Angular 2 with Spring MVC Workshop

Product Manager App

In this workshop, you will create a simple Angular 2 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 app will be integrated into an existing Spring MVC app.

The starter code is located at: https://github.com/bellingson/ng-workshop-starter

Please follow the setup instructions in the README.md file.

The starter project contains:

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

IntelliJ Ultimate Edition is highly recommended for this workshop. The community edition works well, but lacks many helpful features; such as, JavaScript navigation, JavaScript code completion, and auto imports. IntelliJ has great support for both Angular 2 and Java. You can use a free trial to complete the workshop.

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. If using IntelliJ, select Open Project and chose the 01-typescript.ipr file.

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"

Excercise #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 "app works!".

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 - 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
mkdir src/app/product
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() {
  }

}
```
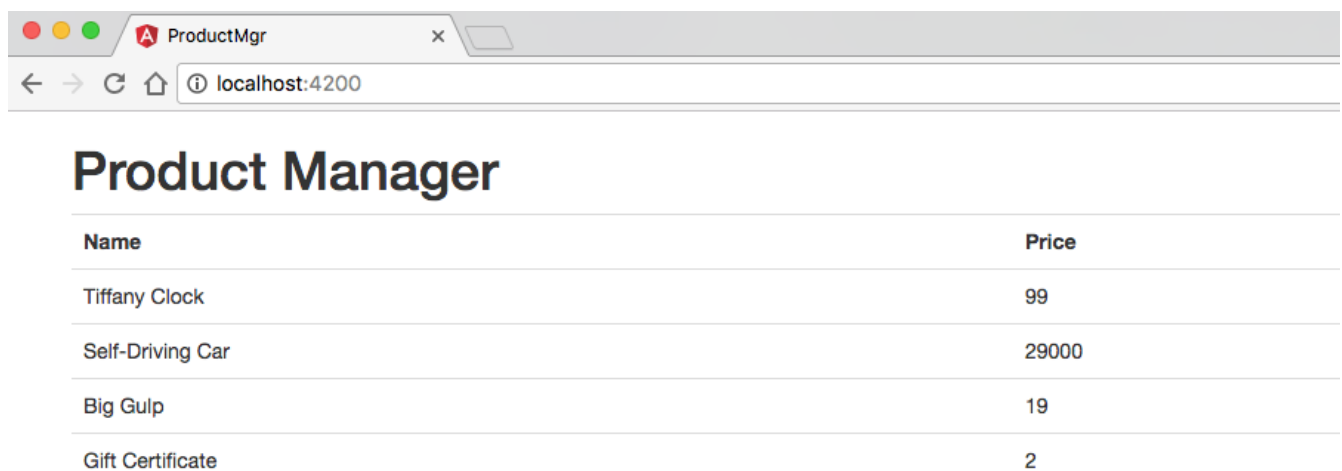
Open app.component.html and modify as follows:

app.component.html

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

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

App should now look like this:

Exercise #4 - 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 2 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>
```
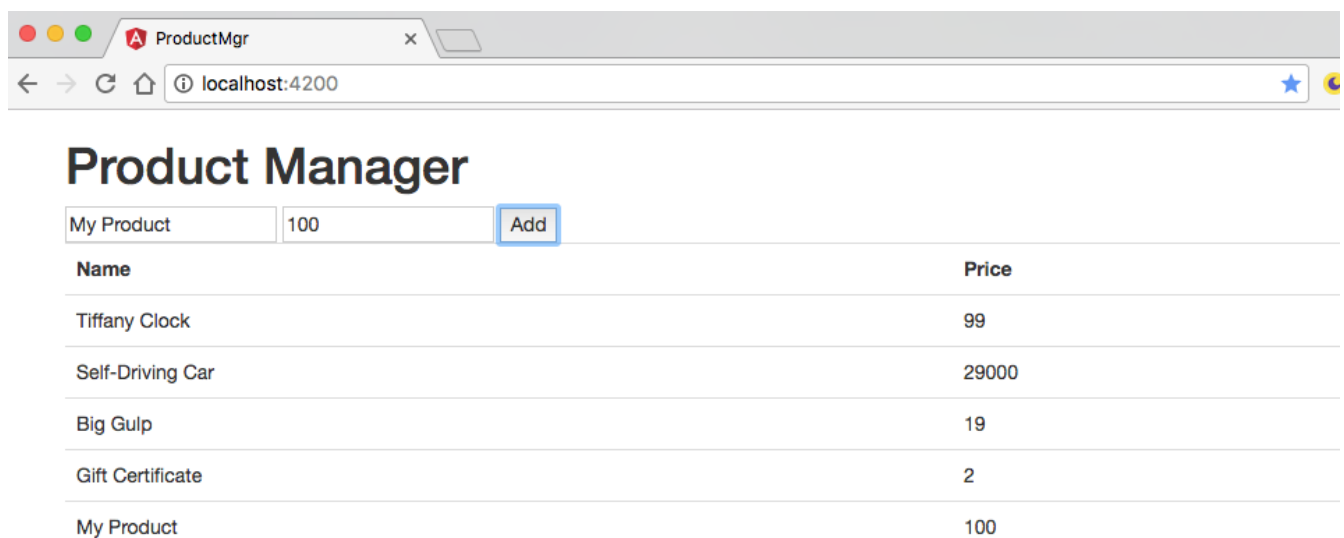
Now, let's Angularize the form:

