tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Angular 8 is an open source, TypeScript based frontend web application framework. Angular 8 has been released by **Google's Angular** community. This tutorial starts with the architecture of *Angular 8*,setup simple project, data binding, then walks through forms, templates, routing and explains about Angular 8 new features. Finally, conclude with step by step working example.

Audience

This tutorial is prepared for professionals who are aspiring to make a career in the field of Web application developer. This tutorial is intended to make you comfortable in getting started with the Angular8 concepts with examples.

Prerequisites

Before proceeding with the various types of concepts given in this tutorial, we assume that the readers have the basic knowledge on HTML, CSS and OOPS concepts. In addition to this, it will be very helpful, if the readers have a sound knowledge on TypeScript and JavaScript.

Copyright & Disclaimer

Table of Contents

1. Angular 8 — Introduction

Angular 8 is a TypeScript based full-stack web framework for building web and mobile applications. One of the major advantage is that the Angular 8 support for web application that can fit in any screen resolution. Angular application is fully compatible for mobiles, tablets, laptops or desktops. Angular 8 has an excellent user interface library for web developers which contains reusable UI components.

This functionality helps us to create Single Page Applications (SPA). SPA is reactive and fast application. For example, if you have a button in single page and click on the button then the action performs dynamically in the current page without loading the new page from the server. Angular 8 is Typescript based object oriented programming and support features for server side programming as well.

Comparison of angular versions

As we know already, Google releases the version of **Angular** for the improvement of mobile and web development capabilities. All the released versions are backward compatible and can be updated easily to the newer version. Let's go through the comparison of released versions.

AngularJS

AngularJs is very powerful JavaScript framework. It was released in October 2010. AngularJS based on Model View Controller (MVC) architecture and automatically handles JavaScript code suitable for each browser.

Angular 2.0

Angular 2.0 was released in September 2016. It is re-engineered and rewritten version of AngularJS. AngularJs had a focus on controllers but, version 2 has changed focus on components. Components are the main building block of application. It supports features for speed in rendering, updating pages and building cross-platform native mobile apps for Google Android and iOS.

Angular 4.0

Angular 4.0 was released in March 2017. It is updated to TypeScript 2.2, supports ng if-else conditions whereas Angular 2 supported only if conditions. Angular 4.0 introduces animation packages, Http search parameters and finally angular 4 applications are smaller and faster.

Angular 5.0

Angular 5.0 was released in November 2017. It supported some of the salient features such as HTTPClient API, Lambda support, Improved Compiler and build optimizer.

Angular 6.0

Angular 6.0 was released in May 2018. Features added to this version are updated Angular CLI, updated CDK, updated Angular Material, multiple validators and usage of reactive JS library.

Angular 7.0

Angular 7.0 was released in October 2018. Some of salient features are Google supported community, POJO based development, modular structure, declarative user interface and modular structure.

Angular 8 New features

Angular 8 comes up with the following new attractive features:

- **Bazel support** - If your application uses several modules and libraries, Bazel concurrent builds helps to load faster in your application.
- **Lazy loading** - Angular 8 splits **AppRoutingModule** into smaller bundles and loads the data in the DOM.
- **Differential loading** - When you create an application, Angular CLI generates modules and this will be loaded automatically then browser will render the data.
- **Web worker** - It is running in the background, without affecting the performance of a page.
- **Improvement of CLI workflow** - Angular 8 CLI commands **ng-build, ng-test** and **ng-run** are extended to third party libraries.
- **Router Backward Compatibility** - Angular router backward compatibility feature helps to create path for larger projects so user can easily add their coding with the help of lazy coding.
- **Opt-in usage sharing** - User can opt into share Angular CLI usage data.

Applications

Some of the popular website using Angular Framework are listed below:

- **Weather.com** - It is one of the leading forecasting weather report website.
- **Youtube** - It is a video and sharing website hosted by **Google.**
- **Netflix** - It is a technology and media services provider.
- **PayPal** - It is an online payment system.

2. Angular 8 — Installation

This chapter explains about how to install **Angular 8** on your machine. Before moving to the installation, let's verify the prerequisite first.

Prerequisite

As we know already, Angular is written in **TypeScript**. We need **Node** and **npm** to compile the files into **JavaScript** after that, we can deploy our application. For this purpose, **Node.js** must be installed in your system. Hopefully, you have installed **Node.js** on your machine.

We can check it using the below command:

```
node --version
```

You could see the version of node. It is show below:

```
v14.2.0
```

If **Node** is not installed, you can download and install by visiting the following link:

https://nodejs.org/en/download/.

Angular 8 installation

Angular 8 CLI installation is based on very simple steps. It will take not more than five minutes to install.

npm is used to install **Angular 8** CLI. Once **Node.js** is installed, **npm** is also installed. If you want verify it, type the below command:

```
npm -v
```

You could see the version below:

```
6.14.4
```

Let's install **Angular 8** CLI using **npm** as follows:

```
npm install -g @angular/cli@^8.0.0
```

To verify **Angular 8** is properly installed on your machine, type the below command:

```
ng version
```

You could see the following response:

```
Angular CLI: 8.3.26
```

```
Node: 14.2.0

OS: win32 x64
Angular:
...

Package                      Version
------------------------------------------------------
@angular-devkit/architect    0.803.26
@angular-devkit/core         8.3.26
@angular-devkit/schematics   8.3.26
@schematics/angular          8.3.26
@schematics/update           0.803.26
rxjs                         6.4.0
```

3. Angular 8 — Creating First Application

Let us create a simple angular application and analyse the structure of the basic angular application.

Let us check whether the Angular Framework is installed in our system and the version of the installed Angular version using below command:

```
ng --version
```

Here,

ng is the CLI application used to create, manage and run Angular Application. It written in JavaScript and runs in NodeJS environment.

The result will show the details of the Angular version as specified below:

```
Angular CLI: 8.3.26
Node: 14.2.0
OS: win32 x64
Angular:
...

Package                      Version
------------------------------------------------------
@angular-devkit/architect    0.803.26
@angular-devkit/core         8.3.26
@angular-devkit/schematics   8.3.26
@schematics/angular          8.3.26
@schematics/update           0.803.26
rxjs                         6.4.0
```

So, Angular is installed in our system and the version is **8.3.26.**

Let us create an Angular application to check our day to day expenses. Let us give **ExpenseManager** as our choice for our new application. Use below command to create the new application.

```
cd /path/to/workspace
ng new expense-manager
```

Here,

new is one of the command of the **ng** CLI application. It will be used to create new application. It will ask some basic question in order to create new application. It is enough to let the application choose the default choices. Regarding routing question as mentioned below, specify **No**. We will see how to create routing later in the **Routing** chapter.

```
Would you like to add Angular routing? No
```

Once the basic questions are answered, the **ng** CLI application create a new Angular application under **expense-manager** folder.

Let us move into the our newly created application folder.

```
cd expense-manager
```

Let us check the partial structure of the application. The structure of the application is as follows:

```
|   favicon.ico
|   index.html
|   main.ts
|   polyfills.ts
|   styles.css
|
+---app
|       app.component.css
|       app.component.html
|       app.component.spec.ts
|       app.component.ts
|       app.module.ts
|
+---assets
|       .gitkeep
|
+---environments
        environment.prod.ts
        environment.ts
```

Here,

- We have shown, only the most important file and folder of the application.
- **favicon.ico** and **assets** are application's icon and application's root asset folder.
- **polyfills.ts** contains standard code useful for browser compatibility.
- **environments** folder will have the application's setting. It includes production and development setup.
- **main.ts** file contains the startup code.
- **index.html** is the application base HTML code.
- **styles.css** is the base CSS code.
- **app folder** contains the Angular application code, which will be learn elaborately in the upcoming chapters.

