Angular 2

By

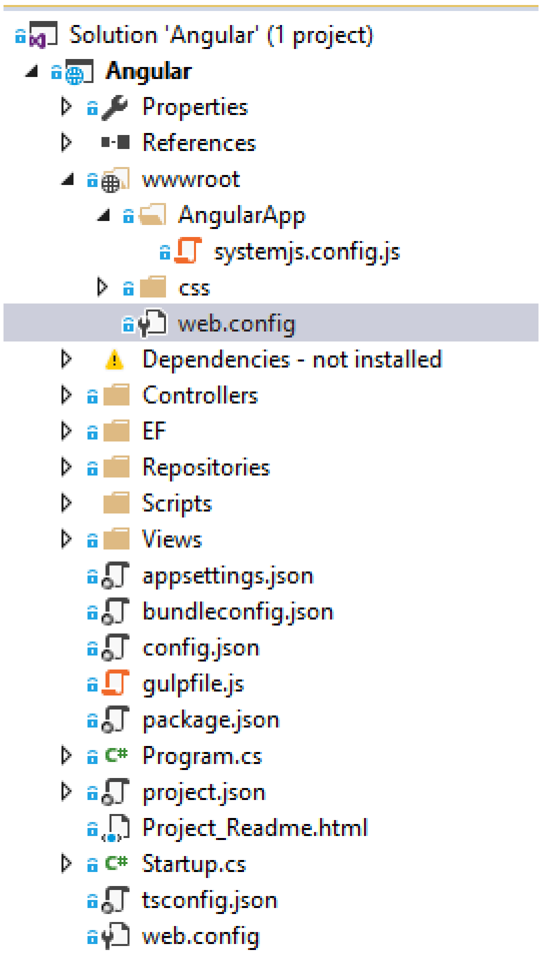
Stefan Popov

Table of Contents

1 Introduction

AngularJS is a complete JavaScript-based open-source front-end web application framework mainly maintained by Google and by a community of individuals and corporations to address many of the challenges encountered in developing single-page applications.

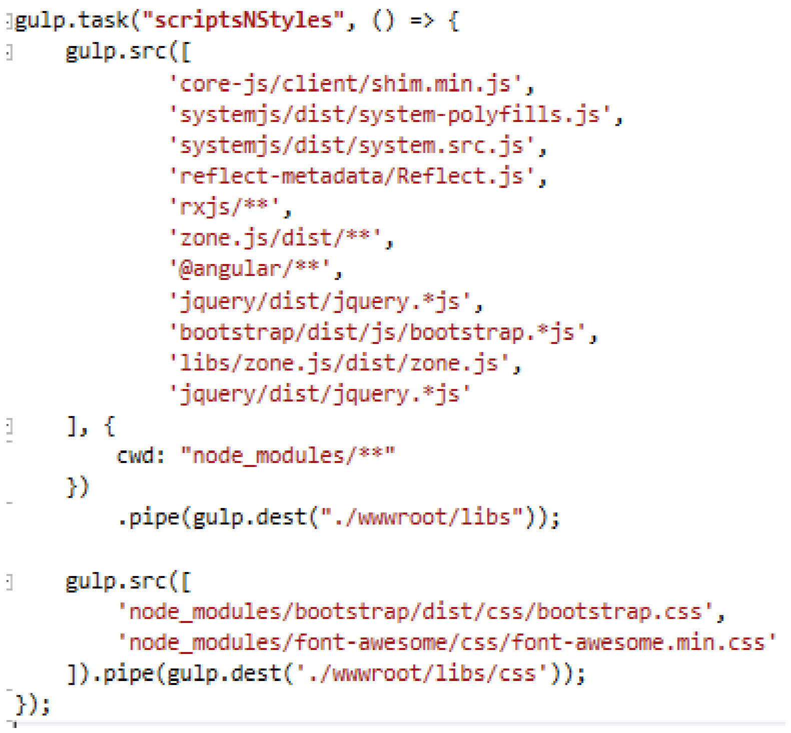
2 Setting up Angular in Visual Studio project



```
Solution 'Angular' (1 project)
  Angular
    Properties
    References
    wwwroot
      AngularApp
        systemjs.config.js
      css
      web.config
    Dependencies - not installed
    Controllers
    EF
    Repositories
    Scripts
    Views
    appsettings.json
    bundleconfig.json
    config.json
    gulpfile.js
    package.json
    Program.cs
    project.json
    Project_Readme.html
    Startup.cs
    tsconfig.json
    web.config
```

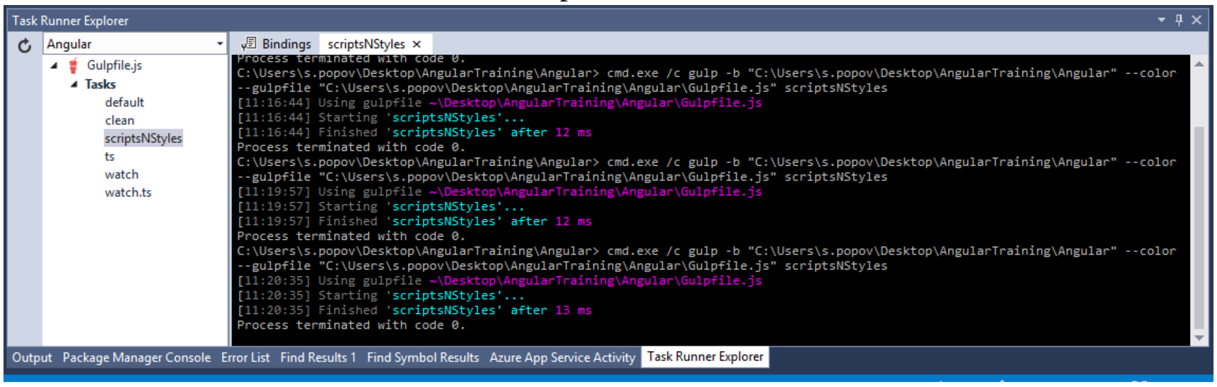
On the image you can see the structure of the project. There are some files that need to be included in the project in order to be able to use Angular and they will be explained one by one.

- **package.json** - identifies npm package dependencies for the project.
  Packages downloaded with npm won't be automatically included in our wwwroot folder. We need to copy them there manually, or we can use gulp or grunt.
- **gulpfile.js** – Gulp is a javascript task runner that lets you automate tasks such as bundling and minifying libraries and stylesheets, refreshing your browser when you save a file, quickly running unit tests, running code analysis… We use it to transfer our dependencies to wwwroot folder.



```
gulp.task("scriptsNStyles", () => {
    gulp.src([
            'core-js/client/shim.min.js',
            'systemjs/dist/system-polyfills.js',
            'systemjs/dist/system.src.js',
            'reflect-metadata/Reflect.js',
            'rxjs/**',
            'zone.js/dist/**',
            '@angular/**',
            'jquery/dist/jquery.*js',
            'bootstrap/dist/js/bootstrap.*js',
            'libs/zone.js/dist/zone.js',
            'jquery/dist/jquery.*js'
    ], {
        cwd: "node_modules/**"
    })
        .pipe(gulp.dest("./wwwroot/libs"));

    gulp.src([
        'node_modules/bootstrap/dist/css/bootstrap.css',
        'node_modules/font-awesome/css/font-awesome.min.css'
    ]).pipe(gulp.dest('./wwwroot/libs/css'));
});
```

This is the task which will transfer all our dependencies to libs folder in wwwroot folder.



```
Process terminated with code 0.
C:\Users\s.popov\Desktop\AngularTraining\Angular> cmd.exe /c gulp -b "C:\Users\s.popov\Desktop\AngularTraining\Angular" --color
--gulpfile "C:\Users\s.popov\Desktop\AngularTraining\Angular\Gulpfile.js" scriptsNStyles
[11:16:44] Using gulpfile ~\Desktop\AngularTraining\Angular\Gulpfile.js
[11:16:44] Starting 'scriptsNStyles'...
[11:16:44] Finished 'scriptsNStyles' after 12 ms
Process terminated with code 0.
C:\Users\s.popov\Desktop\AngularTraining\Angular> cmd.exe /c gulp -b "C:\Users\s.popov\Desktop\AngularTraining\Angular" --color
--gulpfile "C:\Users\s.popov\Desktop\AngularTraining\Angular\Gulpfile.js" scriptsNStyles
[11:19:57] Using gulpfile ~\Desktop\AngularTraining\Angular\Gulpfile.js
[11:19:57] Starting 'scriptsNStyles'...
[11:19:57] Finished 'scriptsNStyles' after 12 ms
Process terminated with code 0.
C:\Users\s.popov\Desktop\AngularTraining\Angular> cmd.exe /c gulp -b "C:\Users\s.popov\Desktop\AngularTraining\Angular" --color
--gulpfile "C:\Users\s.popov\Desktop\AngularTraining\Angular\Gulpfile.js" scriptsNStyles
[11:20:35] Using gulpfile ~\Desktop\AngularTraining\Angular\Gulpfile.js
[11:20:35] Starting 'scriptsNStyles'...
[11:20:35] Finished 'scriptsNStyles' after 13 ms
Process terminated with code 0.
```

In the task runner explorer you should see all the gulp tasks defined in gulpfile.js. If you don’t see it refresh it. Running scriptsNStyles should transfer all the needed dependencies to our wwwroot/libs folder.

- **tsconfig.json** defines how the TypeScript compiler generates JavaScript from the project's files.
- **systemjs.config.js** provides information to a module loader about where to find application modules, and registers all the necessary packages. It also contains other packages that will be needed by later documentation examples.
- **config.json** defines connection string to your local database, you need to change it so it points to your local server
- **seed.sql** contains seed data which you should insert into database once it is created

3 Creating Angular 2 application

3.1 TypeScript and RxJS

Before we start creating Angular 2 application we are going to mention two technologies that Angular 2 heavily depends on. Those are TypeScript and RxJS.

Typescript is a typed superset of JavaScript that compiles to plain JavaScript. Let's analyse that definition. Typed means that Typescript supports types. Superset of JavaScript means that any JavaScript code is valid Typescript code, Typescript just adds types, interfaces, future ES2016+ features (such as annotations/decorators and async/await). Typescript compiles (transpiles) to plain JavaScript code with the help of tools like Gulp, Grunt, WebPack and SystemJS with JSPM. In this course we are going to use Gulp to transpile our Typescript code. Also with typescript we can now get better intellisense in Visual Studio and some more IDEs, but also compile time errors.

Reactive Extensions for JavaScript (RxJS) is a set of libraries for composing asynchronous and event-based programs using observable sequences and fluent query operators that are also knows by Array#extras in JavaScript. Using RxJS developers represent asynchronous data streams with Observables, query asynchronous data streams using our operators and parameterize the concurrency in the asynchronous data streams using Schedulers.

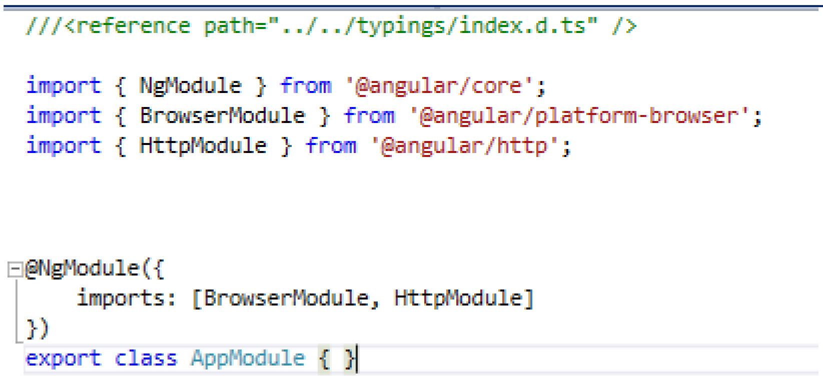
Observables can help manage async data. They are similar to promises but with few differences. Observables emit multiple values over time, while promises once called will always return one value or one error.

3.2 Modules (@NgModule)

From official Angular documentation, Angular modules consolidate components, directives and pipes into cohesive blocks of functionality, each focused on a feature area, application business domain, workflow or common collection of utilities.

Basically modules are classes that are decorated with @NgModule metadata. In @NgModule metadata we can specify which components belong to that module, what other modules that module imports, it can specify what component will be bootstrapped, what are providers that are included in the module…

Every angular application has a root module class, by convention it's called AppModule in a file named app.module.ts.



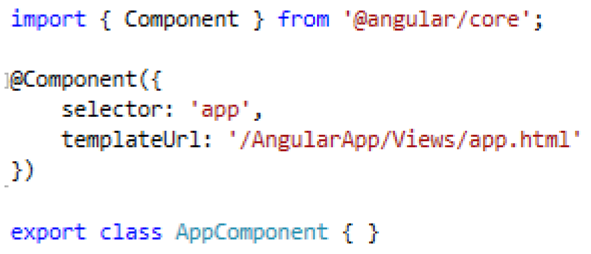
```
///<reference path="../../typings/index.d.ts" />

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



@NgModule({
    imports: [BrowserModule, HttpModule]
})
export class AppModule { }
```

This is our root module, we should create a new file in Scripts/app and call it app.module.ts.

In angular 2 everything is imported from modules. As you can see we are importing NgModule from '@angular/core' module which is decorator metadata. BrowserModule which registers critical application service providers, it also includes common directives like NgIf and NgFor which become immediatel visible and usable in any of this modules component templates and HttpModule which enables us to create http requests which we are going to use later.

3.3 Components (@Component)

Like module component is also a class, but it has different decorator metadata @Component. We should create new typescript file in Scripts/app and call it app.component.ts. This will be our root component, and by convention we should call it app.