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);
  }

}
```

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 #5 - Integrate with Spring MVC app

Instructions

1. stop your Angular CLI development server

2. close your IDE

3. open the stuffmart app in your IDE

4. create a directory src/main/js

5. move product-mgr to src/main/js

6. start application ./gradlew appRun (or ./gradlew.bat appRun on windows)

7. open application in Chrome

8. log into the app

9. navigate to your Product Manager (product-mgr)

Step-by-step

Close your IDE. Move the product-mgr src/main/js.

```
cd stuffmart
mkdir src/main/js
mv ../product-mgr src/main/js
```
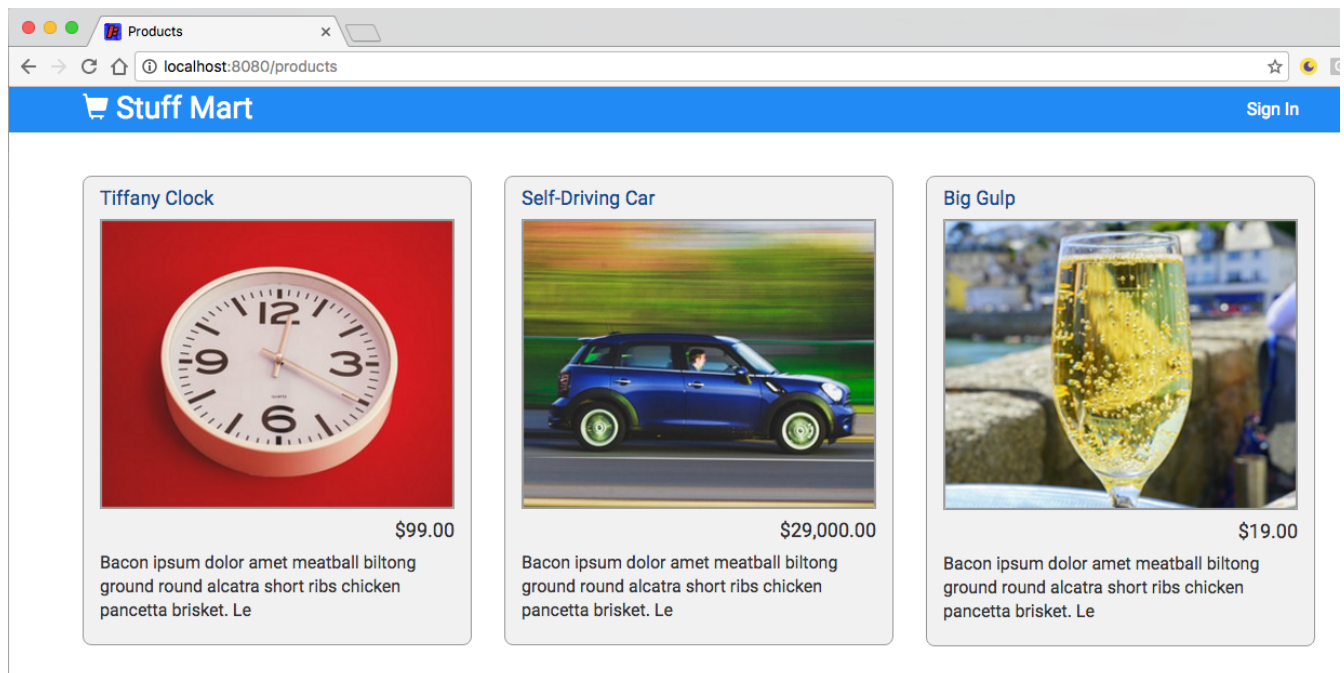
Verify that project moved correctly

Install dependencies