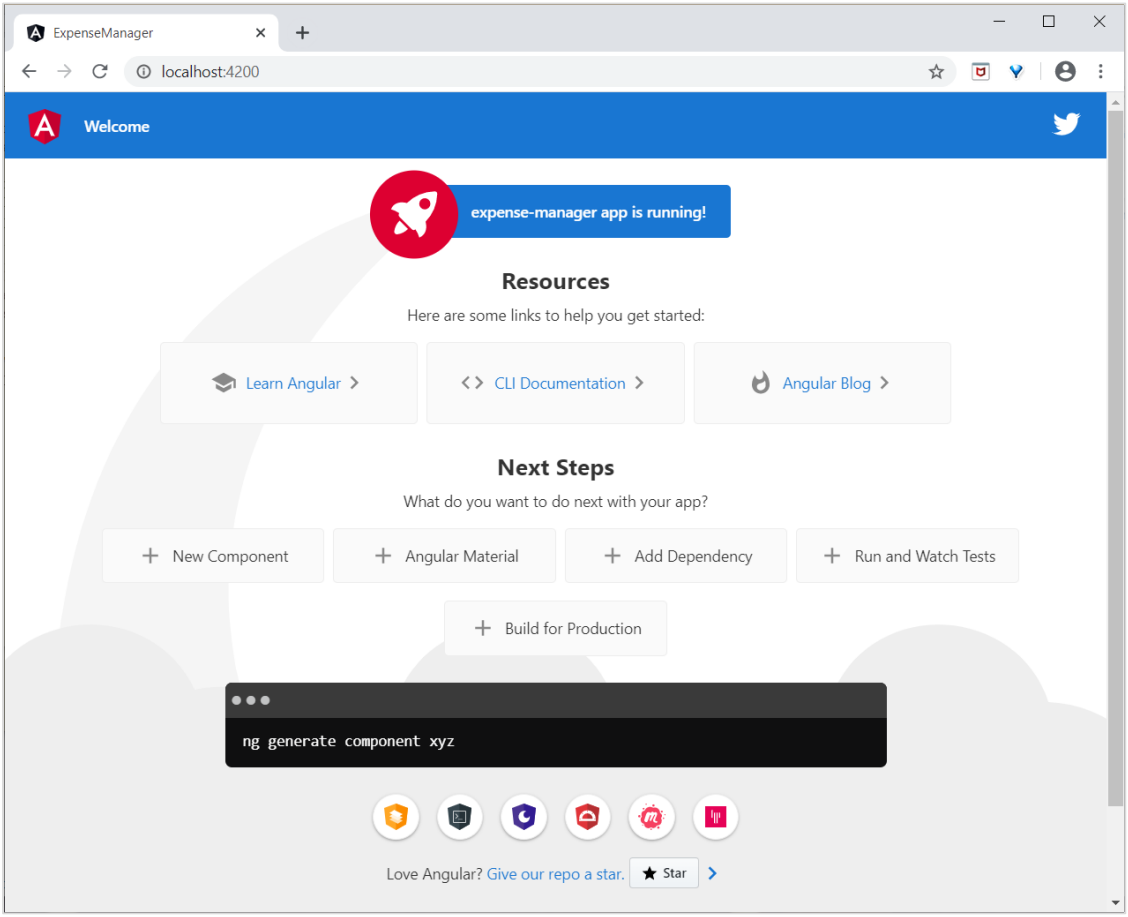
Let us start the application using below command:

```
ng serve

10% building 3/3 modules 0 activei wds: Project is running at
http://localhost:4200/webpack-dev-server/


i wds: webpack output is served from /
```

```
i wds: 404s will fallback to //index.html

chunk {main} main.js, main.js.map (main) 49.2 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 269 kB [initial]
[rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 9.75 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.81 MB [initial] [rendered]
Date: 2020-05-26T05:02:14.134Z - Hash: 0dec2ff62a4247d58fe2 - Time: 12330ms
** Angular Live Development Server is listening on localhost:4200, open your
browser on http://localhost:4200/ **
i wdm: Compiled successfully.
```

Here, **serve** is the sub command used to compile and run the Angular application using a local development web server. **ng server** will start a development web server and serves the application under port, 4200.

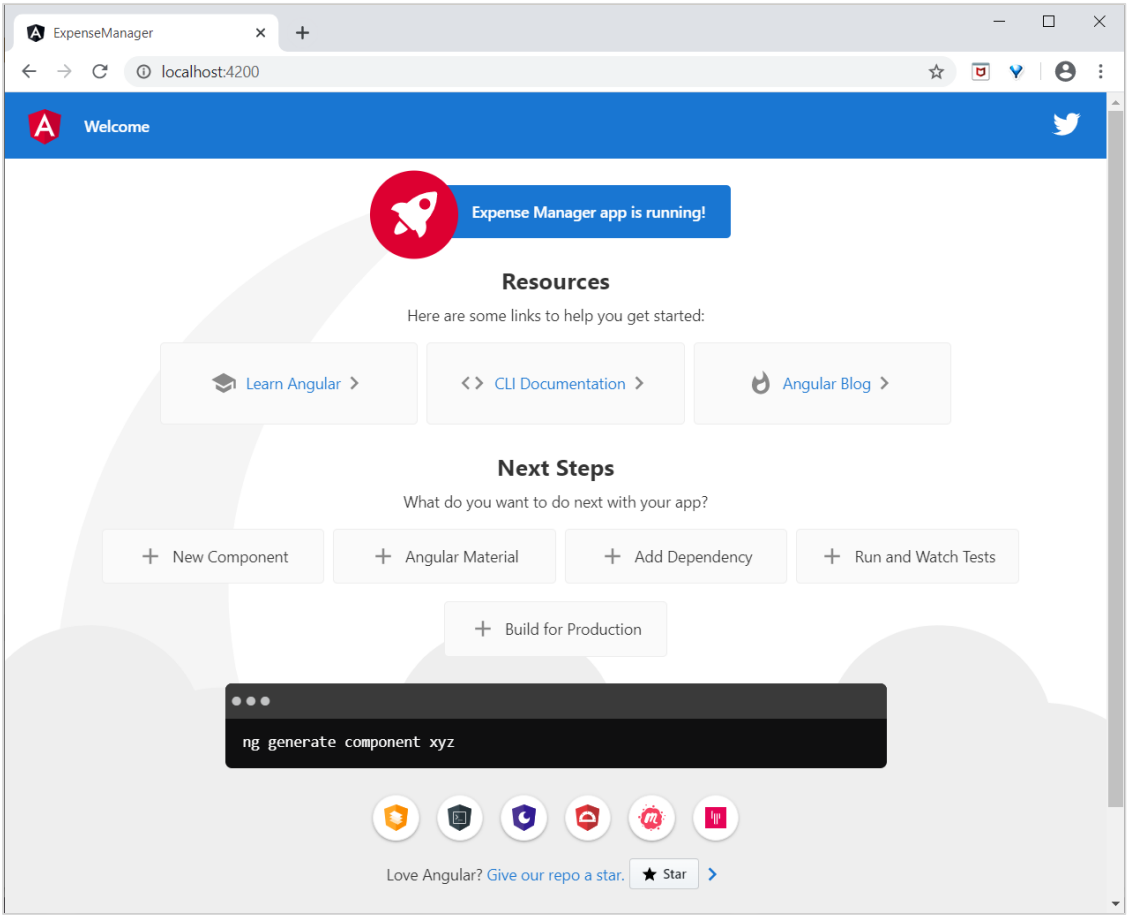
Let us fire up a browser and opens **http://localhost:4200**. The browser will show the application as shown below:



Let us change the title of the application to better reflect our application. Open **src/app/app.component.ts** and change the code as specified below:

```
export class AppComponent {
  title = 'Expense Manager';
}
```

Our final application will be rendered in the browser as shown below:



We will change the application and learn how to code an Angular application in the upcoming chapters.

4. Angular 8 — Architecture

Let us see the architecture of the Angular framework in this chapter.

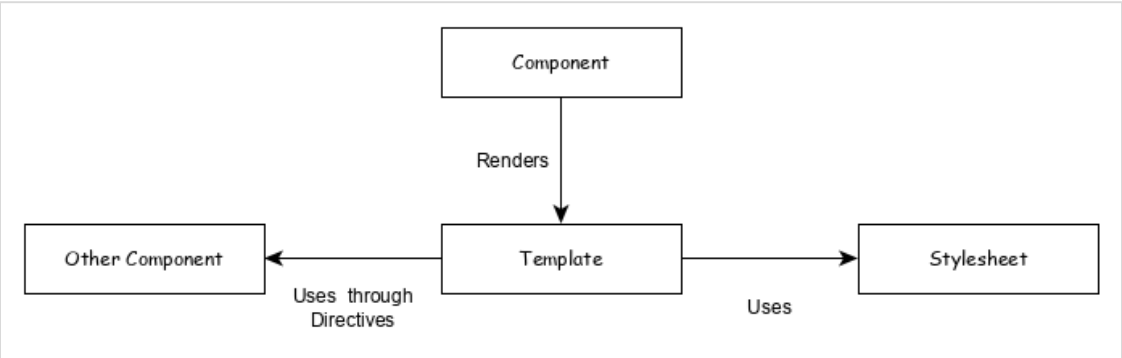
Angular framework is based on four core concepts and they are as follows:

- Components.
- Templates with **Data binding** and **Directives**.
- Modules.
- Services and dependency injection.

Component

The core of the Angular framework architecture is **Angular Component**. Angular Component is the building block of every Angular application. Every angular application is made up of one more **Angular Component**. It is basically a plain JavaScript / Typescript class along with a HTML template and an associated name.

The HTML template can access the data from its corresponding JavaScript / Typescript class. Component's HTML template may include other component using its selector's value (name). The Angular Component may have an optional CSS Styles associated it and the HTML template may access the CSS Styles as well.



Let us analyse the **AppComponent** component in our **ExpenseManager** application. The **AppComponent** code is as follows:

```
// src/app/app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Expense Manager';
}
```

@Component is a decorator and it is used to convert a normal Typescript class to **Angular Component.**

app-root is the selector / name of the component and it is specified using **selector** meta data of the component's decorator. **app-root** can be used by application root document, **src/index.html** as specified below:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ExpenseManager</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

app.component.html is the HTML template document associated with the component. The component template is specified using **templateUrl** meta data of the **@Component** decorator.

app.component.css is the CSS style document associated with the component. The component style is specified using **styleUrls** meta data of the **@Component** decorator.

AppComponent property (**title**) can be used in the HTML template as mentioned below:

```
{{ title }}
```

Template

Template is basically a super set of HTML. Template includes all the features of HTML and provides additional functionality to bind the component data into the HTML and to dynamically generate HTML DOM elements.