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

@Component({
    selector: 'app',
    templateUrl: '/AngularApp/Views/app.html'
})

export class AppComponent { }
```

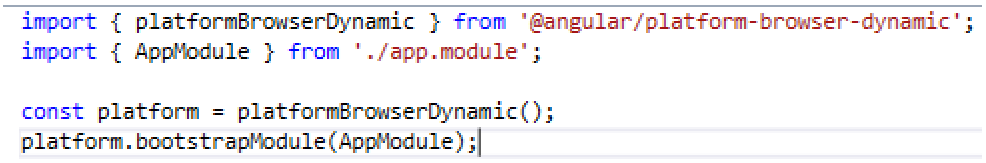
Component is directive that is responsible for behaviour of part of our page. First we import component metadata from @angular/core and create class App. After that we add decorator @Component and set selector and templateUrl properties. **Selector property** specifies a simple CSS selector for an HTML element that represents the component and which will be controlled by component. **TemplateUrl property** specifies path to the view which will be rendered into the element which component controls.

Remember that decorator is there just to add some information about the class. So this won't be component and visual studio will show error if there is no class defined.

3.4 Bootstrapping Angular application

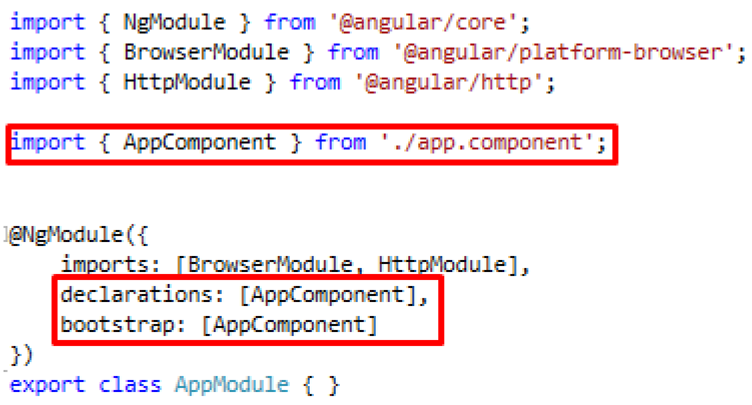
Now that we have our root module and root component, we need to tell Angular to use those to start up Angular application, process known as bootstrapping.

First we need to create new typescript file called main.ts. Again by convention file, that bootstraps application, should be named main.



```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

We are using platformBrowserDynamic because our application is running directly in browser. We also need to import our root module. Bootstrapping module is pretty easy, as you can see on image above. Now that we have told Angular which will be our root module, we still need to tell our root module which will be root component.



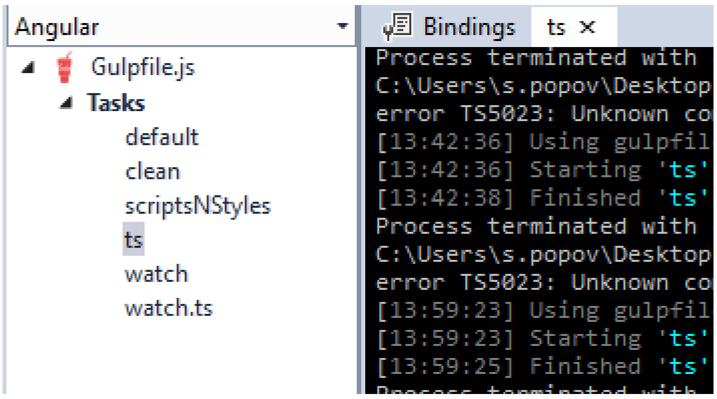
```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpModule } from '@angular/http';

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


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

We do it by specifying bootstrap property inside of @NgModule decorator and passing the root component which will be called when the module bootstraps. We have passed App component to bootstrap property after we have imported it. **Declarations** property tells angular which components belong to that module.

3.5 Starting the application

First of all, you have probably noticed export keyword next to classes, we use that keyword to tell angular that the class will be available to other angular components outside of component/module in which it has been defined.

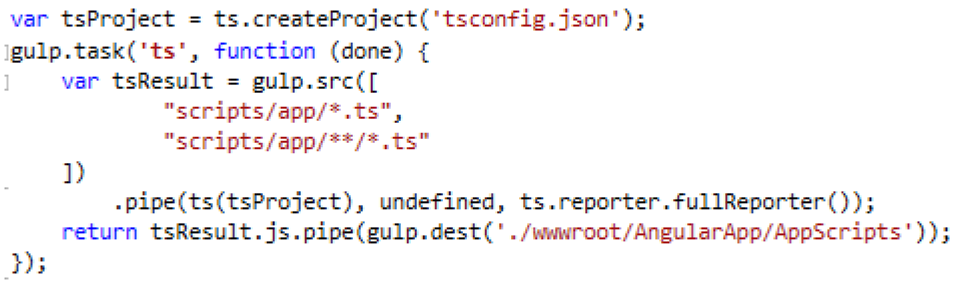
We are not quite ready to start up our application just yet. First you need to run scriptsNStyles from task runner explorer if you already didn't do that. That should transfer all needed dependencies to wwwroot/libs folder.



```
Angular
◢ Gulpfile.js
  ◢ Tasks
      default
      clean
      scriptsNStyles
      ts
      watch
      watch.ts
```

```
Bindings   ts ×
Process terminated with
C:\Users\s.popov\Desktop
error TS5023: Unknown co
[13:42:36] Using gulpfil
[13:42:36] Starting 'ts'
[13:42:38] Finished 'ts'
Process terminated with
C:\Users\s.popov\Desktop
error TS5023: Unknown co
[13:59:23] Using gulpfil
[13:59:23] Starting 'ts'
[13:59:25] Finished 'ts'
```

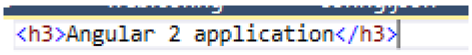
After we have run scriptsNStyles task, we should run ts task, this task should transpile all files from Scripts folder and move them to wwwroot/AngularApp/AppScripts folder because in ASP.NET 5 projects, only files in wwwroot are visible in project. On the image below you can see how the task looks like in gulpfile.js.



```
var tsProject = ts.createProject('tsconfig.json');
gulp.task('ts', function (done) {
    var tsResult = gulp.src([
            "scripts/app/*.ts",
            "scripts/app/**/*.ts"
    ])
        .pipe(ts(tsProject), undefined, ts.reporter.fullReporter());
    return tsResult.js.pipe(gulp.dest('./wwwroot/AngularApp/AppScripts'));
});
```

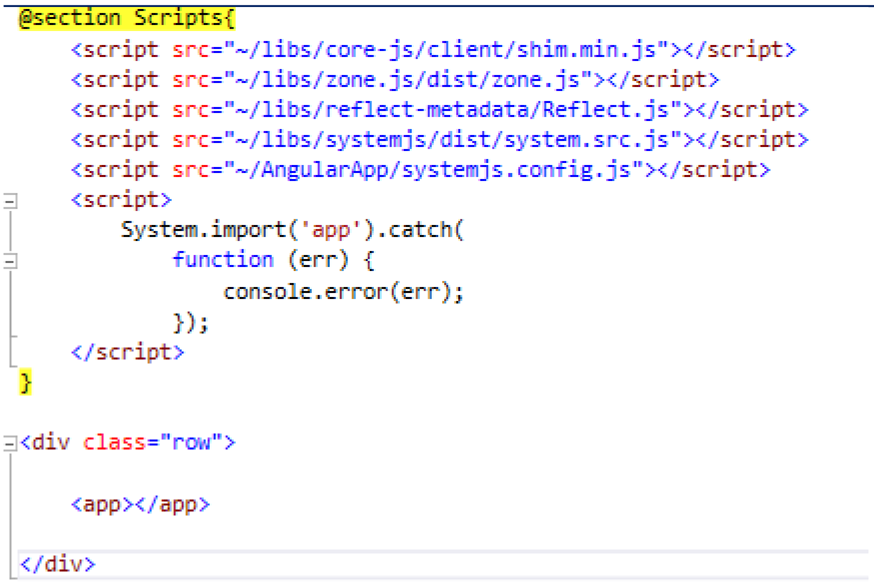
You need to run this task whenever you change something in your typescript files, so the changes are transferred to wwwroot folder.

And finally we should create our view in wwwroot/AngularApp/Views called app.html. For now we should just add h3 with some text, like shown below.



```
<h3>Angular 2 application</h3>
```

And Views/Home/Index.cshtml should look like on image below below.



```
@section Scripts{
    <script src="~/libs/core-js/client/shim.min.js"></script>
    <script src="~/libs/zone.js/dist/zone.js"></script>
    <script src="~/libs/reflect-metadata/Reflect.js"></script>
    <script src="~/libs/systemjs/dist/system.src.js"></script>
    <script src="~/AngularApp/systemjs.config.js"></script>
    <script>
        System.import('app').catch(
            function (err) {
                console.error(err);
            });
    </script>
}

<div class="row">

    <app></app>

</div>
```

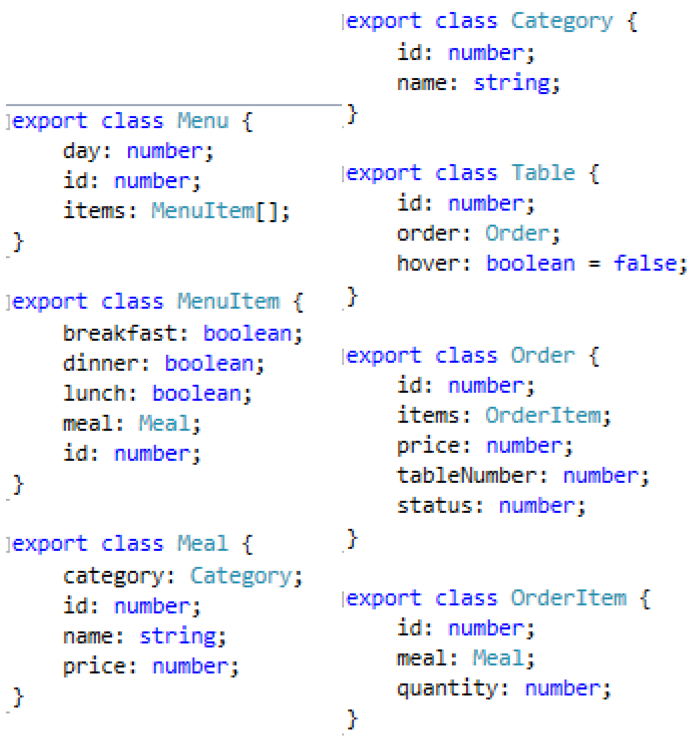
‘app’ from System.import(‘app’) is referencing app from packages in systemjs.config.js.

If we start the application now you should see text from app.html, if it doesn’t show check the console of your browser.

4 Application Overview

Until this point we have been only setting up Angular and created root module and root component which are same for every Angular application. From now on, we are going to create application for managing orders in restaurant.

In the image below you can see our domain classes which we are going to use in the tutorial.



```
export class Menu {
    day: number;
    id: number;
    items: MenuItem[];
}

export class MenuItem {
    breakfast: boolean;
    dinner: boolean;
    lunch: boolean;
    meal: Meal;
    id: number;
}

export class Meal {
    category: Category;
    id: number;
    name: string;
    price: number;
}
```

```
export class Category {
    id: number;
    name: string;
}

export class Table {
    id: number;
    order: Order;
    hover: boolean = false;
}

export class Order {
    id: number;
    items: OrderItem;
    price: number;
    tableNumber: number;
    status: number;
}

export class OrderItem {
    id: number;
    meal: Meal;
    quantity: number;
}
```

We store our domain classes in Scripts/app/shared/domain.classes.ts.

5 Routing

Now that we have our application up and running, we need to have a way to move from one view to the other. Angular is famous for its single page applications and routing is there to make it happen.

First we are going to create component, which we will use in our routing example, in Scripts/app/menu and we should call it menu.component.ts.

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

@Component({
    templateUrl: '/AngularApp/Views/menu.html'
})
export class MenuComponent {

}
```

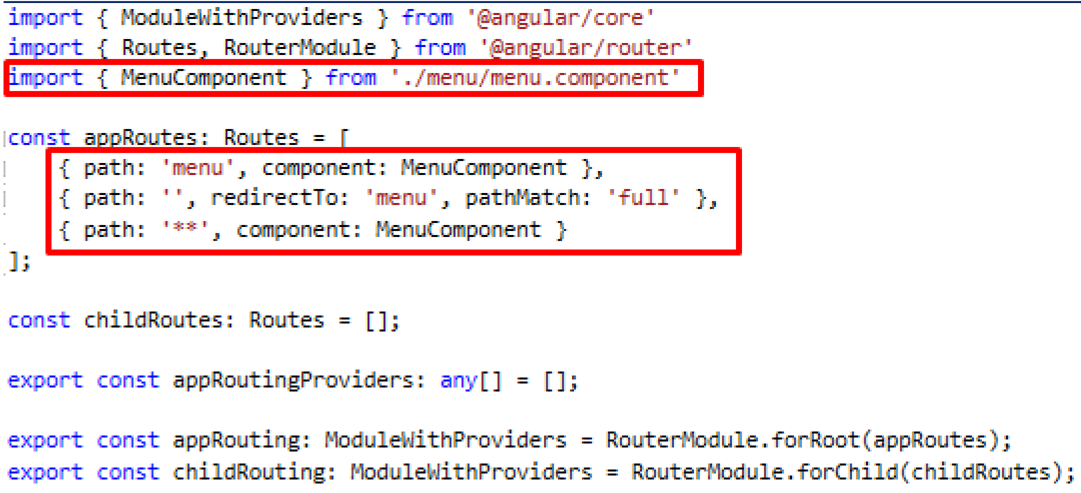
By convention we should define routing in separate file. We should create new file in Scripts/app called app.routing.ts with the following lines of code.

```
import { ModuleWithProviders } from '@angular/core'
import { Routes, RouterModule } from '@angular/router'

const appRoutes: Routes = [];

const childRoutes: Routes = [];

export const appRoutingProviders: any[] = [];

export const appRouting: ModuleWithProviders = RouterModule.forRoot(appRoutes);
export const childRouting: ModuleWithProviders = RouterModule.forChild(childRoutes);
```

ModulWithProviders is wrapper around a module that also includes providers, we will define routes in Rotues, and RouterModule extends the module with router. As you can see we are defining 3 constants appRoutingProviders, routing and childRouting, which we will include in our module later.