```
cd stuffmart
npm install
```

Start app

```
cd stuffmart
./gradlew appRun
./gradlew.bat appRun (on windows)
```

Open app http://localhost:8080 You should now see this:

Login into the app by clicking the "Sign In" link at the top right of the page. The Sign In Page will have prefilled credentials, just click OK.

Once logged in there will be an "Admin" menu, click the "Products" link you should now see your product-mgr app.

Excercise #6 - Product Service

Instructions

1. open stuffmart app in your IDE

2. generate the product service

3. import Http and inject it in your product service constructor

4. inject Http into your product service constructor

5. import rxjs and implement the query method to retrieve a list products from the product REST service

6. add ProductService to providers in app.module.ts

7. inject ProductService into product-list.component.ts constructor

8. implement the fetchProducts method in product-list.component.ts

9. start the bin/watch script

10. implement ProductService methods for: get, add, update, and delete

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

Step-by-step

Open the stuffmart project in your IDE. If using IntelliJ, open the stuffmart.ipr file.

Generate the product service.

```
cd src/main/js/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 { Http } from '@angular/http';

@Injectable()
export class ProductService {

  constructor(private http: Http) { }

}
```

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 { Http } from '@angular/http';

import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import {Product} from "./product.model";

@Injectable()
export class ProductService {

    baseUrl = '/data/admin/product';

  constructor(private http: Http) { }

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

}
```

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,
    HttpModule
  ],
  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;
  });
}
```

Start the bin/watch script (bin/watch.bat on windows). This will monitor your code and build the app as you make changes.