The core concept of the template can be categorised into two items and they are as follows:

Data binding

Used to bind the data from the component to the template.

```
{{ title }}
```

Here, **title** is a property in **AppComponent** and it is bind to template using **Interpolation.**

Directives

Used to include logic as well as enable creation of complex HTML DOM elements.

```
<p *ngIf="canShow">
```

```
    This sectiom will be shown only when the *canShow* propery's value in
    the corresponding component is *true*
</p>
<p [showToolTip]='tips' />
```

Here, **ngIf** and **showToolTip** (just an example) are directives. **ngIf** create the paragraph DOM element only when **canShow** is true. Similarly, **showToolTip** is **Attribute Directives**, which adds the tooltip functionality to the paragraph element.

When user mouse over the paragraph, a tooltip with be shown. The content of the tooltip comes from **tips** property of its corresponding component.

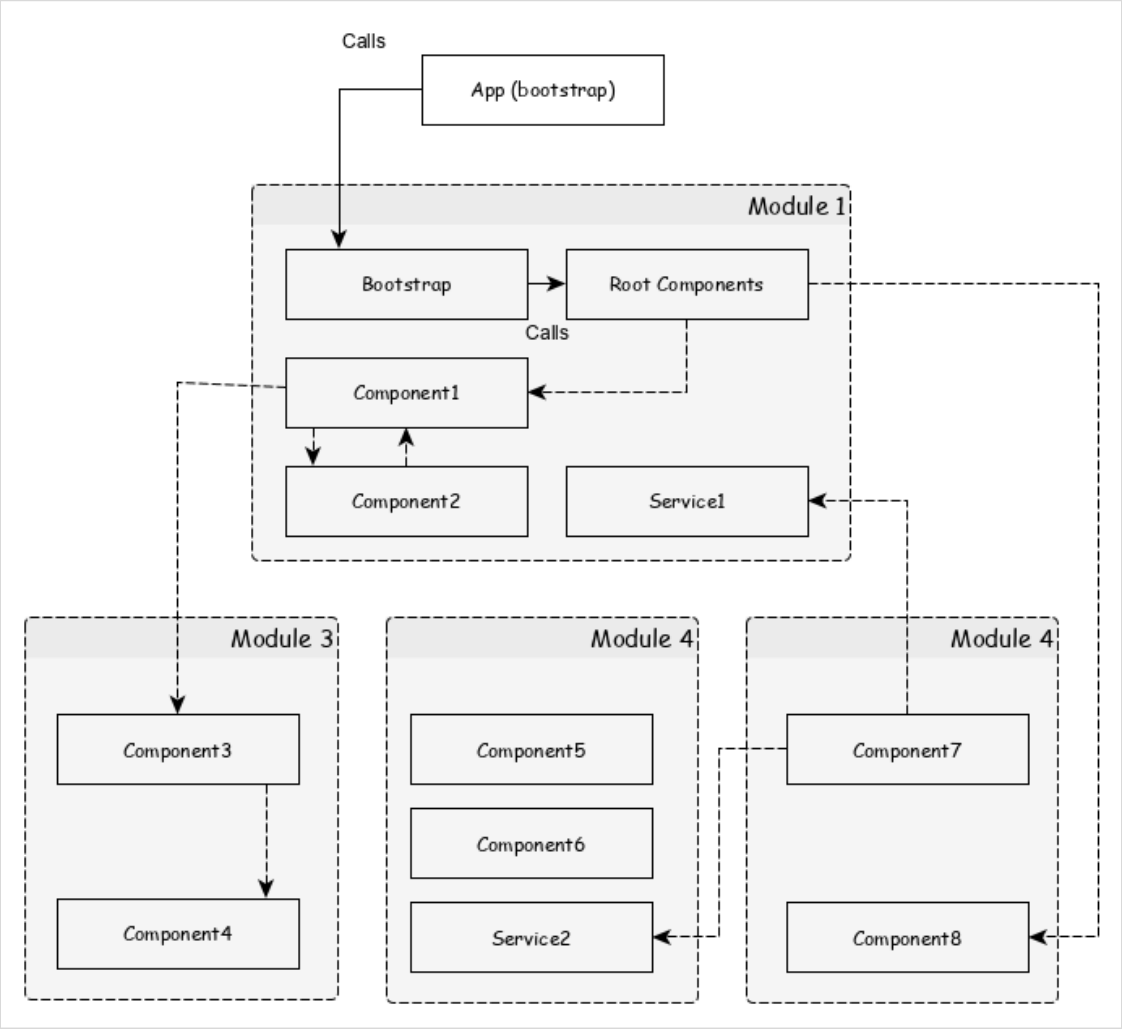
Modules

Angular Module is basically a collection of related features / functionality. **Angular Module** groups multiple components and services under a single context.

For example, animations related functionality can be grouped into single module and Angular already provides a module for the animation related functionality, **BrowserAnimationModule** module.

An Angular application can have any number of modules but only one module can be set as root module, which will bootstrap the application and then call other modules as and when necessary. A module can be configured to access functionality from other module as well. In short, components from any modules can access component and services from any other modules.

Following diagram depicts the interaction between modules and its components.



Let us check the root module of our **Expense Manager** application.

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

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

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

Here,

- **NgModule** decorator is used to convert a plain Typescript / JavaScript class into **Angular module.**

- **declarations** option is used to include components into the **AppModule** module.
- **bootstrap** option is used to set the root component of the **AppModule** module.
- **providers** option is used to include the services for the **AppModule** module.
- **imports** option is used to import other modules into the **AppModule** module.

The following diagram depicts the relationship between Module, Component and Services.

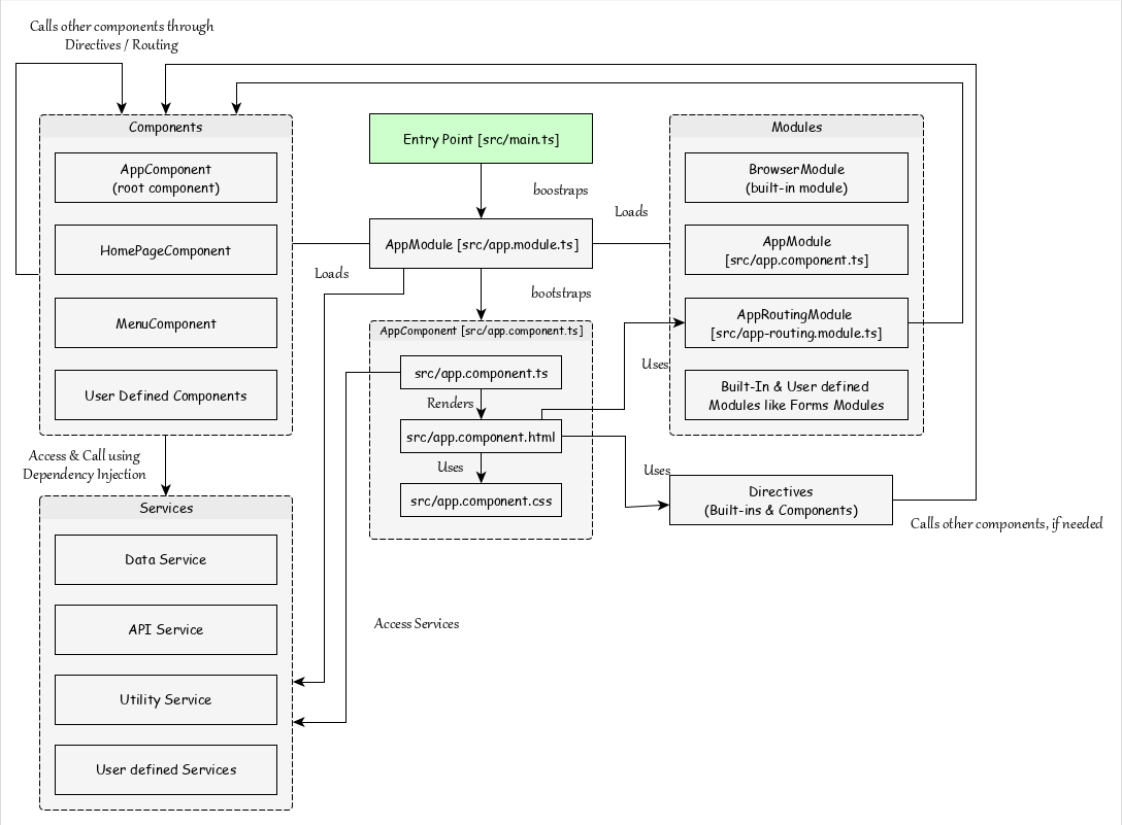
Services

Services are plain Typescript / JavaScript class providing a very specific functionality. **Services** will do a single task and do it best. The main purpose of the service is reusability. Instead of writing a functionality inside a component, separating it into a service will make it useable in other component as well.

Also, **Services** enables the developer to organise the business logic of the application. Basically, component uses services to do its own job. **Dependency Injection** is used to properly initialise the service in the component so that the component can access the services as and when necessary without any setup.

Workflow of Angular application

We have learned the core concepts of Angular application. Let us see the complete flow of a typical Angular application.



src/main.ts is the entry point of Angular application.

src/main.ts bootstraps the AppModule (src/app.module.ts), which is the root module for every Angular application.

```
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

AppModule bootstraps the **AppComponent** *(*src/app.component.ts*)*, which is the root component of every Angular application.

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

Here,

AppModule loads modules through **imports** option.

AppModule also loads all the registered service using **Dependency Injection (DI)** framework.

AppComponent renders its template (**src/app.component.html**) and uses the corresponding styles (**src/app.component.css**). **AppComponent** name, **app-root** is used to place it inside the **src/index.html**.