You should use RouterModule.forRoot only for root application module (AppModule), calling it in any other module, particularly in lazy loaded module is contrary to the intent and is likely to produce a runtime error, from angular documentation.

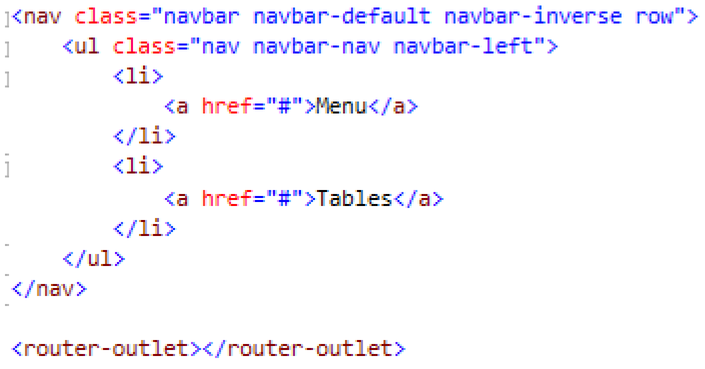
Since we are adding routing to our root module, we will use appRoutes constant.



```
import { ModuleWithProviders } from '@angular/core'
import { Routes, RouterModule } from '@angular/router'
import { MenuComponent } from './menu/menu.component'

const appRoutes: Routes = [
    { path: 'menu', component: MenuComponent },
    { path: '', redirectTo: 'menu', pathMatch: 'full' },
    { path: '**', component: MenuComponent }
];

const childRoutes: Routes = [];

export const appRoutingProviders: any[] = [];

export const appRouting: ModuleWithProviders = RouterModule.forRoot(appRoutes);
export const childRouting: ModuleWithProviders = RouterModule.forChild(childRoutes);
```

We still haven't created menu component, we will do it in few moments. First route we have defined will load MenuComponent class from menu.component every time we go to '/menu'. Second route will take us to '/menu' every time we visit '/', note that pathMatch: 'full' is there to ensure that only when we got to '/' we will be redirected to '/menu'. If we didn't specify pathMatch, for example if we would visit '/someroute/' we would be redirected to 'menu'. Last route '**' is wildcard route, meaning that it will match anything.

One thing you should know is that you should specify your more specific routes first, and more general routes later, because redirect happens when the first route is matched.



```
<nav class="navbar navbar-default navbar-inverse row">
    <ul class="nav navbar-nav navbar-left">
        <li>
            <a href="#">Menu</a>
        </li>
        <li>
            <a href="#">Tables</a>
        </li>
    </ul>
</nav>

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

Above you can see what we need to add in our app.html. It is just simple HTML mark-up with few Bootstrap classes. Router outlet element serves as a place where Angular Router will place views when we navigate to them.

There are few more things we need to do before routing starts working. We need to tell angular what the base route for our app is. We do that by adding base element in the head section of our application. In the file Views/Shared/_Layout.cshtml we are going to add that base element.

```
<!DOCTYPE html>

<html>
<head>
    <base href="/">
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" type="text/css" href="~/libs/css/bootstrap.css" />
    <link rel="stylesheet" type="text/css" href="~/css/Site.css" />
    <link rel="stylesheet" type="text/css" href="~/libs/css/font-awesome.min.css" />
</head>
```

As you can see we are providing href attribute, so you can set here whatever route you want and this route will show up whenever we are using Angular router, before the routes we specified. For example, if we had set href of base element to '/Restaurant/' and if we started up our application, we would be redirected to '/Restaurant/menu'.

And the last thing we need to do to make routing work is to include it in our module.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpModule } from '@angular/http';
import { appRouting, appRoutingProviders } from './app.routing';

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

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

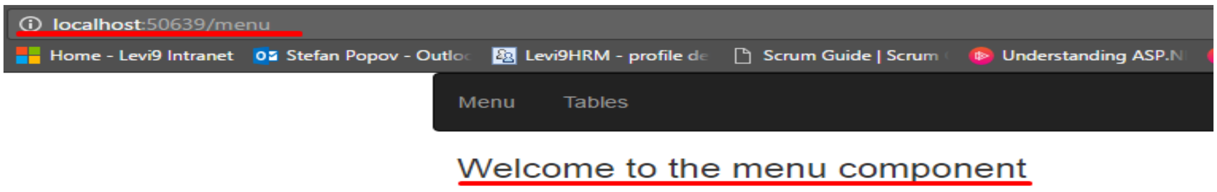
Since we are setting up routes for root module, we are going to use appRouting constant from app.routing.ts, we can remove childRouting from the same file, since we are not going to use them, and if we create routes for child modules, they will be defined in separate file.

In providers part of NgModule we register services that are going to be available in the module.

One more thing we need to do is to create wwwroot/AngularApp/Views/menu.html. For now we are just going to add an h3, so we can be sure our rooting works.

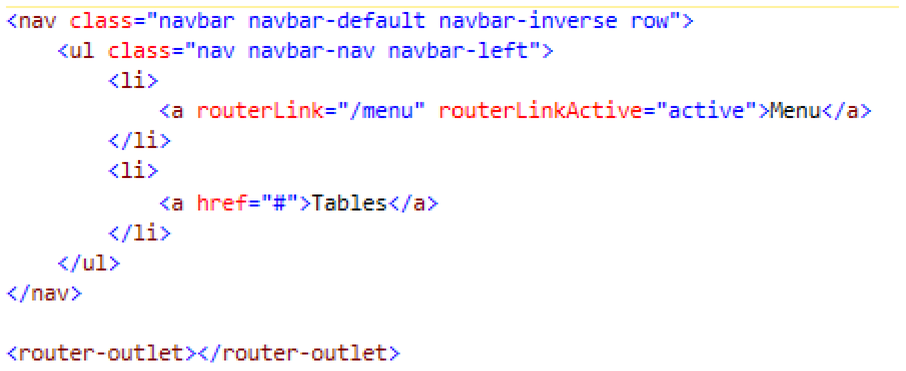
```
<h3>Welcome to the menu component</h3>
```

If we run the application now, we should be redirected to /menu and the h3 should show below the menu.



Welcome to the menu component

One more thing is left about routing, that is the way to pass routes to link, we use routerLink attribute for that, we only need to pass the route to which we want to redirect.



```
<nav class="navbar navbar-default navbar-inverse row">
    <ul class="nav navbar-nav navbar-left">
        <li>
            <a routerLink="/menu" routerLinkActive="active">Menu</a>
        </li>
        <li>
            <a href="#">Tables</a>
        </li>
    </ul>
</nav>

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

6 Services

6.1 Dependency Injection in Angular

Dependency injection is a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

Angular ships with its own dependency injection framework. This framework can also be used as a standalone module by other applications and frameworks. We don't need to create an Angular injector, Angular creates an application wide injector for us during bootstrap process. What we need to do is to configure injector by registering the providers that create the services our application requires.

A provider registered in an NgModule is registered in the root injector, which means that every provider registered within an NgModule will be accessible in the entire application. If we register provider in an application component it will be available only on that component and all its children.

One thing you should know that all services are singletons across components. Which means that you should register provider at component that is on the highest level in the hierarchy tree. There can be only one instance of a service type in a particular injector, but we can have multiple injectors operating at different levels of the application's component tree.

@Injectable() metadata marks a class as available to an injector for instantiation. Generally speaking, an injector will report an error when trying to instantiate a class that is not marked as @Injectable().

Angular team recommends adding @Injectable() to every service class, even those that don't have dependencies and do not technically require it for two reasons:

- Future proofing: no need to remember @Injectable() when we add a dependency later
- Consistency: all services follow the same rules, and we don't have to wonder why a decorator is missing.

@Component decorator is a subtype of Injectable, so we don't need to decorate our components with @Injectable decorator when they have dependencies injected.

A provider provides the concrete, runtime version of a dependency value. The injector relies on providers to create instance of the services that the injector injects into components and other services.

You can read more about dependency injection and to understand the angular hierarchical structure in Angular documentation.

6.2 RxJS – Reactive Extensions for JavaScript

The Reactive Extensions for JavaScript (RxJS) is a set of libraries for composing asynchronous and event-based programs using observable sequences and fluent query operators also known as Array#extras in JavaScript. Using RxJS, developers represent asynchronous data streams with Observables, query asynchronous data streams using operators and parametrize concurrency in the asynchronous data streams using Schedulers. Simply put, RxJS = Observables + Operators + Schedulers.

Observables represent any set of values over any amount of time.

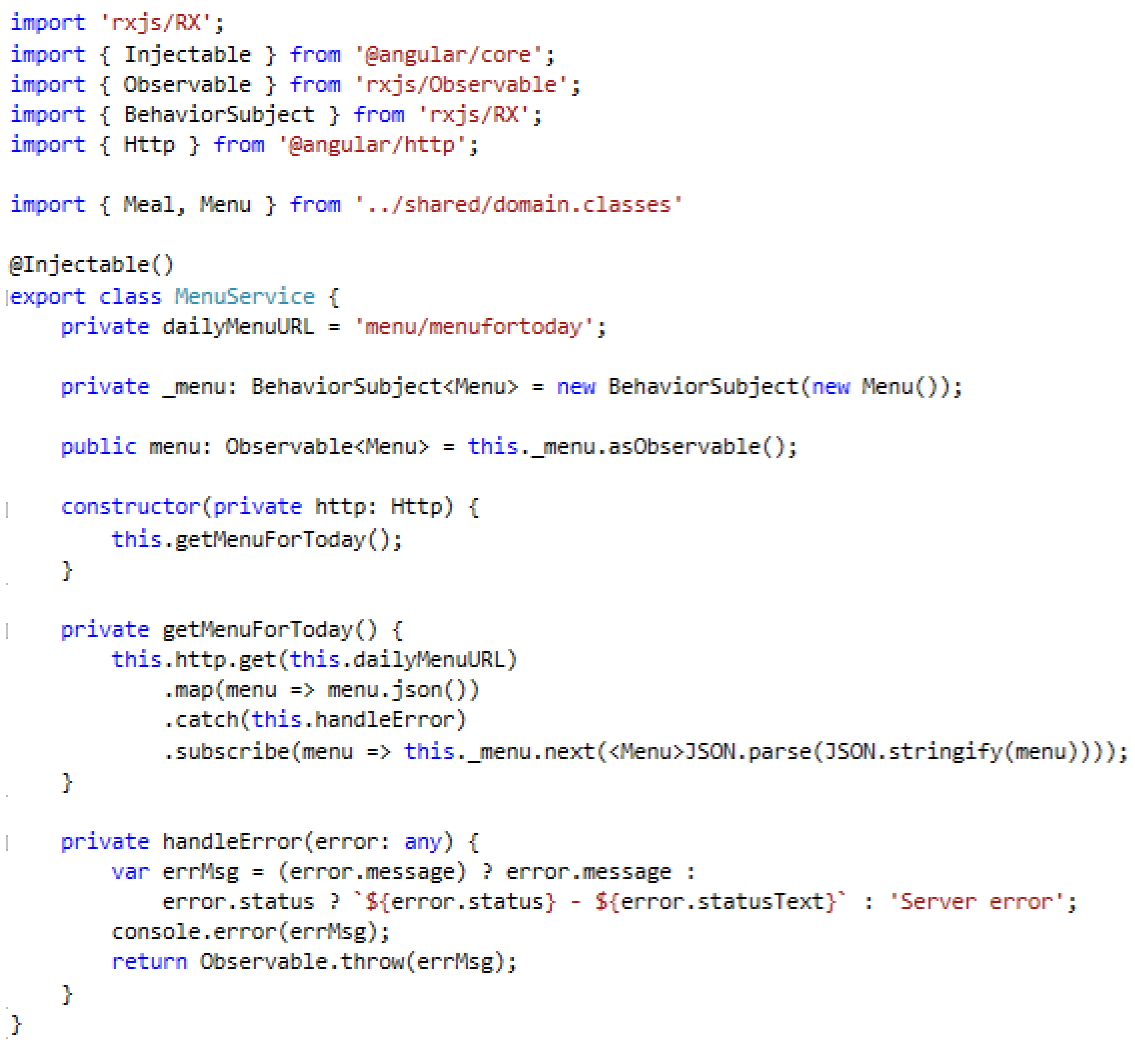
Subject is a sort of bridge or proxy that is available in some implementations of ReactiveX that acts both as an observer and as an Observable. Because it is an observer, it can subscribe to one or more Observables, and because it is observable, it can pass through the items it observes by reemitting them, and it can also emit new items.

In this tutorial we are going to use BehaviorSubject. When observer subscribes to BehaviorSubject, it begins emitting the item most recently emitted by the source Observable and then continues to emit any other items emitted later by the source Observable. If the source Observable terminates with an error, the BehaviourSubject will not emit any items to subsequent observers, but will simply pass along the error notification from the source Observable.

You can find more information about subjects and different kinds of subjects on this link.

6.3 Creating a service

We need to fetch menu for today to show it in our MenuComponent. We need to fetch it from /menu/menuforoday. First we need to create a MenuService class as shown in the image below at Scripts/app/menu in the file menu.service.ts.



```
import 'rxjs/RX';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { BehaviorSubject } from 'rxjs/RX';
import { Http } from '@angular/http';

import { Meal, Menu } from '../shared/domain.classes'

@Injectable()
export class MenuService {
    private dailyMenuURL = 'menu/menufortoday';

    private _menu: BehaviorSubject<Menu> = new BehaviorSubject(new Menu());

    public menu: Observable<Menu> = this._menu.asObservable();

    constructor(private http: Http) {
        this.getMenuForToday();
    }

    private getMenuForToday() {
        this.http.get(this.dailyMenuURL)
            .map(menu => menu.json())
            .catch(this.handleError)
            .subscribe(menu => this._menu.next(<Menu>JSON.parse(JSON.stringify(menu))));
    }

    private handleError(error: any) {
        var errMsg = (error.message) ? error.message :
            error.status ? `${error.status} - ${error.statusText}` : 'Server error';
        console.error(errMsg);
        return Observable.throw(errMsg);
    }
}
```

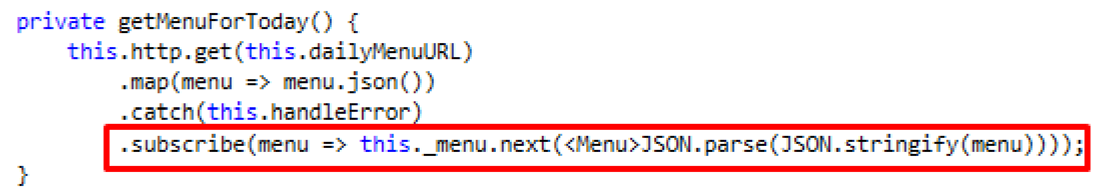
We are importing everything from ‘rxjs/RX’, because we need operators, Injectable and Observable, which we already covered in chapter 6.1, BehaviorSubject which we covered in chapter 6.2, Http, Meal and Menu.