```
cd stuffmart
bin/watch
```

Your app should now be retrieving products from the product REST service. The products should match those displayed in the front-end app.
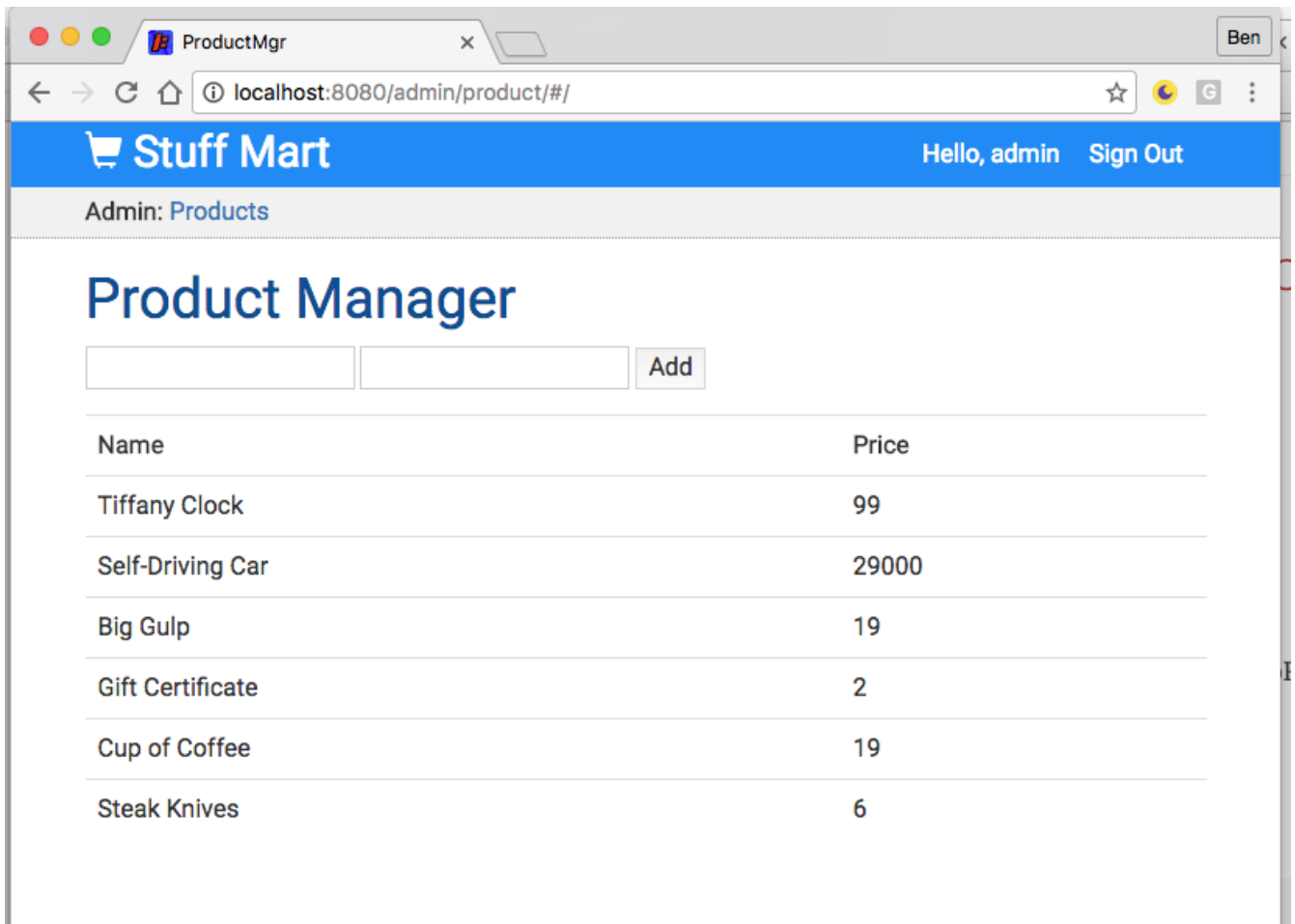
Implement ProductService methods for: get, add, update, and delete

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

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

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

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

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 #7 - Routing

Instructions

1. create app.routes.ts and setup routing for the Product List

2. add routing to your app.module.ts

3. add router-outlet directive to app.component.html

4. configure HashLocationStrategy in app.module.ts

5. verify routing is working

Step-by-step

Create src/app/app.routes.ts file in your IDE and setup routing for the Product List

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

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

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

Add routing to your app.module.ts

```
...
import { routing } from './app.routes';
...
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    routing
  ],
```

Add router-outlet directive to app.component.html (Replace app-list-component directive).

app.component.html

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

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

Configure HashLocationStrategy in app.module.ts