```
<!doctype html>
<html lang="en">
        <head>
                <meta charset="utf-8">
                <title>ExpenseManager</title>
                <base href="/">
                <meta name="viewport" content="width=device-width, initial-
scale=1">
                <link rel="icon" type="image/x-icon" href="favicon.ico">
        </head>
        <body>
                <app-root></app-root>
        </body>
</html>
```

AppComponent can use any other components registered in the application.

```
@NgModule({
        declarations: [

            AppComponent

            AnyOtherComponent
```

```
        ],
        imports: [
            BrowserModule
        ],
        providers: [],
        bootstrap: [AppComponent]
})
export class AppModule { }
```

Component use other component through directive in its template using target component's selector name.

```
<component-selector-name></component-selector-name>
```

Also, all registered services are accessible to all Angular components through **Dependency Injection (DI)** framework.

5. Angular 8 — Angular Components and Templates

As we learned earlier, **Components** are building block of Angular application. The main job of Angular Component is to generate a section of web page called **view**. Every component will have an associated template and it will be used to generate views.

Let us learn the basic concept of component and template in this chapter.

Add a component

Let us create a new component in our **ExpenseManager** application.

Open command prompt and go to **ExpenseManager** application.

```
cd /go/to/expense-manager
```

Create a new component using **ng generate component** command as specified below:

```
ng generate component expense-entry
```

Output

The output is mentioned below:

```
CREATE src/app/expense-entry/expense-entry.component.html (28 bytes)
CREATE src/app/expense-entry/expense-entry.component.spec.ts (671 bytes)
CREATE src/app/expense-entry/expense-entry.component.ts (296 bytes)
CREATE src/app/expense-entry/expense-entry.component.css (0 bytes)
UPDATE src/app/app.module.ts (431 bytes)
```

Here,

- **ExpenseEntryComponent** is created under src/app/expense-entry folder.
- Component class, Template and stylesheet are created.
- AppModule is updated with new component.

Add title property to **ExpenseEntryComponent** (src/app/expense-entry/expense-entry.component.ts) component.

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

@Component({
        selector: 'app-expense-entry',
        templateUrl: './expense-entry.component.html',
        styleUrls: ['./expense-entry.component.css']
})
export class ExpenseEntryComponent implements OnInit {
```

```
        title: string;

        constructor() { }

        ngOnInit() {
                this.title = "Expense Entry"
        }
}
```

Update template, **src/app/expense-entry/expense-entry.component.html** with below content.

```
<p>{{ title }}</p>
```

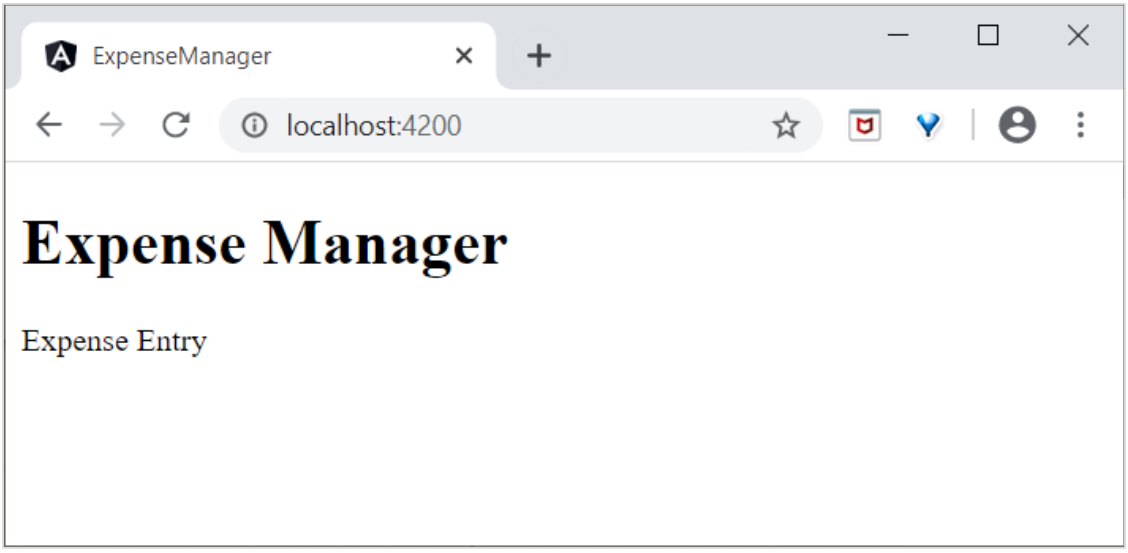
Open **src/app/app.component.html** and include newly created component.

```
<h1>{{ title }}</h1>
<app-expense-entry></app-expense-entry>
```

Here,

app-expense-entry is the selector value and it can be used as regular HTML Tag.

Finally, the output of the application is as shown below:



We will update the content of the component during the course of learning more about templates.

Templates

The integral part of Angular component is **Template**. It is used to generate the HTML content. **Templates** are plain HTML with additional functionality.

Attach a template

Template can be attached to Angular component using **@component** decorator's meta data. Angular provides two meta data to attach template to components.

templateUrl

We already know how to use templateUrl. It expects the relative path of the template file. For example, AppComponent set its template as app.component.html.

```
  templateUrl: './app.component.html',
```

template

template enables to place the HTML string inside the component itself. If the template content is minimal, then it will be easy to have it **Component** class itself for easy tracking and maintenance purpose.

```
@Component({
  selector: 'app-root',
  templateUrl: `<h1>{{ title }}</h1>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'Expense Manager';

  constructor(private debugService : DebugService) {}

  ngOnInit() {
          this.debugService.info("Angular Application starts");
  }
}
```

Attach Stylesheet

Angular Templates can use CSS styles similar to HTML. Template gets its style information from two sources, a) from its component b) from application configuration.

Component configuration

Component decorator provides two option, **styles** and **styleUrls** to provide CSS style information to its template.

- Styles: **styles** option is used to place the CSS inside the component itself.

```
styles: ['h1 { color: '#ff0000'; }']
```

- styleUrls: **styleUrls** is used to refer external CSS stylesheet. We can use multiple stylesheet as well.

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

Application configuration

Angular provides an option in project configuration (**angular.json**) to specify the CSS stylesheets. The styles specified in **angular.json** will be applicable for all templates. Let us check our **angular.json** as shown below:

```
{

  "projects": {
    "expense-manager": {
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/expense-manager",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "aot": false,
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
            "styles": [
              "src/styles.css"
            ],
            "scripts": []
          },
        },
      }
    }},
  "defaultProject": "expense-manager"
}
```

Here,

styles option sets **src/styles.css** as global CSS stylesheet. We can include any number of CSS stylesheets as it supports multiple values.

Include bootstrap

Let us include bootstrap into our **ExpenseManager** application using **styles** option and change the default template to use bootstrap components.

Open command prompt and go to ExpenseManager application.

```
cd /go/to/expense-manager
```

Install **bootstrap** and **JQuery** library using below commands:

```
npm install --save bootstrap@4.5.0 jquery@3.5.1
```

Here,

We have installed JQuery, because, bootstrap uses jquery extensively for advanced components.

Option **angular.json** and set bootstrap and jquery library path.

```
{
  "projects": {
    "expense-manager": {
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/expense-manager",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "aot": false,
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
            "styles": [
              "./node_modules/bootstrap/dist/css/bootstrap.css",
              "src/styles.css"
            ],
            "scripts": [
              "./node_modules/jquery/dist/jquery.js",
              "./node_modules/bootstrap/dist/js/bootstrap.js"
            ]
          },
        },
      }
    }},
  "defaultProject": "expense-manager"
}
```

Here,

scripts option is used to include JavaScript library. **JavaScript** registered through **scripts** will be available to all Angular components in the application.

Open **app.component.html** and change the content as specified below:

```
<!-- Navigation -->
<nav class="navbar navbar-expand-lg navbar-dark bg-dark static-top">
    <div class="container">
        <a class="navbar-brand" href="#">{{ title }}</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarResponsive" aria-controls="navbarResponsive" aria-
expanded="false" aria-label="Toggle navigation">
                <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarResponsive">
                <ul class="navbar-nav ml-auto">
                        <li class="nav-item active">
                        <a class="nav-link" href="#">Home


                                <span class="sr-only">(current)</span>
```

```
                        </a>
                        </li>
                        <li class="nav-item">
                        <a class="nav-link" href="#">Report</a>
                        </li>
                        <li class="nav-item">
                        <a class="nav-link" href="#">Add Expense</a>
                        </li>
                        <li class="nav-item">
                        <a class="nav-link" href="#">About</a>
                        </li>
                </ul>
        </div>
    </div>
</nav>

<app-expense-entry></app-expense-entry>
```

Here,

Used bootstrap navigation and containers.

