: Product) {
      this.productService.add(product)
          .subscribe(r => {
              this.fetchProducts();
          });
  }
```

Test the application to verify that when you add products they are displayed in the front-end.

Exercise #9 - Routing

Instructions

1. create app/app.routing.ts
2. add appRouting to imports in app.module.ts
3. in app.module.ts add HashLocationStrategy as the LocationStrategy provider
4. replace app-product-list with router-outlet in app.component.html

app/app.routing.ts

```
import { Routes, RouterModule } from '@angular/router';
import {ProductListComponent} from "./product/product-list.component";

const routes: Routes = [
    { path: '', pathMatch: 'full', redirectTo: 'list' },
    { path: 'list', component: ProductListComponent }
];

export const appRouting = RouterModule.forRoot(routes);
```

app.module.ts

```
import { LocationStrategy, HashLocationStrategy } from '@angular/common';
import {appRouting} from "./app.routing";

@NgModule({
  declarations: [
    AppComponent,
    ProductListComponent,
    ProductAddComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    appRouting
  ],
  providers: [ ProductService,
      { provide: LocationStrategy, useClass: HashLocationStrategy }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.html

```
<h1>
  Product Manager
</h1>


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

Exercise #10 - Add ProductUpdateComponent route

Instructions

1. generate ProductUpdateComponent using the angular cli
2. add route in app.routing.ts
3. add routerLink in product-list.component.html

generate ProductUpdateComponent

```
ng g component product/product-update --flat
```

add route in app.routing.ts

```
const routes: Routes = [
    { path: '', pathMatch: 'full', redirectTo: 'list' },
    { path: 'list', component: ProductListComponent },
    { path: ':id', component: ProductUpdateComponent }
];
```

add routerLink in product-list.component.html

```
<table class="table table-striped">
    <tr>
        <th>Name</th>
        <th>Price</th>
    </tr>
    <tr *ngFor="let product of products">
        <td><a [routerLink]="['../', product.id]">{{product.name}}</a></td>
        <td>{{product.price | currency: 'USD':true: '1.2' }}</td>
    </tr>
</table>
```

Exercise #11 - Implement ProductUpdateComponent form

Instructions

1. add form to product-update.component.html
2. inject ProductService, ActivatedRoute, and Router into ProductUpdateComponent
3. get the product id from the ActivatedRoute
4. create a fetchProduct method to get the product from the productService
5. Angularize the product update form
6. Implement updateProduct() method
7. Add delete form to product-update.component.html
8. Implement deleteProduct() method

Add form to product-update.component.html

```
<form>
    <table>
        <tr>
            <th>Name</th>
            <td><input type="text" name="name" required/></td>
        </tr>
        <tr>
            <th>Price</th>
            <td><input type="text" name="price" required/> </td>
        </tr>
        <tr>
            <th></th>
            <td><button>Update</button></td>
        </tr>
    </table>
</form>
```

Inject ProductService, ActivatedRoute, and Router into ProductUpdateComponent Get the product id from the ActivatedRoute create a fetchProduct method to get the product from the productService

```
export class ProductUpdateComponent implements OnInit {

   product: Product;

  constructor(private productService: ProductService,
              private route: ActivatedRoute,
              private router: Router) { }

  ngOnInit() {

      this.route.params.subscribe(params => {
           let id = +params['id'];
           this.fetchProduct(id);
      });

  }

  fetchProduct(id: number) {
      this.productService.get(id)
          .subscribe(product => this.product = product);
  }

}
```

Angularize the product update form

```
<div *ngIf="product">

<form #f="ngForm" (ngSubmit)="updateProduct()">
    <table>
        <tr>
            <th>Name</th>
            <td><input type="text" name="name" [(ngModel)]="product.name"
required/></td>
        </tr>
        <tr>
            <th>Price</th>
            <td><input type="text" name="price" [(ngModel)]="product.price" required/>
</td>
        </tr>
        <tr>
            <th></th>
            <td><button [disabled]="!f.valid">Update</button></td>
        </tr>
    </table>
</form>

</div>
```

Implement updateProduct() method

```
  updateProduct() {
      this.productService.update(this.product)
          .subscribe(r => {
                this.router.navigateByUrl('/list');
          });
  }
```

Add delete form to product-update.component.html

```
<h3>Delete Product</h3>

    <form (ngSubmit)="deleteProduct()">
        <button>Delete</button>
    </form>
```

Implement deleteProduct() method

```
  deleteProduct() {
      if(!confirm("Are you sure you want to delete this?"))
          return;

      this.productService.delete(this.product)
          .subscribe(r => {
             this.router.navigateByUrl('/list');
          });
  }
```

Exercise #12 - Create a Reactive search form

Instructions

1. copy modal dialog from resource/add-product-modal.html into bottom of product-list.component.html
2. modify product-list.component.html so that products are added via a modal dialog
3. add html to product-list.component.html so that add button is top-right of product list
4. modify addProduct() method so that modal closes on completion
5. add an input control for your search query text
6. add ReactiveFormsModule to your app.module.ts @NgModule imports
7. in product-list.component.ts add your findText: FormControl member and initialize it
8. add [formControl]="findText" attribute to your find input element
9. modify productService.query() and fetchProducts() methods to accept an optional findText value
10. subscribe to the findText valueChanges with debounceTime operator

Add html to product-list.component.html so that add button is top-right of product list

```
<div class="tools clearfix">
    <div class="pull-left">


    </div>
    <div class="pull-right">
        <a class="btn btn-primary" data-toggle="modal" href="#addProduct"><span
class="glyphicon glyphicon-add"></span> Add</a>
    </div>
</div>
```

product-add.component.ts - modify addProduct() method so that modal closes on completion

```
// above @Component annotation
declare var jQuery: any;

...

  addProduct(product: Product) {
      this.newProduct.emit(product);
      this.form.reset();
      jQuery('.modal').modal('hide');
  }
```

Add an input control for your search query text product-list.component.html

```
    <div class="pull-left">

        <input type="text" name="findText" placeholder="Search..."/>

    </div>
```

add ReactiveFormsModule to your app.module.ts @NgModule imports

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

product-list.component.ts add your findText: FormControl member and initialize it

```
...
import { FormControl } from '@angular/forms';
...

  findText = new FormControl();

  constructor(private productService: ProductService) { }

  ngOnInit() {

      this.fetchProducts();
  }
```

add [formControl]="findText" attribute to your find input element

```
<input [formControl]="findText" type="text" name="findText" placeholder="Search..."/>
```

modify productService.query() and fetchProducts() methods to accept an optional findText value

```
    // ProductService
  query(findText?: string) : Observable<Array<Product>> {

      let url = this.baseUrl;
      if(findText) {
          url += `?findText=${encodeURI(findText)}`;
      }

      return this.http.get<Array<Product>>(url);
  }

    // ProductListComponent
  fetchProducts(findText?: string) {
      this.productService.query(findText)
          .subscribe(products => this.products = products);
  }
```

subscribe to the findText valueChanges with debounceTime operator

```
import 'rxjs/add/operator/debounceTime';
...

  ngOnInit() {

      this.findText = new FormControl();
      this.findText.valueChanges
          .debounceTime(500)
          .subscribe(value => {
              this.fetchProducts(value);
          });

      this.fetchProducts();
  }
```

The end...