```
...
import { LocationStrategy, HashLocationStrategy  } from '@angular/common';
...
providers: [ ProductService,
      { provide: LocationStrategy, useClass: HashLocationStrategy }
  ],
```

The complete app.module.ts file looks like this.

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

import { AppComponent } from './app.component';
import { ProductListComponent } from './product/product-list.component';
import { ProductAddComponent } from './product/product-add.component';
import {ProductService} from "./product/product.service";

import { routing } from './app.routes';

import { LocationStrategy, HashLocationStrategy  } from '@angular/common';

@NgModule({
  declarations: [
    AppComponent,
    ProductListComponent,
    ProductAddComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    routing
  ],
  providers: [ ProductService,
      { provide: LocationStrategy, useClass: HashLocationStrategy }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```
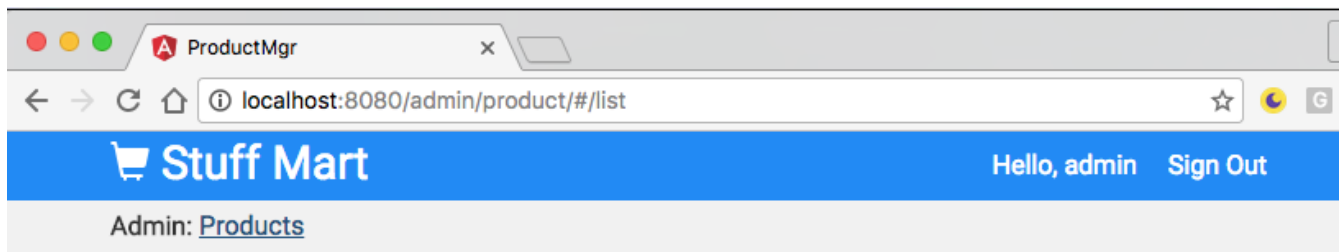
Verify that routing is working. Refresh your app. It should redirect to the url: http://localhost:8080/admin/product/#/list

Exercise #8 - Product Update Component

Instructions

1. create a Product Update Component

2. create a route for the component in app.routes.ts

3. add routerLink in product-list.component.html to link from product name to the Product Update Component

4. verify that the routerLink works

5. create a simple product update form

6. angularize the product update form

7. inject ProductService, ActivatedRoute, and Router into the product-update.component.ts

8. get the route's product id and implement the fetchProduct method

9. wrap the product update form in a div with an *ngIf="product" directive

10. test that the form displays your product name and price

11. implement the updateProduct method

12. verify that the app updates your products and navigates back to the product list

13. add a simple delete form to delete a product

Step-by-step

Create a Product Up

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

Create a route for the component in app.routes.ts

```
...
import {ProductUpdateComponent} from "./product/product-update.component";

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

Add routerLink in product-list.component.html to link from product name to the Product Update Component

```
...
<table class="table">
    <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}}</td>
    </tr>

</table>
```

Verify that the routerLink works. You should be able to navigate to the new component in the application.
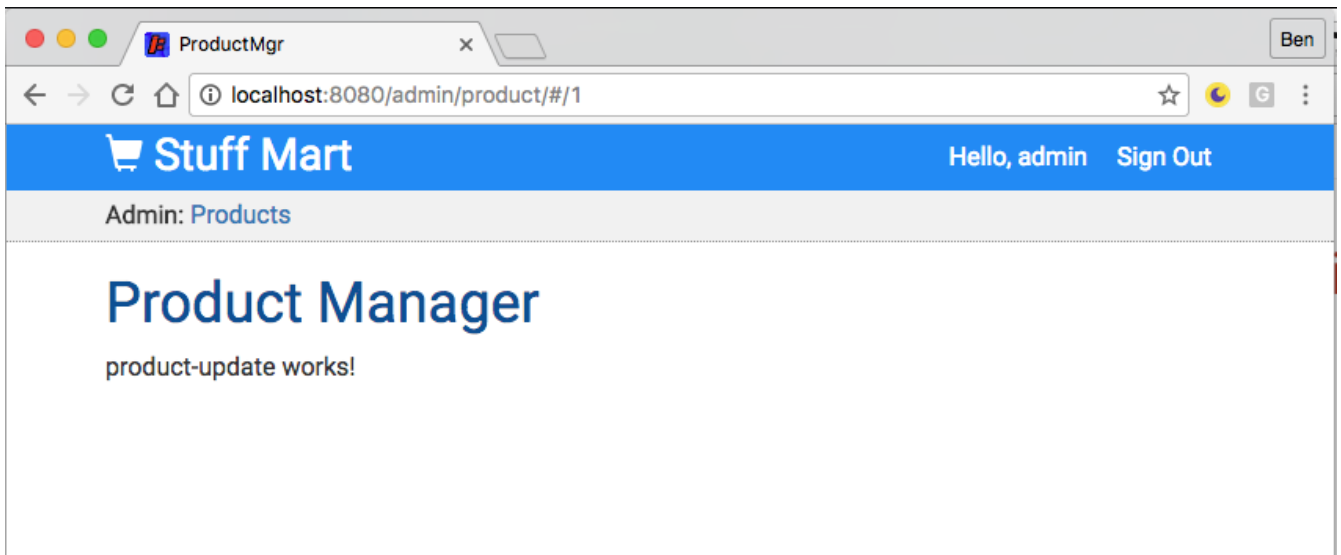
Create a simple product update form.