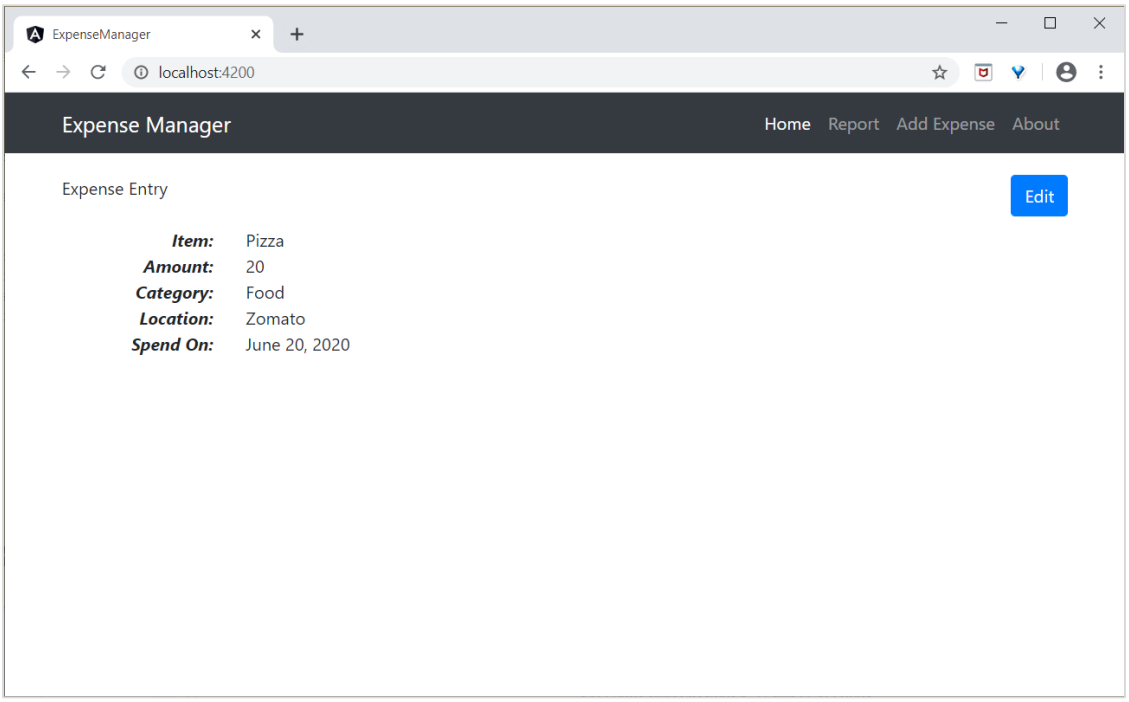
Open **src/app/expense-entry/expense-entry.component.html** and place below content.

```
<!-- Page Content -->
<div class="container">
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">
                        <div class="container" style="padding-left: 0px;
padding-right: 0px;">
                                <div class="row">
                                <div class="col-sm" style="text-align: left;">
                                        {{ title }}
                                </div>
                                <div class="col-sm" style="text-align: right;">
                                                <button type="button"
class="btn btn-primary">Edit</button>
                                </div>
                                </div>
                        </div>
                        <div class="container box" style="margin-top: 10px;">
                                <div class="row">
                                <div class="col-2" style="text-align: right;">
                                        <strong><em>Item:</em></strong>
                                </div>
                                <div class="col" style="text-align: left;">
                                        Pizza
                                </div>


                                </div>
                                <div class="row">
                                <div class="col-2" style="text-align: right;">
```

```
                                    <strong><em>Amount:</em></strong>
                                </div>
                                <div class="col" style="text-align: left;">
                                    20
                                </div>
                                </div>
                                <div class="row">
                                <div class="col-2" style="text-align: right;">
                                    <strong><em>Category:</em></strong>
                                </div>
                                <div class="col" style="text-align: left;">
                                    Food
                                </div>
                                </div>
                                <div class="row">
                                <div class="col-2" style="text-align: right;">
                                    <strong><em>Location:</em></strong>
                                </div>
                                <div class="col" style="text-align: left;">
                                   Zomato
                                </div>
                                </div>
                                <div class="row">
                                <div class="col-2" style="text-align: right;">
                                    <strong><em>Spend On:</em></strong>
                                </div>
                                <div class="col" style="text-align: left;">
                                    June 20, 2020
                                </div>
                                </div>
                            </div>
        </div>
    </div>
</div>
```

Restart the application.

The output of the application is as follows:



We will improve the application to handle dynamic expense entry in next chapter.

6. Angular 8 — Data Binding

Data binding deals with how to bind your data from component to HTML DOM elements (Templates). We can easily interact with application without worrying about how to insert your data. We can make connections in two different ways one way and two-way binding.

Before moving to this topic, let's create a component in Angular 8.

Open command prompt and create new Angular application using below command:

```
cd /go/to/workspace
ng new databind-app
cd databind-app
```

Create a **test** component using Angular CLI as mentioned below:

```
ng generate component test
```

The above create a new component and the output is as follows:

```
CREATE src/app/test/test.component.scss (0 bytes)
CREATE src/app/test/test.component.html (19 bytes)
CREATE src/app/test/test.component.spec.ts (614 bytes)
CREATE src/app/test/test.component.ts (262 bytes)
UPDATE src/app/app.module.ts (545 bytes)
```

Run the application using below command:

```
ng serve
```

One-way data binding

One-way data binding is a one-way interaction between component and its template. If you perform any changes in your component, then it will reflect the HTML elements. It supports the following types:

String interpolation

In general, **String interpolation** is the process of formatting or manipulating strings. In Angular, **Interpolation** is used to display data from component to view (DOM). It is denoted by the expression of **{{ }}** and also known as mustache syntax.

Let's create a simple string property in component and bind the data to view.

Add the below code in **test.component.ts** file as follows:

```
export class TestComponent implements OnInit {
  appName = "My first app in Angular 8";
}
```

Move to **test.component.html** file and add the below code:

```
<h1>{{appName}}</h1>
```

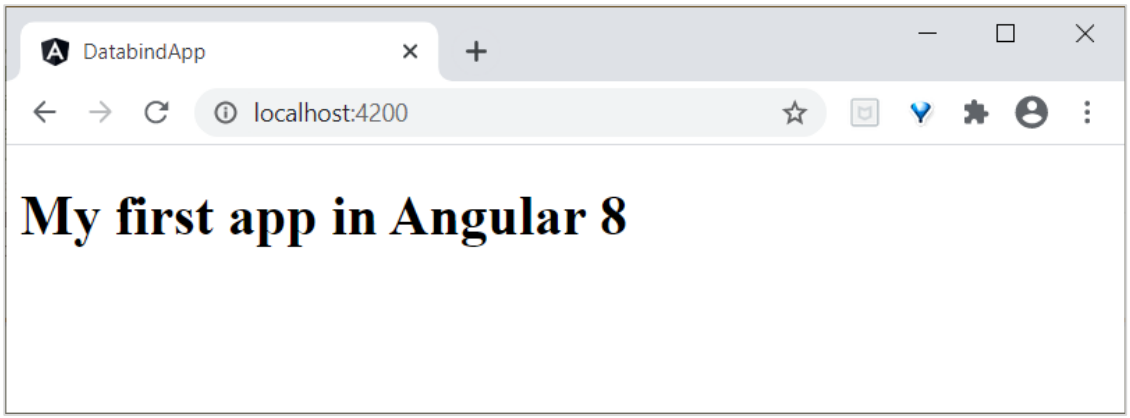
Add the test component in your **app.component.html** file by replacing the existing content as follows:

```
<app-test></app-test>
```

Finally, start your application (if not done already) using the below command:

```
ng serve
```

You could see the following output on your screen:



Event binding

Events are actions like mouse click, double click, hover or any keyboard and mouse actions. If a user interacts with an application and performs some actions, then event will be raised. It is denoted by either parenthesis **()** or **on-**. We have different ways to bind an event to DOM element. Let's understand one by one in brief.

Component to view binding

Let's understand how simple button click even handling works.

Add the following code in **test.component.ts** file as follows:

```
export class TestComponent {

showData($event: any){
        console.log("button is clicked!");
        if($event) {
                console.log($event.target);
                console.log($event.target.value);
        }
    }
}
```

$event * refers the fired event. In this scenario, * click * is the event. * $event* has all the information about event and the target element. Here, the target is **button**. **$event.target** property will have the target information.

We have two approaches to call the component method to view (**test.component.html**). First one is defined below:

```
<h2>Event Binding</h2>

<button (click)="showData($event)">Click here</button>
```

Alternatively, you can use **prefix - on** using canonical form as shown below:

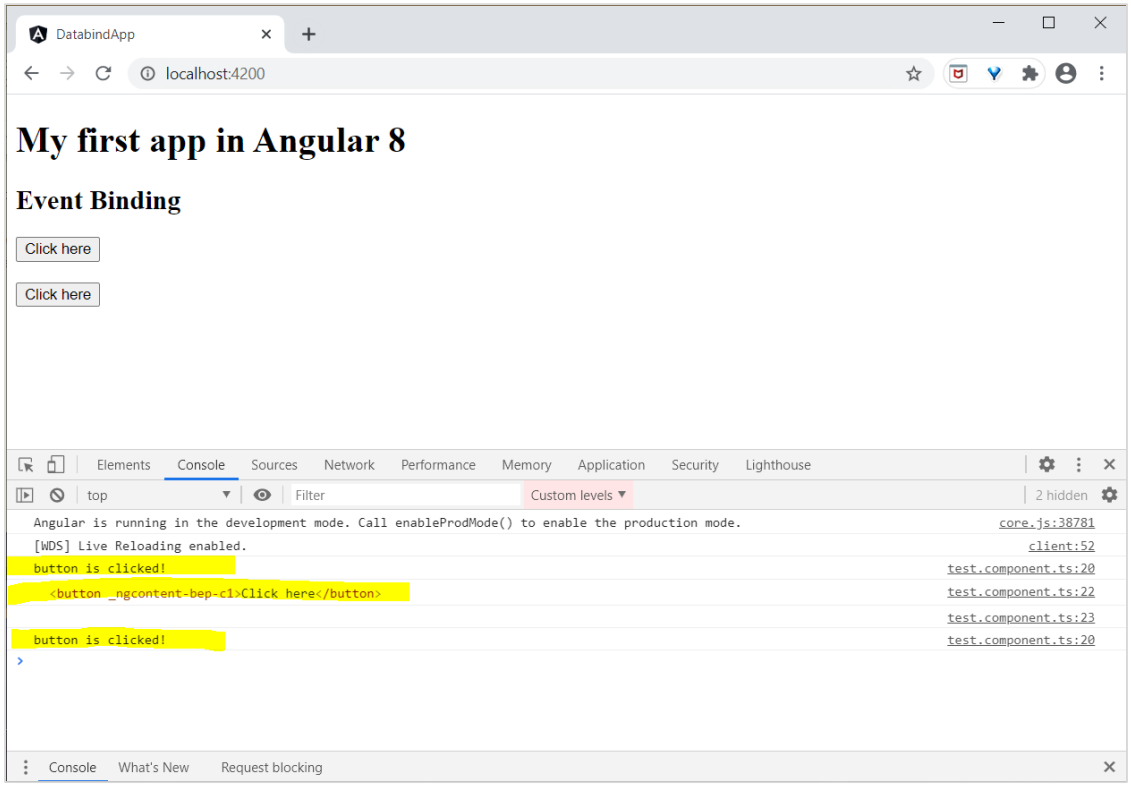
```
<button on-click = "showData()">Click here</button>
```

Here, we have not used **$event** as it is optional.

Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



Here, when the user clicks on the button, event binding understands to button click action and call component **showData()** method so we can conclude it is one-way binding.

Property binding

Property binding is used to bind the data from property of a component to DOM elements. It is denoted by **[]**.

Let's understand with a simple example.

Add the below code in **test.component.ts** file.

```
export class TestComponent {
  userName:string = "Peter";
}
```

Add the below changes in view test.component.html,

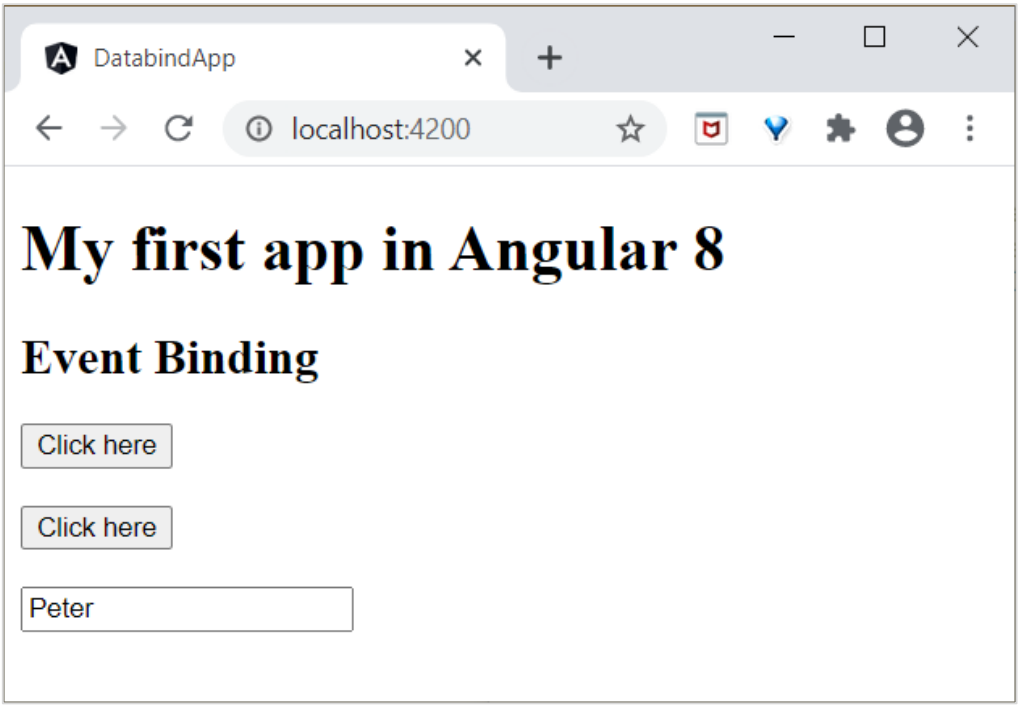
```
<input type="text" [value]="userName">
```

Here,

userName property is bind to an attribute of a DOM element **<input>** tag.

Finally, start your application (if not done already) using the below command:

```
ng serve
```



Attribute binding

Attribute binding is used to bind the data from component to HTML attributes. The syntax is as follows:

```
<HTMLTag [attr.ATTR]="Component data">
```

For example,

```
<td [attr.colspan]="columnSpan"> ... </td>
```

Let's understand with a simple example.

Add the below code in **test.component.ts** file.

```
export class TestComponent {
  userName:string = "Peter";
}
```

Add the below changes in view **test.component.html**,

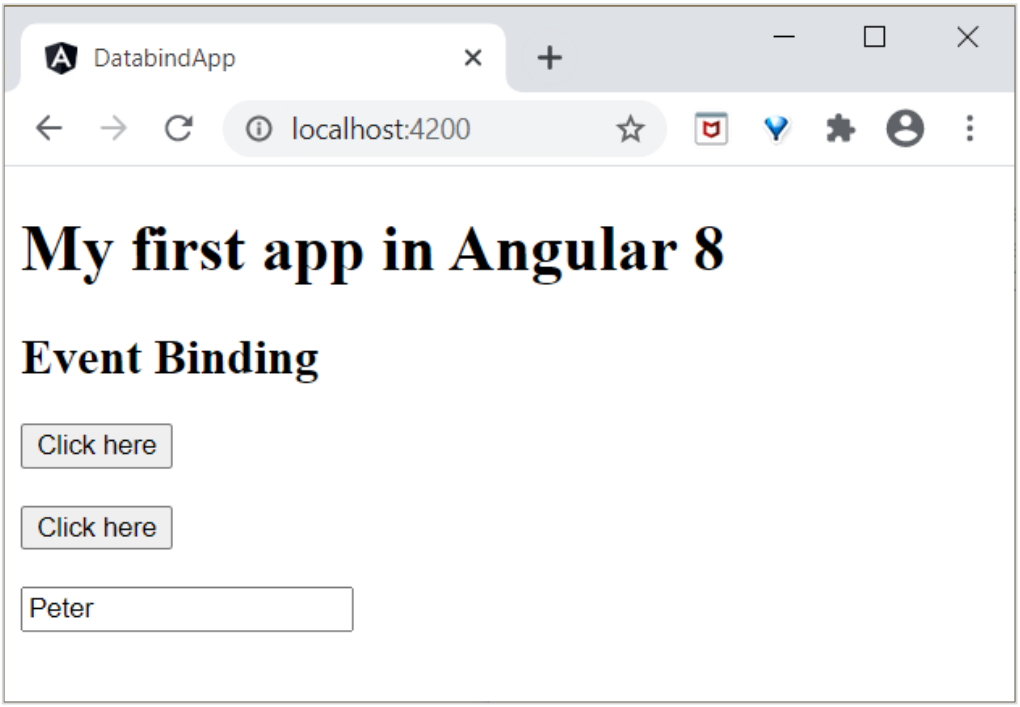
```
<input type="text" [value]="userName">
```

Here,

userName property is bind to an attribute of a DOM element **<input>** tag.

Finally, start your application (if not done already) using the below command:

```
ng serve
```



Class binding

Class binding is used to bind the data from component to HTML class property. The syntax is as follows:

```
<HTMLTag [class]="component variable holding class name">
```

Class Binding provides additional functionality. If the component data is boolean, then the class will bind only when it is true. Multiple class can be provided by string ("foo bar") as well as Array of string. Many more options are available.

For example,

```
<p [class]="myClasses">
```

Let's understand with a simple example.