Because we have imported HttpModule in AppModule angular injector already has reference to Http service, so we just need to pass it through the constructor.

As you can see we have private field _menu which is BehaviorSubject, where we set starting value to new Menu, and we have public observable which is just our BehaviourSubject converted to observable.

We use http service to get menu from given url, we map the response to json with map operator. If there was an error during this process we handle it through handleError method, notice that this method also returns observable. Also we need to use ` in order to interpolate string – insert JavaScript code inside of string.

Http calls return observables that is why we need to subscriber to it.



```
private getMenuForToday() {
    this.http.get(this.dailyMenuURL)
        .map(menu => menu.json())
        .catch(this.handleError)
        .subscribe(menu => this._menu.next(<Menu>JSON.parse(JSON.stringify(menu))));
}
```

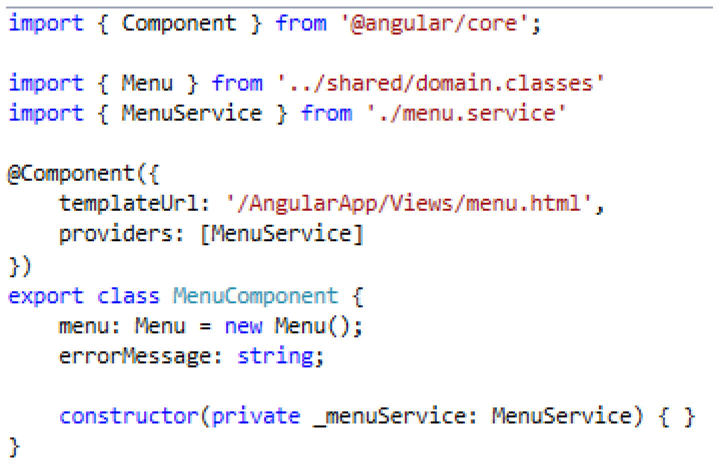
In order to cast it to desired type in typescript we need to first stringify it then parse it as JSON and finally cast it to type we want. One of the reasons we've used BehaviorSubject instead of just observables, since all Http calls return observable, is that it has method next() with which we can notify observers that there is new value in observable and that they should take it. We will see later that we can also access the current value of BehaviorSubject.

Also services don't have ngOnInit method, so everything should be initialized in constructor, that is why we are calling getMenuForToday() in constructor.

6.4 Using service in components (providers in @Component decorator)

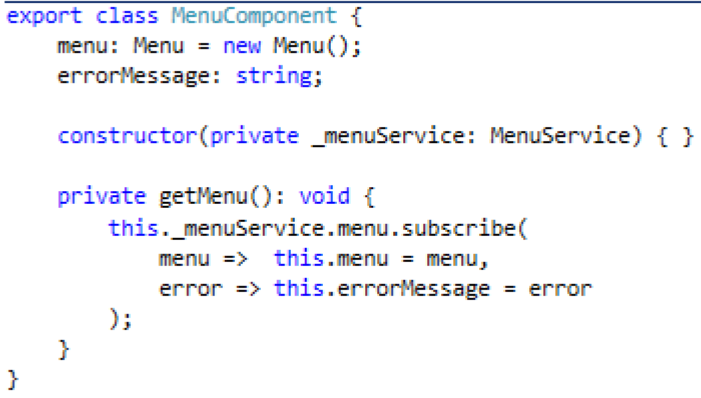
We need to import MenuService in our MenuComponent in order to use it, also we need to tell our MenuComponent that it is going to use MenuService, we do that by passing MenuService to providers array inside of @Component decorator which will register the service inside of injector. We could pass MenuService in our AppModule, but I don't need that service included in whole app, I just want it to be available in MenuComponent and its child components, which Angular's hierarchical injector allows us if we pass service inside providers array in component.

Other than that we need menu field where we will store menu we get from service, errorMessage field, and we need to tell angular that it should inject service in the component by passing it in constructor.



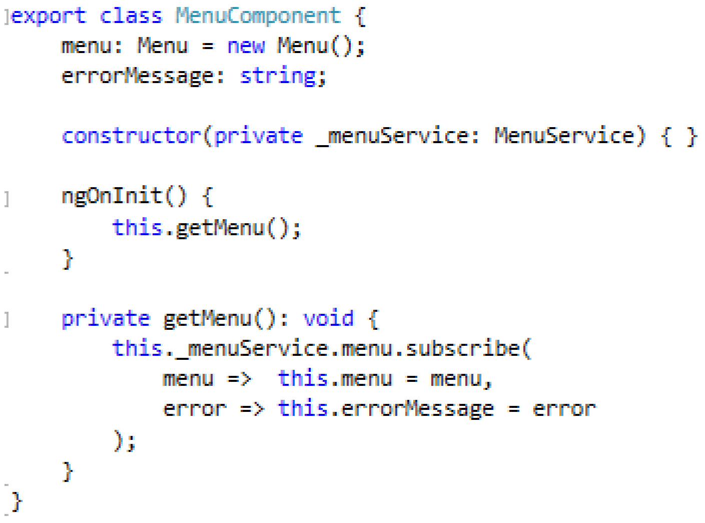
```
import { Component } from '@angular/core';

import { Menu } from '../shared/domain.classes'
import { MenuService } from './menu.service'

@Component({
    templateUrl: '/AngularApp/Views/menu.html',
    providers: [MenuService]
})
export class MenuComponent {
    menu: Menu = new Menu();
    errorMessage: string;

    constructor(private _menuService: MenuService) { }
}
```

In order to get menu from MenuService we only need to subscribe to menu observable from MenuService. Once we do that if menu changes in menu observable from MenuService, menu field from MenuComponent will automatically be updated. We need to create method that will subscribe to menu Observable, like shown below.



```
export class MenuComponent {
    menu: Menu = new Menu();
    errorMessage: string;

    constructor(private _menuService: MenuService) { }

    private getMenu(): void {
        this._menuService.menu.subscribe(
            menu =>  this.menu = menu,
            error => this.errorMessage = error
        );
    }
}
```

Finally we need to call that function when component initializes as you can see on image below.



```
export class MenuComponent {
    menu: Menu = new Menu();
    errorMessage: string;

    constructor(private _menuService: MenuService) { }

    ngOnInit() {
        this.getMenu();
    }

    private getMenu(): void {
        this._menuService.menu.subscribe(
            menu =>  this.menu = menu,
            error => this.errorMessage = error
        );
    }
}
```

7 Creating menu view

7.1 Data bindings

There are 4 types of data binding in Angular 2:

- Interpolation – one way data binding from component to view, we use {{}} for interpolation
- Property binding – one way data binding form component to the view, we use [ ] for property binding
- Event binding – one way data binding from view to component, we use ( ) for event binding.
- ngModel – two way data binding, we use [( )] for ngModel.

7.2 Structural directives

Directives are called structural if they change the structure of the DOM. For example *ngFor is structural directive because it repeats the element as many times as there are elements in array. *ngIf is structural directive that adds or removes element from DOM based on condition.

7.3 Creating menu overview (interpolation and *ngFor)

We need to show all meals that are on the menu. We will use table for this with some bootstrap classes for styling. As you can see on the image below we are using *ngFor structural directive to create one row in table for every menu item in array.

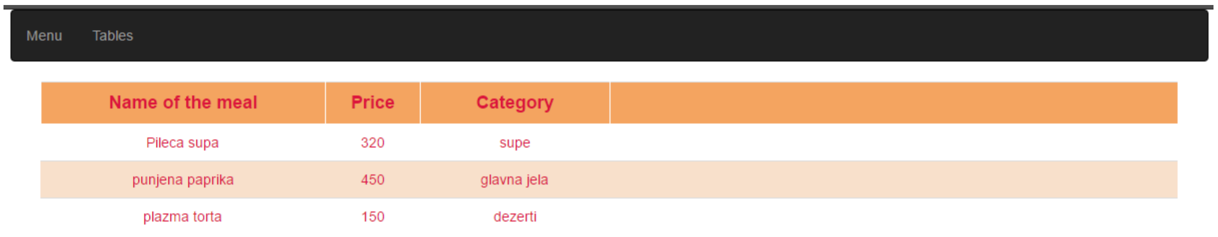
The let statement declares a block scope local variable, optionally initializing it to a value. Variables declared by let have as their scope the block in which they are defined, as well as in any contained sub-blocks. In this way, let works very much like var, the mein difference is that the scope of a var variable is entire enclosing function.

Menu from "let item of menu.items" represent menu field from MenuComponent which we have filled with data from MenuService.

As you can see on the image we are using interpolation for showing the data from our component. For every item in menu.items we are meal name, meal price and category name of the given meal.

```
<div class="col-xs-12">
    <table class="table table-striped" id="MenuItemsTable">
        <tr>
            <th class="col-xs-3">Name of the meal</th>
            <th class="col-xs-1">Price</th>
            <th class="col-xs-2">Category</th>
            <th class="col-xs-6"></th>
        </tr>
        <tr *ngFor="let item of menu.items">
            <td class="col-xs-3">{{ item.meal.name }}</td>
            <td class="col-xs-1">{{ item.meal.price }}</td>
            <td class="col-xs-2">{{ item.meal.category.name }}</td>
            <td class="col-xs-6">
            </td>
        </tr>
    </table>
</div>
```

After we save project, run ts task from task runner and run the project, we should see the following screen (note that the data shown might be different for you).



Name of the meal	Price	Category	
Pileca supa	320	supe	
punjena paprika	450	glavna jela	
plazma torta	150	dezerti	

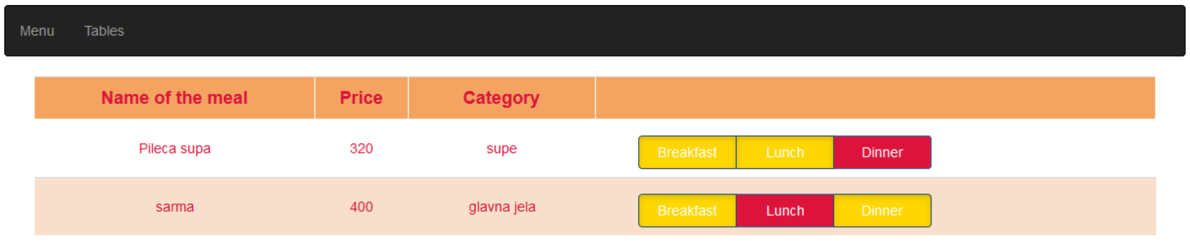
7.4 Adding buttons for breakfast, lunch and dinner (Property binding and event binding)

```
<tr *ngFor="let item of menu.items">
    <td class="col-xs-3">{{ item.meal.name }}</td>
    <td class="col-xs-1">{{ item.meal.price }}</td>
    <td class="col-xs-2">{{ item.meal.category.name }}</td>
    <td class="col-xs-6">
        <div class="col-xs-8">
            <div class="btn-group button-group-meal">
                <button onfocus="this.blur();"
                        [ngClass]="{'btn':true, 'btn-primary':true,'active': item.Breakfast}"
                        (click)="changeStateOfMeal(item, 'breakfast')">
                    <span>Breakfast</span>
                </button>
                <button onfocus="this.blur();"
                        [ngClass]="{'btn':true, 'btn-primary':true,'active': item.Lunch}"
                        (click)="changeStateOfMeal(item, 'lunch')">
                    <span>Lunch</span>
                </button>
                <button onfocus="this.blur();"
                        [ngClass]="{'btn':true, 'btn-primary':true, 'active': item.Dinner}"
                        (click)="changeStateOfMeal(item, 'dinner')">
                    <span> Dinner </span>
                </button>

            </div>
        </div>
    </td>
</tr>
```

Now I want three buttons in the fourth column which will show weather the meal is served for breakfast, lunch or dinner. Also I should be able to change weather the meal is served for breakfast, lunch or dinner or not.

As you can see on the image, three buttons are created. Onfocus="this.blur()" is there to remove focus state because of some bootstrap styles which we don't need.

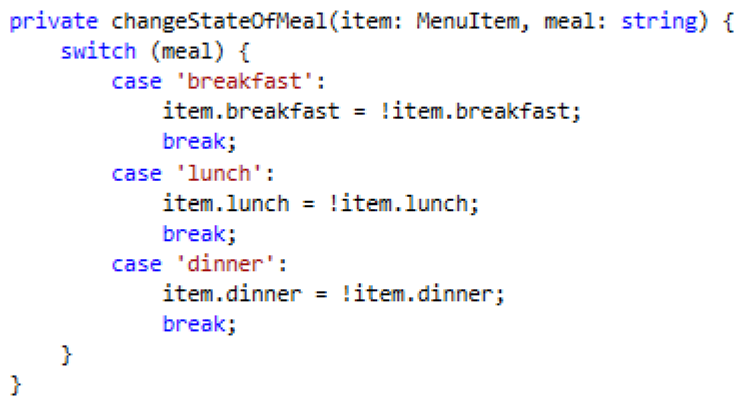
As you can see, we are using property binding with ngClass directive. We are adding bootstrap classes btn and btn-primary to every button, and class active only if corresponding value of item.Breakfast, item.Lunch or item.Dinner is true. Now if we run the project we should see table like on the image.