product-update.component.html

```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>
        <td></td>
        <td>
            <button>Update</button>
        </td>
    </tr>
</table>
</form>
```

Angularize the product update form

```
<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>
        <td></td>
        <td>
            <button [disabled]="!f.valid">Update</button>
        </td>
    </tr>
</table>
</form>
```

Inject ProductService, ActivatedRoute and Router into the product-update.component.ts

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

import { ActivatedRoute, Router } from '@angular/router';
import {ProductService} from "./product.service";

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

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

  ngOnInit() {
  }

}
```

Get the route's product id and implement the fetchProduct method.

```
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;
          });
  }

}
```

Wrap the form in product-update.component.html in a div with an *ngIf="product" directive. This is necesary to prevent errors since the product is loaded asyncronously.

product-update.component.html

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

<form #f="ngForm" (ngSubmit)="updateProduct()">

...

</form>
</div>
```
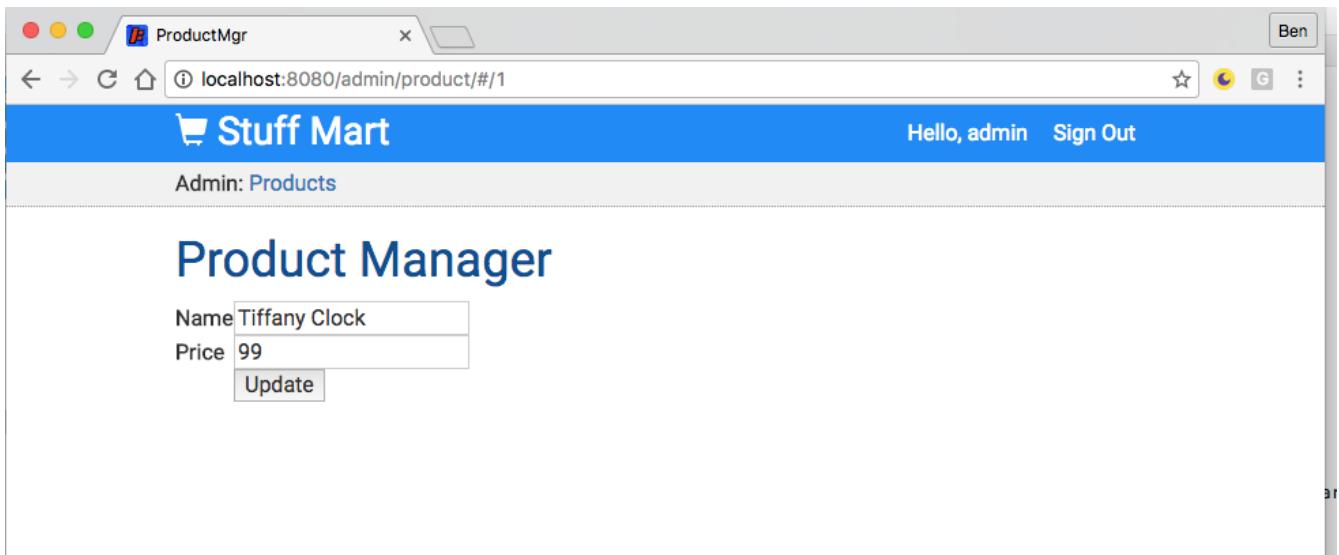
Test that the form displays your product name and price.

Implement the updateProduct method

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

Verify that the app updates your products and navigates back to the product list.

Add a simple delete form to product-update.component delete a product.

product-update.component.html

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

product-update.component.ts

```
    deleteProduct() {

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

        this.productService.delete(this.product)
            .subscribe(r => {
                this.router.navigate(['/', 'list']);
            });
    }
```

Verify that the product is deleted.

The End