Add the below code in **test.component.ts** file,

```
export class TestComponent {
myCSSClass = "red";
applyCSSClass = false;
}
```

Add the below changes in view **test.component.html.**

```
<p [class]="myCSSClass">This paragraph class comes from *myClass* property </p>
<p [class.blue]="applyCSSClass">This paragraph class does not apply</p>
```

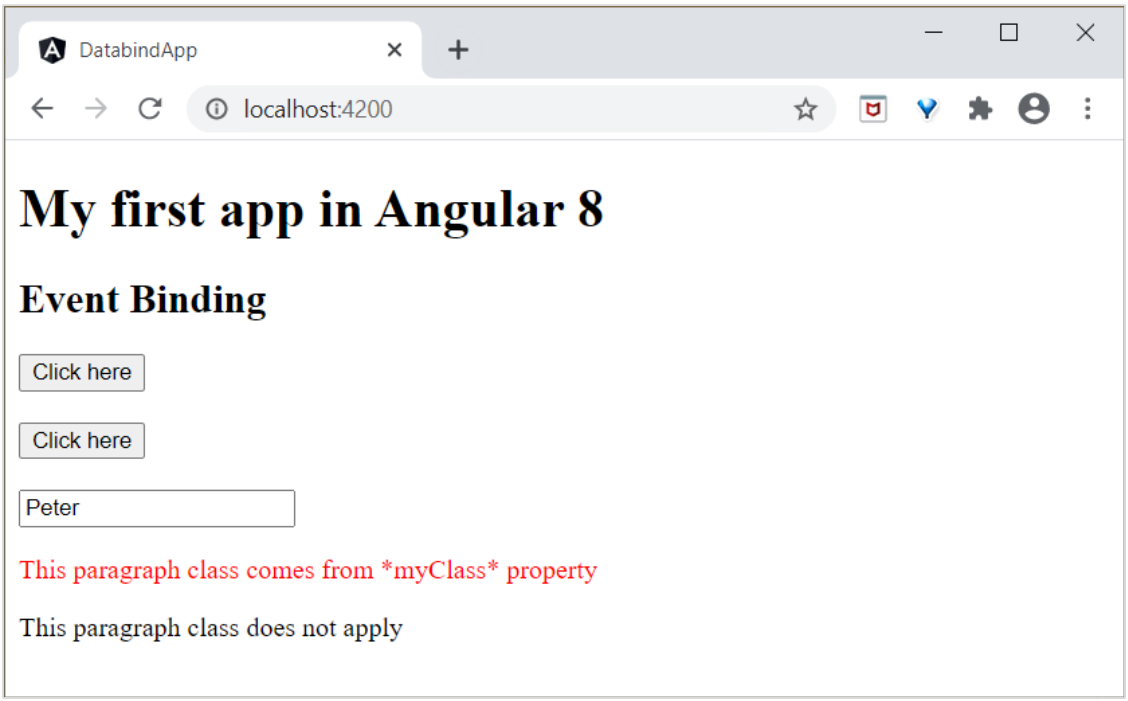
Add the below content in **test.component.css.**

```
.red {
        color: red;
}

.blue {
        color: blue;
}
```

Finally, start your application (if not done already) using the below command:

```
ng serve
```

The final output will be as shown below:



Style binding

Style binding is used to bind the data from component into HTML style property. The syntax is as follows:

```
<HTMLTag [style.STYLE]="component data">
```

For example,

```
<p [style.color]="myParaColor"> ... </p>
```

Let's understand with a simple example.

Add the below code in **test.component.ts** file.

```
myColor = 'brown';
```

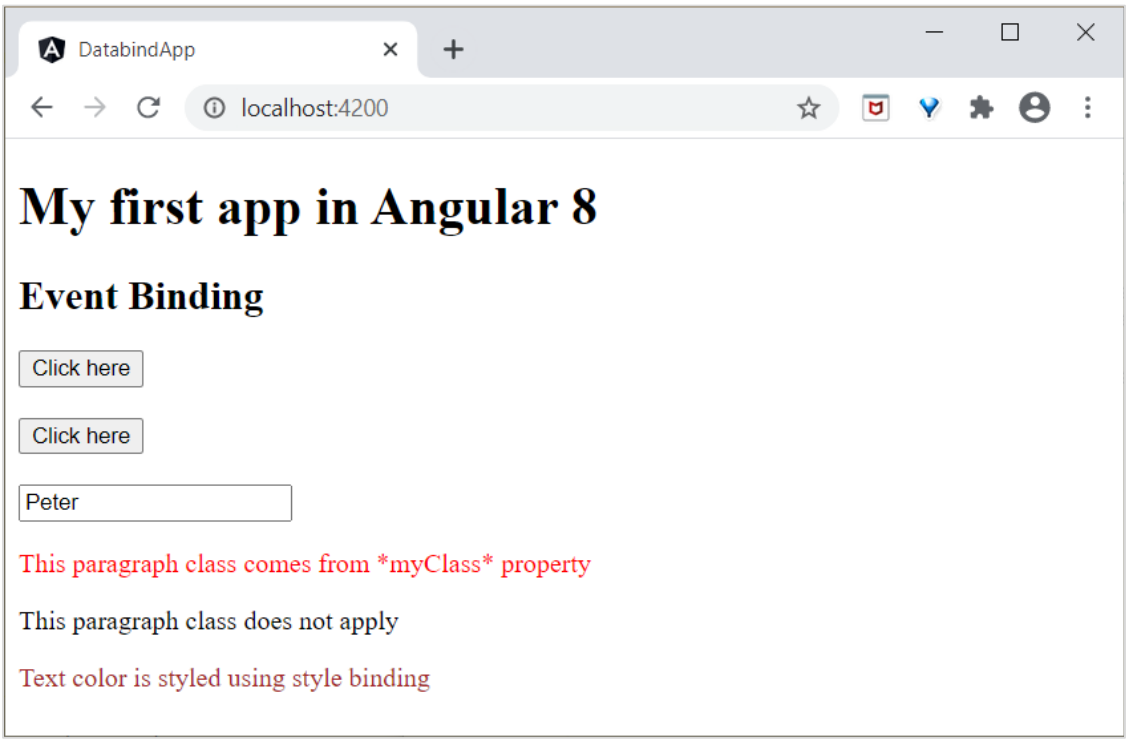
Add the below changes in view **test.component.html**.

```
<p [style.color]="myColor">Text color is styled using style binding</p>
```

Finally, start your application (if not done already) using the below command:

```
ng serve
```

The final output will be as shown below:



Two-way data binding

Two-way data binding is a two-way interaction, data flows in both ways (from component to views and views to component). Simple example is **ngModel.** If you do any changes in your property (or model) then, it reflects in your view and vice versa. It is the combination of property and event binding.

NgModel

NgModel is a standalone directive. **ngModel** directive binds form control to property and property to form control. The syntax of **ngModel** is as follows:

```
<HTML [(ngModel)]="model.name" />
```

For example,

```
<input type="text" [(ngModel)]="model.name" />
```

Let's try to use **ngModel** in our test application.

Configure **FormsModule** in **AppModule** (src/app/app.module.ts)

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule



  ]
})
export class AppModule { }
```

FormModule do the necessary setup to enable two-way data binding.

Update **TestComponent** view (**test.component.html**) as mentioned below:

```
<input type="text" [(ngModel)]="userName">

<p>Two way binding! Hello {{ userName }}!</p>
```

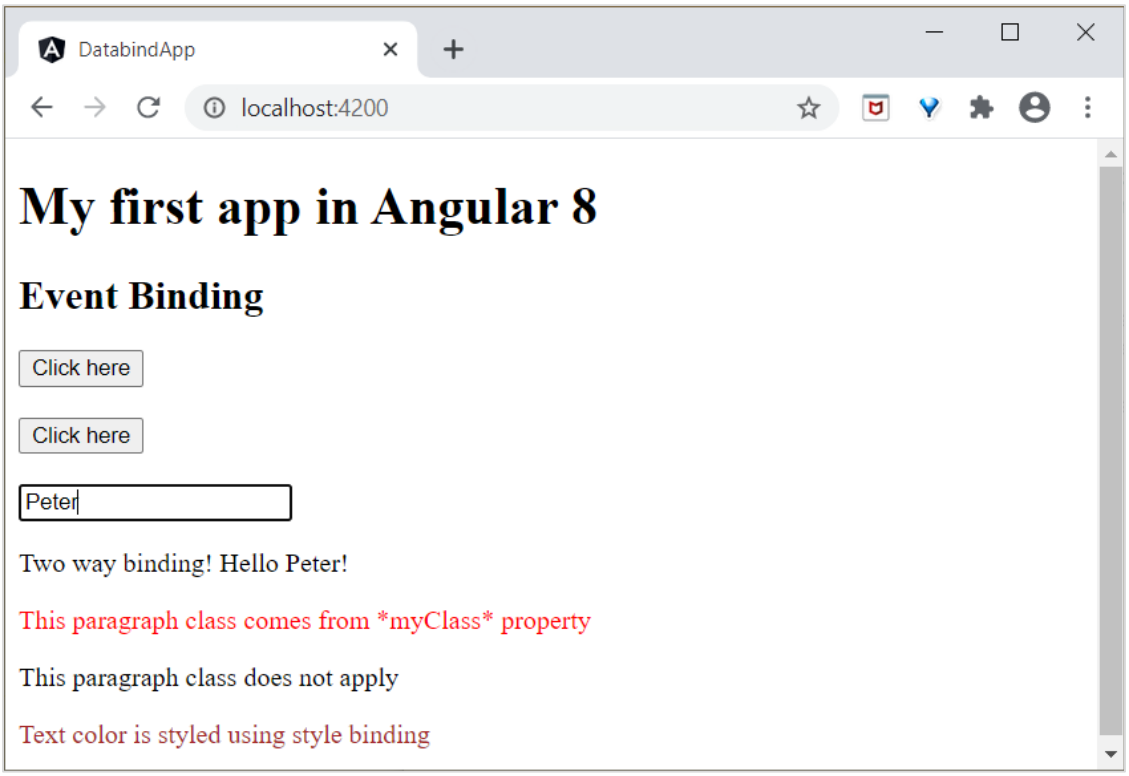
Here,

Property is bind to form control **ngModel** directive and if you enter any text in the textbox, it will bind to the property. After running your application, you could see the below changes:

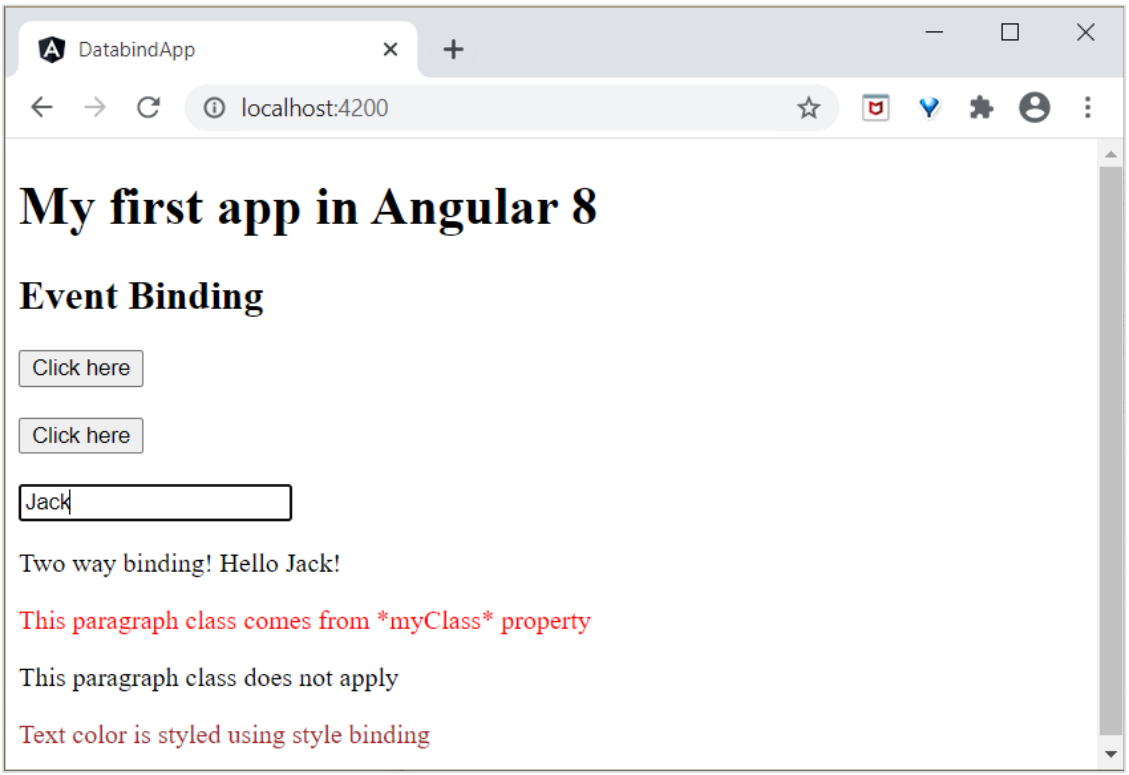
Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



Now, try to change the input value to **Jack.** As you type, the text below the input gets changed and the final output will be as shown below:



We will learn more about form controls in the upcoming chapters.

Working example

Let us implement all the concept learned in this chapter in our **ExpenseManager** application.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Create **ExpenseEntry** interface (src/app/expense-entry.ts) and add **id, amount, category, Location, spendOn** and **createdOn**.