Name of the meal	Price	Category	
Pileca supa	320	supe	Breakfast Lunch Dinner
sarma	400	glavna jela	Breakfast Lunch Dinner

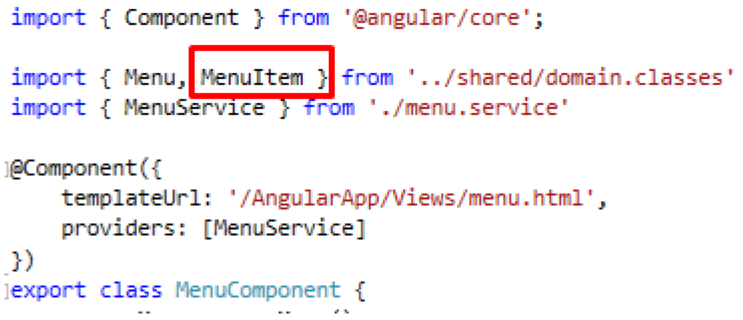
Now we have buttons and different states shown, but we need to enable changing of states. If you see the code above we have used event binding (click)="changeStateOfMeal(item, 'breakfast')". We can bind to any event like click, mousedown, submit, blur, change…

Now we need to create method changeStateOfMeal in MenuComponent.



```
private changeStateOfMeal(item: MenuItem, meal: string) {
    switch (meal) {
        case 'breakfast':
            item.breakfast = !item.breakfast;
            break;
        case 'lunch':
            item.lunch = !item.lunch;
            break;
        case 'dinner':
            item.dinner = !item.dinner;
            break;
    }
}
```

Also MenuItem should be imported.



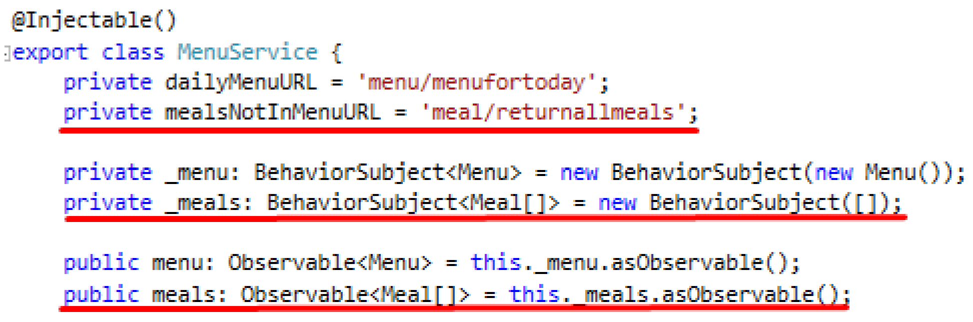
```
import { Component } from '@angular/core';

import { Menu, MenuItem } from '../shared/domain.classes'
import { MenuService } from './menu.service'

@Component({
    templateUrl: '/AngularApp/Views/menu.html',
    providers: [MenuService]
})
export class MenuComponent {
```

Now whenever we click on a button, we should toggle the state for breakfast, lunch or dinner.

7.5 Adding and removing meal from menu (*ngIf)

We should now add functionality to add and remove meals from menu. First we are going to add a meal. We should create dropdown with all meals that are not in the menu. For that we need to get those meals from our web api.

We need to add new observable to the MenuService which will contain all meals that are not in current menu.



```
@Injectable()
export class MenuService {
    private dailyMenuURL = 'menu/menufortoday';
    private mealsNotInMenuURL = 'meal/returnallmeals';

    private _menu: BehaviorSubject<Menu> = new BehaviorSubject(new Menu());
    private _meals: BehaviorSubject<Meal[]> = new BehaviorSubject([]);

    public menu: Observable<Menu> = this._menu.asObservable();
    public meals: Observable<Meal[]> = this._meals.asObservable();
```

Once we have created that we need to write method that will populate observable.

```
private returnMealsThatAreNotInMenu() {
    this.http.get(this.mealsNotInMenuURL)
        .map(meals => meals.json())
        .catch(this.handleError)
        .subscribe(
        meals => this._meals.next(<Meal[]>JSON.parse(JSON.stringify(meals))),
        error => this.handleError(error)
        );
}
```

Next in MenuComponent we need to add field for meals and subscribe to meals observable from MenuService. Also I have added few more fields that we are going to need.

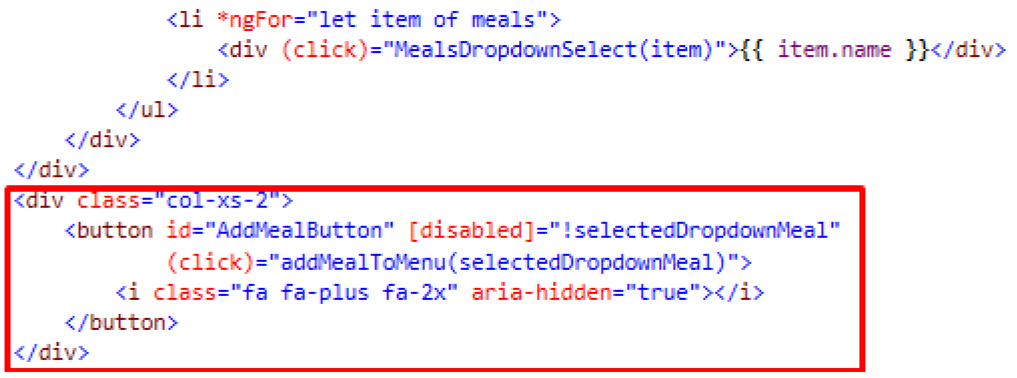
```
export class MenuComponent {
    menu: Menu;
    meals: Meal[];
    errorMessage: string;
    selectedDropdownMeal: Meal;
    noMealSelectedName = ' - Select item - ';

    constructor(private _menuService: MenuService) { }

    ngOnInit() {
        this.getMenu();
        this.getMeals();
    }


  private getMeals(): void {
      this._menuService.meals.subscribe(
          meals => this.meals = meals,
          error => this.errorMessage
      )
  }

  MealsDropdownSelect(meal: Meal) {
      this.selectedDropdownMeal = meal;
  }
```

And also we should add following HTML mark-up to the menu.html.

```
</table>

    <div class="col-xs-6">
        <div class="dropdown">
            <button style="width:100%;" class="btn btn-default dropdow-toggle" type="button"
                    id="MealsAddDropdown" data-toggle="dropdown" aria-haspopup="true" aria-expanded="true">
                {{ selectedDropdownMeal ? selectedDropdownMeal.name : noMealSelectedName }}
                <i class="fa fa-caret-down" aria-hidden="true" id="MenuDropdownMeal"></i>
            </button>
            <ul class="dropdown-menu" style="width:100%">
                <li><div (click)="MealsDropdownSelect(null)">{{noMealSelectedName}}</div></li>
                <li *ngFor="let item of meals">
                    <div (click)="MealsDropdownSelect(item)">{{ item.name }}</div>
                </li>
            </ul>
        </div>
    </div>
```

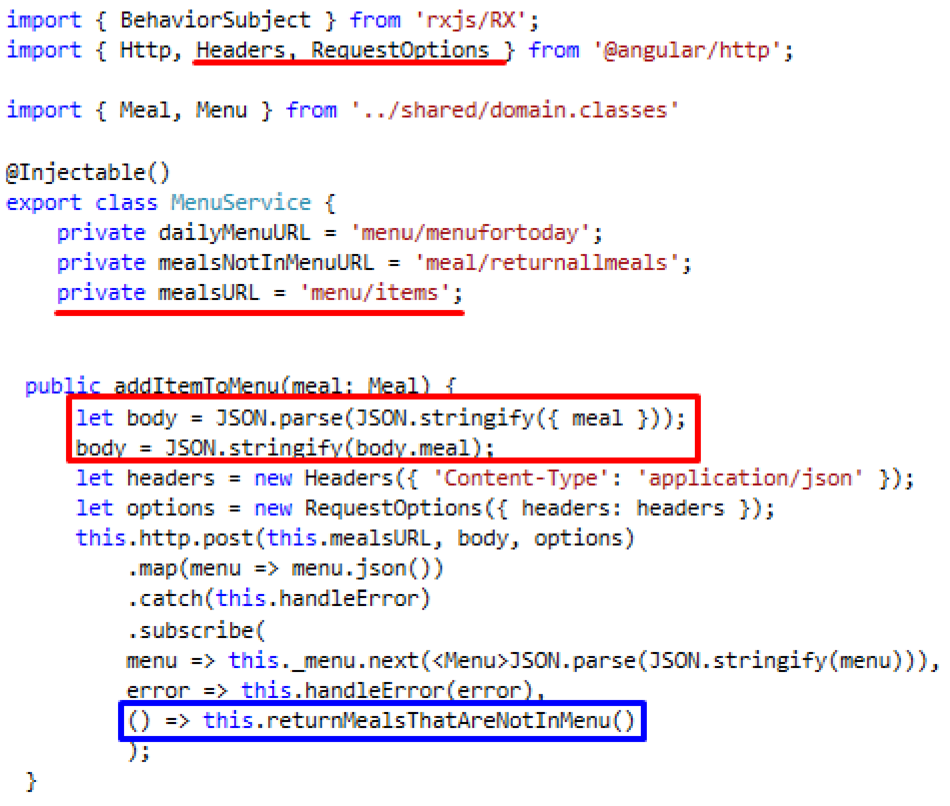
This is our dropdown. We have a button where we are going to set the name of the selected meal, or default text if no meal is selected and we have unordered list, when we click on it dropdown will appear with all the meals that can be added to the menu. As you can see we have *ngFor that will create <li> element for each meal in array. You can also see that we use MealsDropdownSelect() method to set the selectedDoprdownMeal field in component.

Now we need to add a button that will add meal to the menu. Also that button needs to be disabled if no meal is selected from the dropdown.



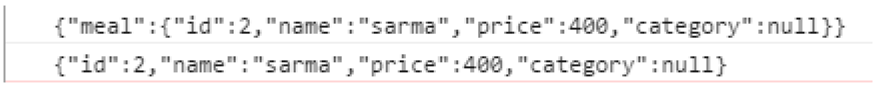
```
            <li *ngFor="let item of meals">
                <div (click)="MealsDropdownSelect(item)">{{ item.name }}</div>
            </li>
        </ul>
    </div>
</div>
<div class="col-xs-2">
    <button id="AddMealButton" [disabled]="!selectedDropdownMeal"
            (click)="addMealToMenu(selectedDropdownMeal)">
        <i class="fa fa-plus fa-2x" aria-hidden="true"></i>
    </button>
</div>
```

You can see that we have used property binding to bind angular logic to disabled property of button element, which will be disabled if there isn’t any meal selected. We now need to implement addMealToMenu function. We need to create that function in MenuService and then call it from MenuComponent.



```
import { BehaviorSubject } from 'rxjs/RX';
import { Http, Headers, RequestOptions } from '@angular/http';

import { Meal, Menu } from '../shared/domain.classes'

@Injectable()
export class MenuService {
    private dailyMenuURL = 'menu/menufortoday';
    private mealsNotInMenuURL = 'meal/returnallmeals';
    private mealsURL = 'menu/items';


  public addItemToMenu(meal: Meal) {
      let body = JSON.parse(JSON.stringify({ meal }));
      body = JSON.stringify(body.meal);
      let headers = new Headers({ 'Content-Type': 'application/json' });
      let options = new RequestOptions({ headers: headers });
      this.http.post(this.mealsURL, body, options)
          .map(menu => menu.json())
          .catch(this.handleError)
          .subscribe(
          menu => this._menu.next(<Menu>JSON.parse(JSON.stringify(menu))),
          error => this.handleError(error),
          () => this.returnMealsThatAreNotInMenu()
          );
  }
```

First two lines in red rectangle need to be in that way in order to format the body needed for ASP .NET MVC. In the image below you can see what first JSON.stringify outputs in the first line, and what second one does in the second line.



```
{"meal":{"id":2,"name":"sarma","price":400,"category":null}}
{"id":2,"name":"sarma","price":400,"category":null}
```

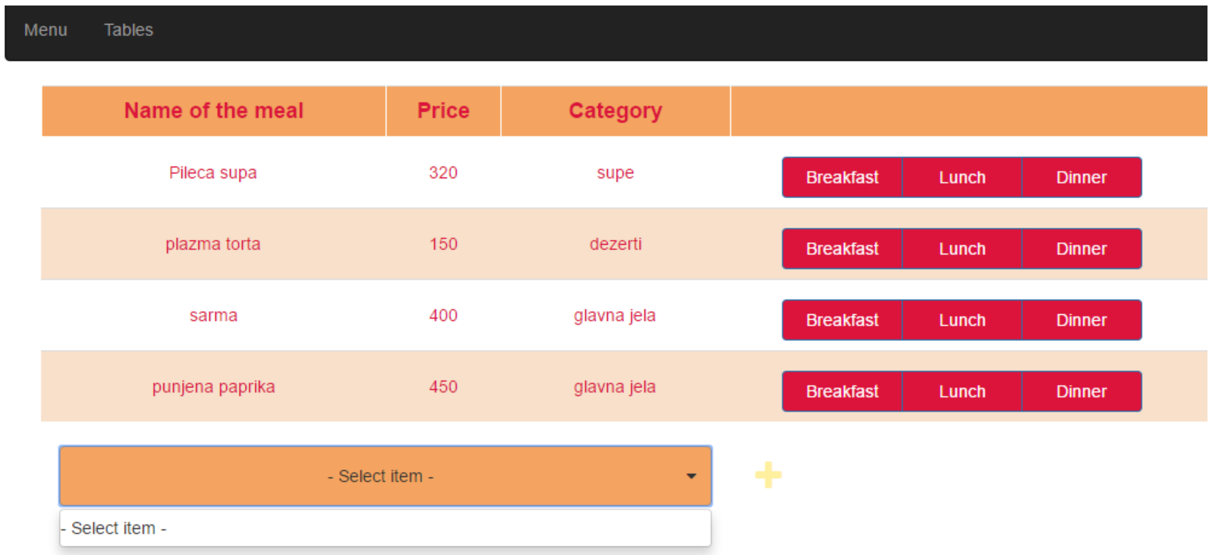
After that we are creating headers and making post request. As we know requests return observables, so we need to subscribe to it since our requests returns updated menu. We now also need to update dropdown and to remove meal we've just added from it. We can call returnMealsTharAreNotInTheMenu function, but we need to call it after observable from addItemToMenu completed. For that we have onCompleted method in blue rectangle.

In MenuComponent we just need to add method for adding the meal to menu that will get Meal object as parameter. Method will call function from service and set currently selected meal to null.

```
addMealToMenu(meal: Meal) {
    this._menuService.addItemToMenu(meal);
    this.selectedDropdownMeal = null;
}
```

As you can see we are not subscribing again to menu and meals observables to get the new data, we get new data automatically as long as we are subscribed to that observable.

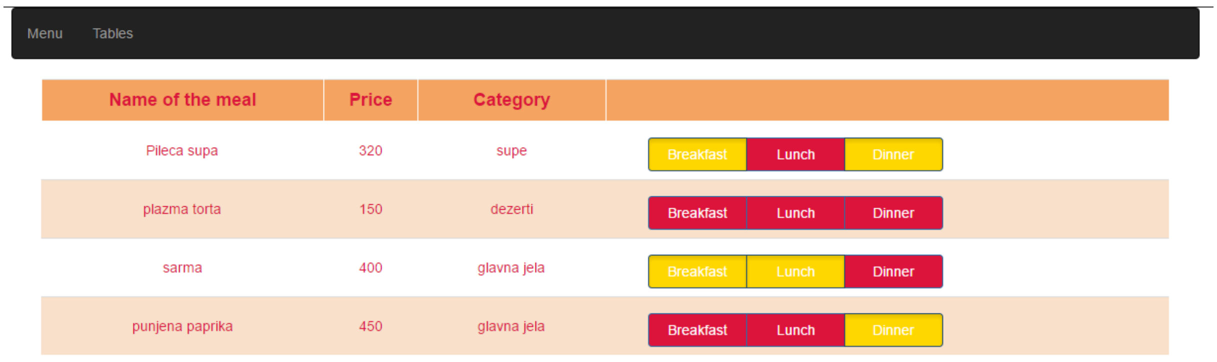
Run ts task and start the project, you will now see that selecting a meal and clicking on a add button will add meal to the menu and remove it from drop down. But there is another problem, now if we add all meals from dropdown to menu, we will still have dropdown with empty item.



Name of the meal	Price	Category	
Pileca supa	320	supe	Breakfast Lunch Dinner
plazma torta	150	dezerti	Breakfast Lunch Dinner
sarma	400	glavna jela	Breakfast Lunch Dinner
punjena paprika	450	glavna jela	Breakfast Lunch Dinner

We should remove the buttons if there are no more meals to add to the menu. We can do that using another built in structural directive *ngIf, it adds element to DOM if the condition is true or removes it if it is false. As you can see below we have just added another div in which we have relocated two buttons and now buttons should be removed when there are no meals to add to the menu.

```
<div *ngIf="meals && meals.length>0">
    <div class="col-xs-6">
        <div class="dropdown">
            <button style="width:100%;" class="btn btn-default dropdow-toggle" type="button"
                    id="MealsAddDropdown" data-toggle="dropdown" aria-haspopup="true" aria-expanded="true">
                {{selectedDropdownMealName}}
                <i class="fa fa-caret-down" aria-hidden="true" id="MenuDropdownMeal"></i>
            </button>
            <ul class="dropdown-menu" style="width:100%">
                <li><div (click)="MealsDropdownSelect(null)"> - Select item - </div></li>
                <li *ngFor="let item of meals">
                    <div> (click)="MealsDropdownSelect(item)">{{ item.Name }}</div>
                </li>
            </ul>
        </div>
    </div>
    <div class="col-xs-2">
        <button id="AddMealButton" [disabled]="selectedDropdownMeal"
                (click)="addMealToMenu(selectedDropdownMeal)">
            <i class="fa fa-plus fa-2x" aria-hidden="true"></i>
        </button>
    </div>
</div>
```



Name of the meal	Price	Category	
Pileca supa	320	supe	Breakfast Lunch Dinner
plazma torta	150	dezerti	Breakfast Lunch Dinner
sarma	400	glavna jela	Breakfast Lunch Dinner
punjena paprika	450	glavna jela	Breakfast Lunch Dinner

Only thing left now is to add remove button in each row in the table. Add new button in each row, and below is markup.

```
<td class="col-xs-6">
    <div class="col-xs-8">
        <div class="btn-group button-group-meal">
            <button onfocus="this.blur();"
                    [ngClass]="{'btn':true, 'btn-primary':true,'active': item.breakfast}"
                    (click)="changeStateOfMeal(item, 'breakfast')">
                <span> Breakfast </span>
            </button>
            <button onfocus="this.blur();"
                    [ngClass]="{'btn':true, 'btn-primary':true,'active': item.lunch}"
                    (click)="changeStateOfMeal(item, 'lunch')">
                <span> Lunch </span>
            </button>
            <button onfocus="this.blur();"
                    [ngClass]="{'btn':true, 'btn-primary':true, 'active': item.dinner}"
                    (click)="changeStateOfMeal(item, 'dinner')">
                <span> Dinner </span>
            </button>

        </div>
    </div>
    <div class="col-xs-2">
        <button id="RemoveMealButton" onfocus="this.blur();" (click)="removeFromMenu(item)">
            <i class="fa fa-minus fa-2x" aria-hidden="true"></i>
        </button>
    </div>
```

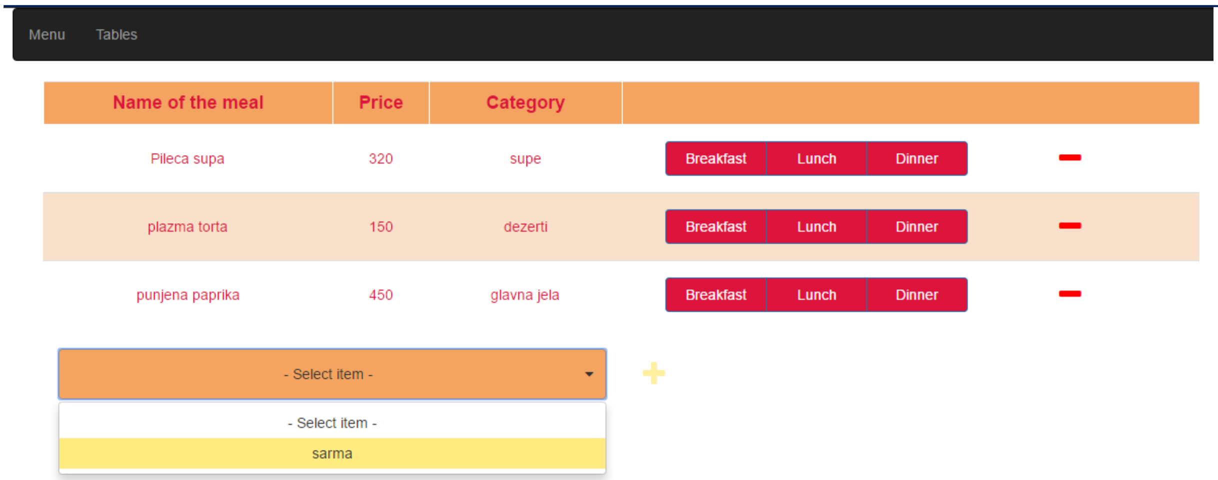
Now we should implement removeFromMenu method. First add method in MenuService.

```
public removeItemFromMenu(id: number) {
    this.http.delete(this.mealsURL + "/" + id)
        .map(menu => menu.json())
        .catch(this.handleError)
        .subscribe(
        menu => this._menu.next(<Menu>JSON.parse(JSON.stringify(menu))),
        error => this.handleError(error),
        () => this.returnMealsThatAreNotInMenu()
        );
}
```

We are using delete method of http service to do that. And also we need to implement method in component that will call method we just implemented in service.

```
removeFromMenu(item: MenuItem) {
    this._menuService.removeItemFromMenu(item.id);
}
```

If we run ts task and start the project we should now see button for removing items from menu.



Name of the meal	Price	Category		
Pileca supa	320	supe	Breakfast Lunch Dinner	−
plazma torta	150	dezerti	Breakfast Lunch Dinner	−
punjena paprika	450	glavna jela	Breakfast Lunch Dinner	−

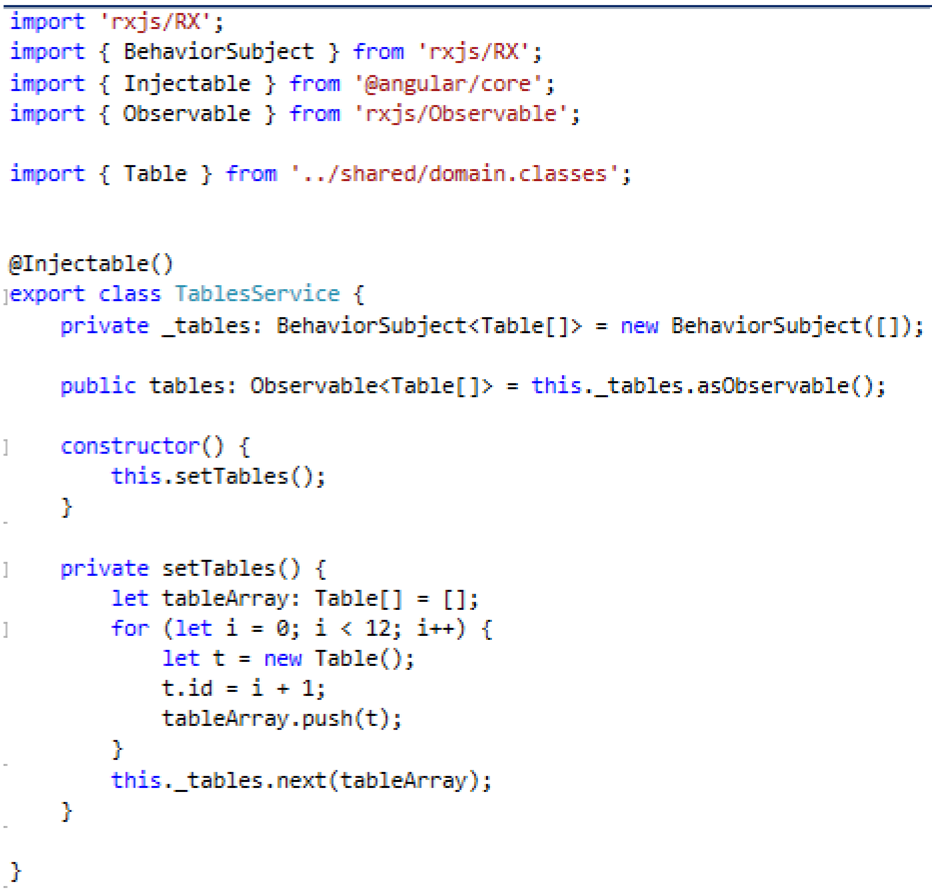
8 Creating table list view

We need to create an overview of tables in restaurant. Currently there are 12 tables in restaurant and we should be able to navigate to the page where we can create new order, if there is no order for that table, and to the page where we can check current order for that table.

We need to create new view at wwwroot/AngularApp/Views/table-overview.html with the following mark-up.

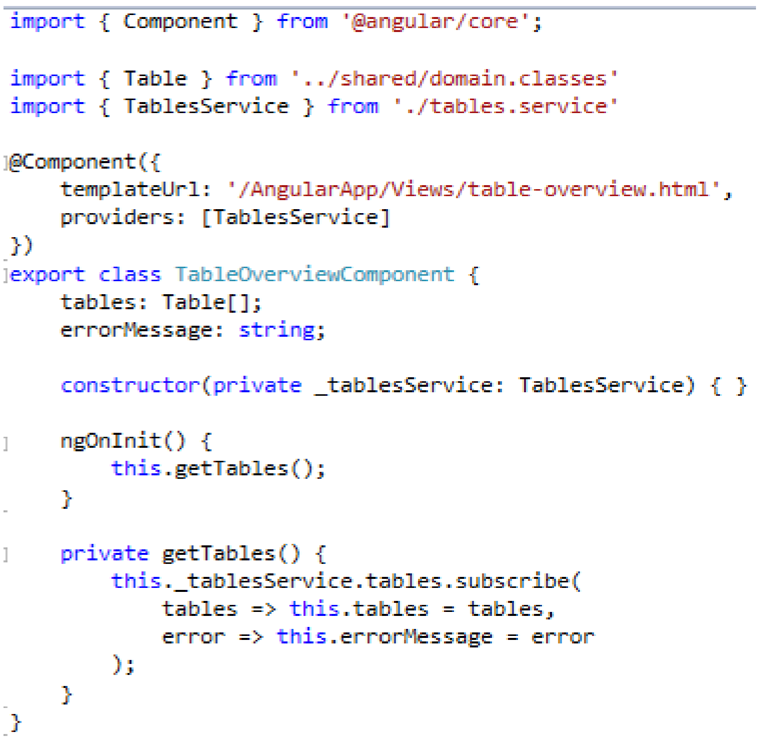
```
<div class="row">
    <div class="col-xs-4" *ngFor="let table of tables">
        <div class="restourant-table">
            <div class="col-xs-4 no-padding left-part-tables" style="font-size:90px; height:100%;">
                <span>{{ table.id }}</span>
            </div>
            <div class="col-xs-8 no-padding right-part-tables">
                <div class="col-xs-12 add-order-div no-padding">
                    <div *ngIf="table.order" class="has-order">
                    </div>
                    <div *ngIf="!table.order" class="no-order">
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

For now we are just showing the id from the table. Now we need to create tables field in our component and to initialize it from our service. Service should be created at Scripts/tables/tables.service.ts and we should create BehaviourSubject and Observable and since we are not getting table objects from database we should create new array and set ids of the tables.



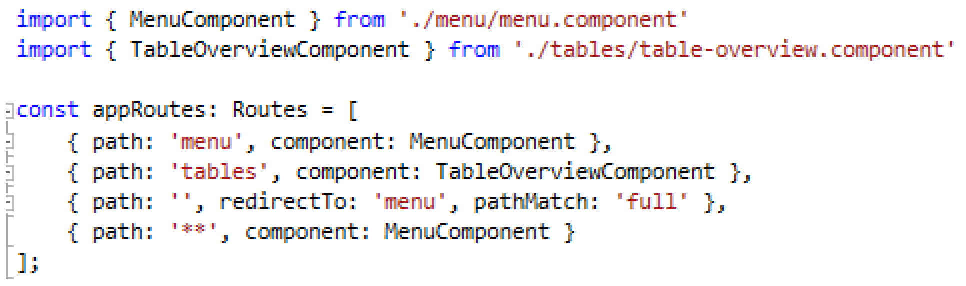
```
import 'rxjs/RX';
import { BehaviorSubject } from 'rxjs/RX';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';

import { Table } from '../shared/domain.classes';


@Injectable()
export class TablesService {
    private _tables: BehaviorSubject<Table[]> = new BehaviorSubject([]);

    public tables: Observable<Table[]> = this._tables.asObservable();

    constructor() {
        this.setTables();
    }

    private setTables() {
        let tableArray: Table[] = [];
        for (let i = 0; i < 12; i++) {
            let t = new Table();
            t.id = i + 1;
            tableArray.push(t);
        }
        this._tables.next(tableArray);
    }

}
```

In component Scripts/app/tables/table-overview.component.ts we should subscribe to the tables observable from TablesService.



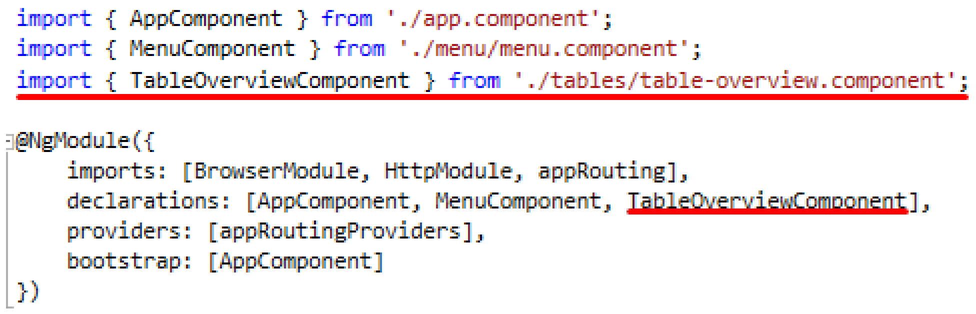
```
import { Component } from '@angular/core';

import { Table } from '../shared/domain.classes'
import { TablesService } from './tables.service'

@Component({
    templateUrl: '/AngularApp/Views/table-overview.html',
    providers: [TablesService]
})
export class TableOverviewComponent {
    tables: Table[];
    errorMessage: string;

    constructor(private _tablesService: TablesService) { }

    ngOnInit() {
        this.getTables();
    }

    private getTables() {
        this._tablesService.tables.subscribe(
            tables => this.tables = tables,
            error => this.errorMessage = error
        );
    }
}
```

In app.routing.ts we need to create new route.



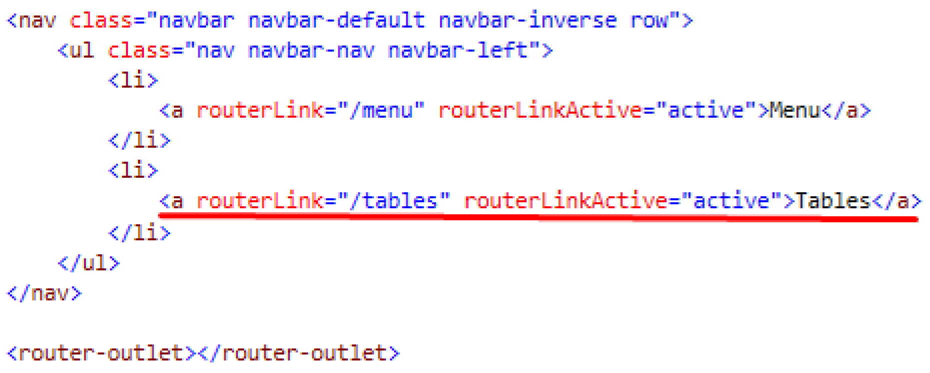
```
import { MenuComponent } from './menu/menu.component'
import { TableOverviewComponent } from './tables/table-overview.component'

const appRoutes: Routes = [
    { path: 'menu', component: MenuComponent },
    { path: 'tables', component: TableOverviewComponent },
    { path: '', redirectTo: 'menu', pathMatch: 'full' },
    { path: '**', component: MenuComponent }
];
```

We also need to include TableOverviewComponent in declarations of our AppModule



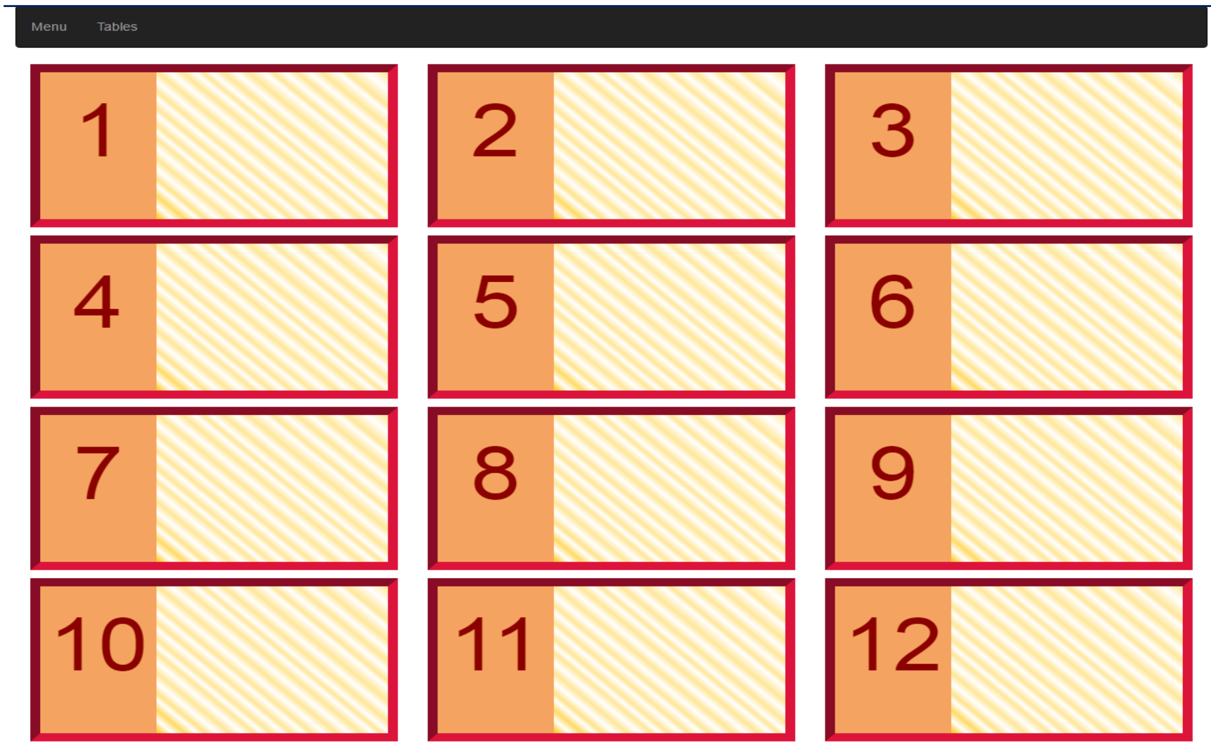
```
import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';
import { TableOverviewComponent } from './tables/table-overview.component';

@NgModule({
    imports: [BrowserModule, HttpModule, appRouting],
    declarations: [AppComponent, MenuComponent, TableOverviewComponent],
    providers: [appRoutingProviders],
    bootstrap: [AppComponent]
})
```

And also app.html.



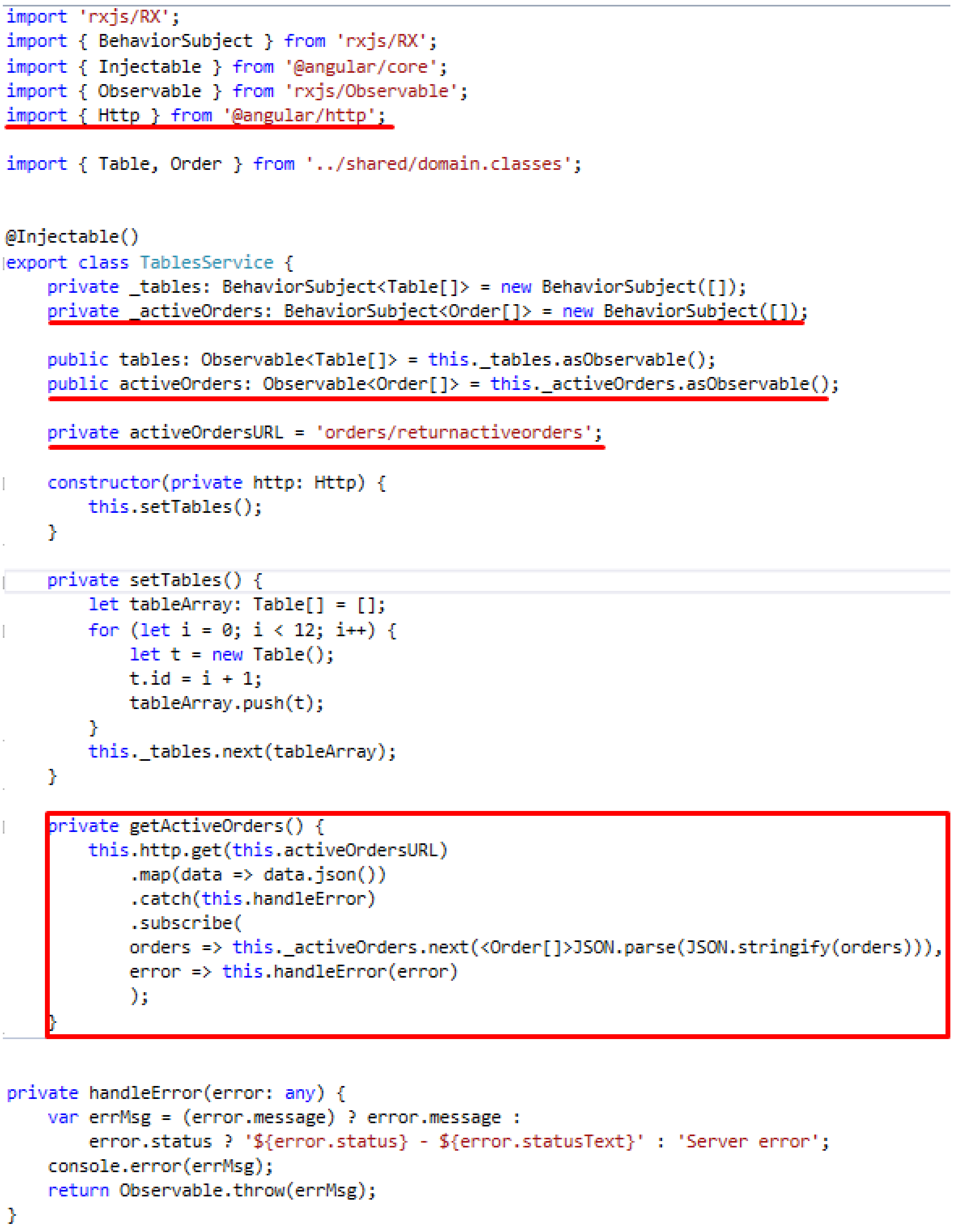
```
<nav class="navbar navbar-default navbar-inverse row">
    <ul class="nav navbar-nav navbar-left">
        <li>
            <a routerLink="/menu" routerLinkActive="active">Menu</a>
        </li>
        <li>
            <a routerLink="/tables" routerLinkActive="active">Tables</a>
        </li>
    </ul>
</nav>

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

Now run ts task and start the application. Following view should show when you visit Tables link.


Menu
Tables
1
2
3
4
5
6
7
8
9
10
11
12

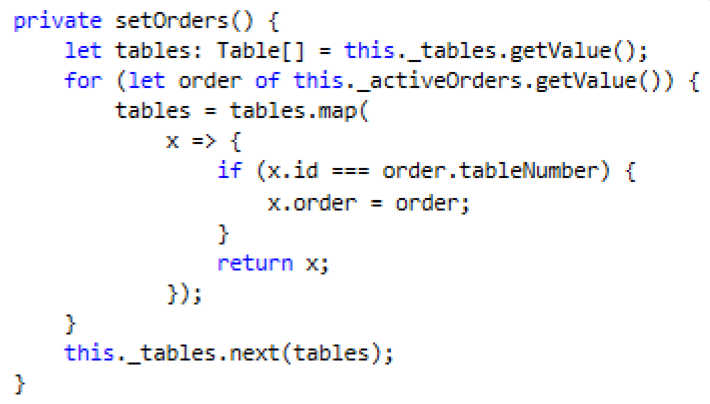
8.1 Setting orders to tables (ngSwitch)

First we need to fetch active orders from our web api. In tables.service.ts we need to make call.



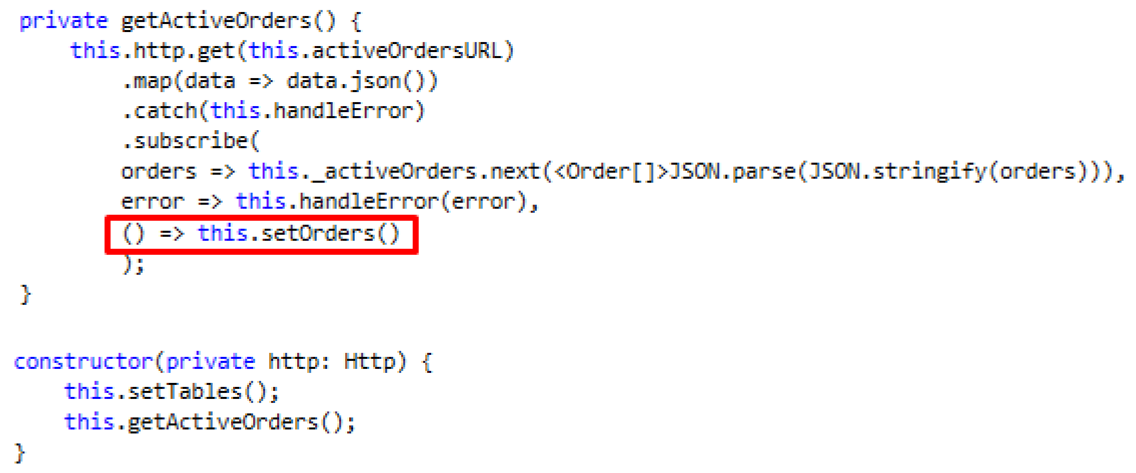
```
import 'rxjs/RX';
import { BehaviorSubject } from 'rxjs/RX';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { Http } from '@angular/http';

import { Table, Order } from '../shared/domain.classes';


@Injectable()
export class TablesService {
    private _tables: BehaviorSubject<Table[]> = new BehaviorSubject([]);
    private _activeOrders: BehaviorSubject<Order[]> = new BehaviorSubject([]);

    public tables: Observable<Table[]> = this._tables.asObservable();
    public activeOrders: Observable<Order[]> = this._activeOrders.asObservable();

    private activeOrdersURL = 'orders/returnactiveorders';

    constructor(private http: Http) {
        this.setTables();
    }

    private setTables() {
        let tableArray: Table[] = [];
        for (let i = 0; i < 12; i++) {
            let t = new Table();
            t.id = i + 1;
            tableArray.push(t);
        }
        this._tables.next(tableArray);
    }

    private getActiveOrders() {
        this.http.get(this.activeOrdersURL)
            .map(data => data.json())
            .catch(this.handleError)
            .subscribe(
            orders => this._activeOrders.next(<Order[]>JSON.parse(JSON.stringify(orders))),
            error => this.handleError(error)
            );
    }

private handleError(error: any) {
    var errMsg = (error.message) ? error.message :
        error.status ? '${error.status} - ${error.statusText}' : 'Server error';
    console.error(errMsg);
    return Observable.throw(errMsg);
}
```

This will fetch orders, but we still need to connect orders with tables. We will do that with the function below. We are using arrow function which are introduced in es2015 to set order to corresponding tables.



```
private setOrders() {
    let tables: Table[] = this._tables.getValue();
    for (let order of this._activeOrders.getValue()) {
        tables = tables.map(
            x => {
                if (x.id === order.tableNumber) {
                    x.order = order;
                }
                return x;
            });
    }
    this._tables.next(tables);
}
```

Finally we are going to call setOrders method in onComplete parameter of getActiveOrders observable and also call getActiveOrders in constructor so we populate



```
private getActiveOrders() {
    this.http.get(this.activeOrdersURL)
        .map(data => data.json())
        .catch(this.handleError)
        .subscribe(
        orders => this._activeOrders.next(<Order[]>JSON.parse(JSON.stringify(orders))),
        error => this.handleError(error),
        () => this.setOrders()
        );
}

constructor(private http: Http) {
    this.setTables();
    this.getActiveOrders();
}
```

Since we are already subscribed to tables observable view is automatically updated.

Now we need to show the status of the order and link to the order if order for the table exists. We are going to use ngSwitch which is yet another built in directive. As you can see below, we specify what field we want to use for switch and then write down cases. You can also specify default case which will fire up if no other cases are satisfied.

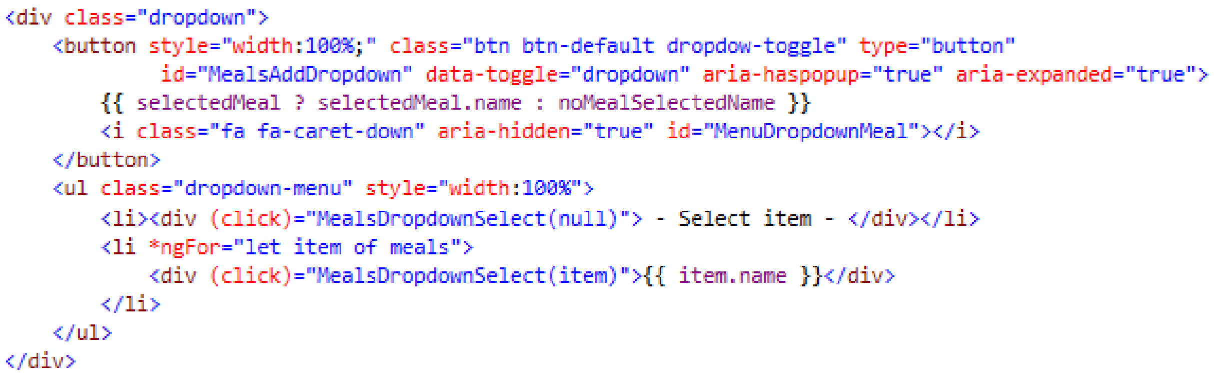
```
<div class="col-xs-12 add-order-div no-padding">
    <div *ngIf="table.order" class="has-order">
        <div id="statusOfTheOrder">
            <div [ngSwitch]="table.order.status">
                <div *ngSwitchCase="1">
                    Ordered
                </div>
                <div *ngSwitchCase="2">
                    <span>Served</span>
                </div>
                <div *ngSwitchDefault>
                    <span>Unknown status</span>
                </div>
            </div>

        </div>
        <div id="orderDetails">
            <a href="#"><i class="fa fa-chevron-right fa-5x" aria-hidden="true"></i></a>
        </div>
    </div>
    <div *ngIf="!table.order" class="no-order">
        <a href="#">
            <div>
                <i class="fa fa-plus-circle fa-5x col-xs-12" aria-hidden="true"></i>
                <span>Add order?</span>
            </div>
        </a>
    </div>
</div>
```

9 Creating meal dropdown component and view

9.1 Meal dropdown view

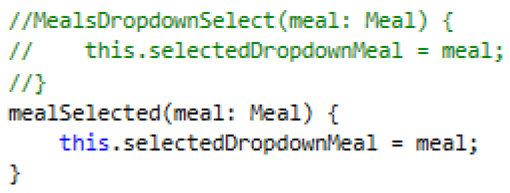
```
<div *ngIf="meals && meals.length>0">
    <div class="col-xs-6">
        <div class="dropdown">
            <div class="dropdown">
                <button style="width:100%;" class="btn btn-default dropdow-toggle" type="button"
                        id="MealsAddDropdown" data-toggle="dropdown" aria-haspopup="true" aria-expanded="true">
                    {{ selectedMeal ? selectedMeal.name : noMealSelectedName }}
                    <i class="fa fa-caret-down" aria-hidden="true" id="MenuDropdownMeal"></i>
                </button>
                <ul class="dropdown-menu" style="width:100%">
                    <li><div (click)="MealsDropdownSelect(null)"> - Select item - </div></li>
                    <li *ngFor="let item of meals">
                        <div (click)="MealsDropdownSelect(item)">{{ item.name }}</div>
                    </li>
                </ul>
            </div>
        </div>
    </div>
    <div class="col-xs-2">
        <button id="AddMealButton" [disabled]="!selectedDropdownMeal"
                (click)="addMealToMenu(selectedDropdownMeal)">
            <i class="fa fa-plus fa-2x" aria-hidden="true"></i>
        </button>
    </div>
</div>
```

Now we are going to move the dropdown button from menu.html to a new view, we are just going to copy HTML mark-up to wwwroot/AngularApp/Views/PartialViews/meal-dropdown.html. The mark-up should be as on the image below.



```
<div class="dropdown">
    <button style="width:100%;" class="btn btn-default dropdow-toggle" type="button"
             id="MealsAddDropdown" data-toggle="dropdown" aria-haspopup="true" aria-expanded="true">
        {{ selectedMeal ? selectedMeal.name : noMealSelectedName }}
        <i class="fa fa-caret-down" aria-hidden="true" id="MenuDropdownMeal"></i>
    </button>
    <ul class="dropdown-menu" style="width:100%">
        <li><div (click)="MealsDropdownSelect(null)"> - Select item - </div></li>
        <li *ngFor="let item of meals">
            <div (click)="MealsDropdownSelect(item)">{{ item.name }}</div>
        </li>
    </ul>
</div>
```

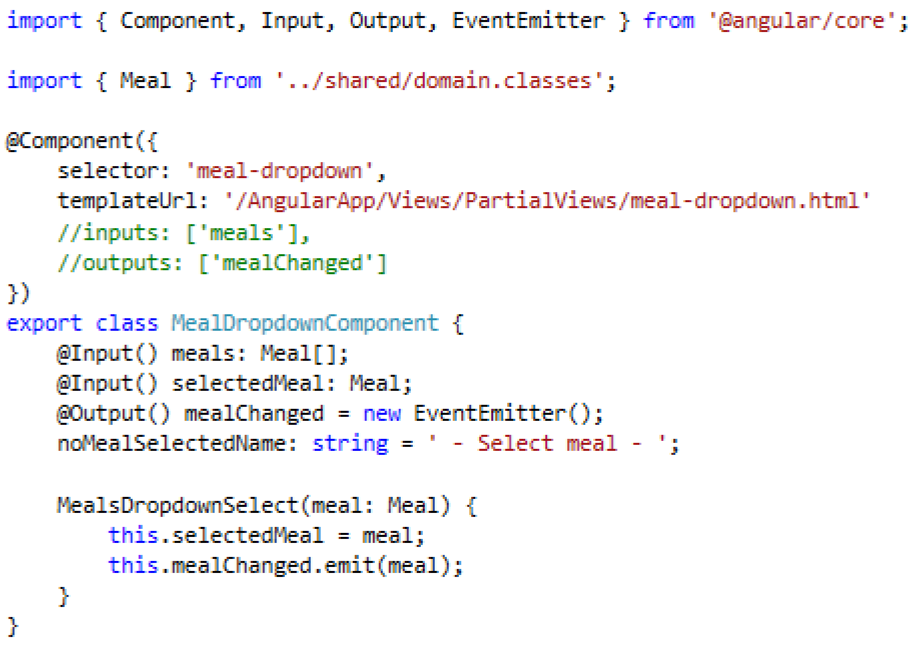
And we need to change menu.component.ts.



```
//MealsDropdownSelect(meal: Meal) {
//    this.selectedDropdownMeal = meal;
//}
mealSelected(meal: Meal) {
    this.selectedDropdownMeal = meal;
}
```

9.2 Meal dropdown component (Input and output)

Create new component at Scripts/app/meals/meal-dropdown.component.ts



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

import { Meal } from '../shared/domain.classes';

@Component({
    selector: 'meal-dropdown',
    templateUrl: '/AngularApp/Views/PartialViews/meal-dropdown.html'
    //inputs: ['meals'],
    //outputs: ['mealChanged']
})
export class MealDropdownComponent {
    @Input() meals: Meal[];
    @Input() selectedMeal: Meal;
    @Output() mealChanged = new EventEmitter();
    noMealSelectedName: string = ' - Select meal - ';

    MealsDropdownSelect(meal: Meal) {
        this.selectedMeal = meal;
        this.mealChanged.emit(meal);
    }
}
```

Defining input and output properties can be done in two ways, first one is in @Component decorator, those are the two commented lines in the image above and by putting @Input and @Output next to the properties that will be imported into the component or exported out of it.

With @Input we are telling component that the value will be imported from parent component, and @Output will notify the parent component whenever value of the property changes. As you can see, whenever dropdown value changes we are emitting new value to the parent component.

Now menu.html should look like this.

```
<div *ngIf="meals && meals.length>0">
    <div class="col-xs-6">
        <div class="dropdown">
            <meal-dropdown [meals]="meals" [selectedMeal]="selectedDropdownMeal"
                          (mealChanged)="mealSelected($event)"></meal-dropdown>
        </div>
    </div>
    <div class="col-xs-2">
        <button id="AddMealButton" [disabled]="!selectedDropdownMeal"
                (click)="addMealToMenu(selectedDropdownMeal)">
            <i class="fa fa-plus fa-2x" aria-hidden="true"></i>
        </button>
    </div>
</div>
```

As you can see, we are using property binding to import meals into the meal-dropdown component and we are using event binding to catch when child component emits new data.

Finally in order for this to work we need to add our component to the module

```
import { appRouting, appRoutingProviders } from './app.routing';
import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';
import { TableOverviewComponent } from './tables/table-overview.component';
import { MealDropdownComponent } from './meals/meal-dropdown.component';

@NgModule({
    imports: [BrowserModule, HttpModule, appRouting, FormsModule],
    declarations: [AppComponent, MenuComponent, TableOverviewComponent, MealDropdownComponent],
    providers: [appRoutingProviders],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

10 Useful link

Template-driven forms

Reactive forms

Style guide

Routing tutorial

11 Tasks

1. Menu view
    a. Save changes to server when clicking on Breakfast/Lunch/Dinner button
2. Orders view
    a. Create orders view which will be used for creating new and editing existing orders
    b. There should be two components one for editing orders and one for creating new orders, but they should both use same view.
    c. You should access edit order component with order id and new order component with table id. You should use route parameters, about which you can find more at Routing tutorial link above.
    d. For forms you should use either Template-driven or Reactive forms
    e. There should be link for discarding order and going back to tables view
    f. Apply appropriate css and bootstrap styles
3. Table-overview view
    a. Link edit order and new order to appropriate buttons on the page
4. Meals view
    a. Create meals view in which we will list all meals, add new meals and add or edit existing meals
    b. Create appropriate component for meals view
    c. Adding and editing meals should happen on the same page, you should use nested router-outlet and define routes inside of component you've used for meals view