```
export interface ExpenseEntry {
    id: number;
    item: string;
    amount: number;
    category: string;
    location: string;
    spendOn: Date;
    createdOn: Date;
}
```

Import **ExpenseEntry** into **ExpenseEntryComponent.**

```
import { ExpenseEntry } from '../expense-entry';
```

Create a **ExpenseEntry** object, **expenseEntry** as shown below:

```
export class ExpenseEntryComponent implements OnInit {
        title: string;
        expenseEntry: ExpenseEntry;
        constructor() { }

        ngOnInit() {
                this.title = "Expense Entry";

                this.expenseEntry = {
                        id: 1,
                        item: "Pizza",
                        amount: 21,
                        category: "Food",
                        location: "Zomato",
                        spendOn: new Date(2020, 6, 1, 10, 10, 10),
                        createdOn: new Date(2020, 6, 1, 10, 10, 10),
                };
        }
}
```

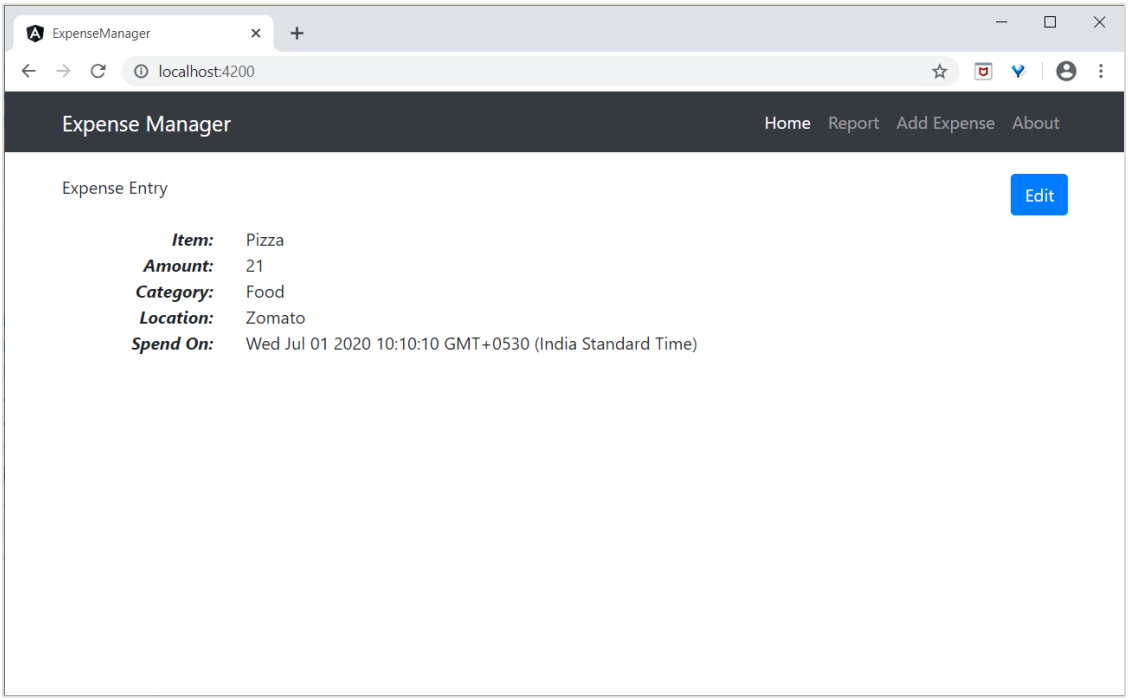
Update the component template using **expenseEntry** object, **src/app/expense-entry/expense-entry.component.html** as specified below:

```
<!-- Page Content -->
<div class="container">
```

```
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">
            <div class="container" style="padding-left: 0px; padding-right:
0px;">
                <div class="row">
                    <div class="col-sm" style="text-align: left;">
                        {{ title }}
                    </div>
                    <div class="col-sm" style="text-align: right;">
                        <button type="button" class="btn btn-
primary">Edit</button>
                    </div>
                </div>
            </div>
            <div class="container box" style="margin-top: 10px;">
                <div class="row">
                    <div class="col-2" style="text-align: right;">
                        <strong><em>Item:</em></strong>
                    </div>
                    <div class="col" style="text-align: left;">
                        {{ expenseEntry.item }}
                    </div>
                </div>
                <div class="row">
                    <div class="col-2" style="text-align: right;">
                        <strong><em>Amount:</em></strong>
                    </div>
                    <div class="col" style="text-align: left;">

                        {{ expenseEntry.amount }}
                    </div>
                </div>
                <div class="row">
                    <div class="col-2" style="text-align: right;">
                        <strong><em>Category:</em></strong>
                    </div>
                    <div class="col" style="text-align: left;">
                        {{ expenseEntry.category }}
                    </div>
                </div>
                <div class="row">
                    <div class="col-2" style="text-align: right;">
                        <strong><em>Location:</em></strong>
                    </div>
                    <div class="col" style="text-align: left;">
                        {{ expenseEntry.location }}
                    </div>
                </div>
                <div class="row">
                    <div class="col-2" style="text-align: right;">

                        <strong><em>Spend On:</em></strong>
```

```
                    </div>
                    <div class="col" style="text-align: left;">
                        {{ expenseEntry.spendOn }}
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

The output of the application is as follows:



7. Angular 8 — Directives

Angular 8 directives are DOM elements to interact with your application. Generally, directive is a **TypeScript** function. When this function executes **Angular** compiler checked it inside DOM element. Angular directives begin with **ng-** where **ng** stands for Angular and extends HTML tags with **@directive** decorator.

Directives enables logic to be included in the Angular templates. Angular directives can be classified into three categories and they are as follows:

Attribute directives

Used to add new attributes for the existing HTML elements to change its look and behaviour.

```
<HTMLTag [attrDirective]='value' />
```

For example,

```
<p [showToolTip]='Tips' />
```

Here, **showToolTip** refers an example directive, which when used in a HTML element will show tips while user hovers the HTML element.

Structural directives

Used to add or remove DOM elements in the current HTML document.

```
<HTMLTag [structuralDirective]='value' />
```

For example,

```
<div *ngIf="isNeeded">
    Only render if the *isNeeded* value has true value.
</div>
```

Here, **ngIf** is a built-in directive used to add or remove the HTML element in the current HTML document. Angular provides many built-in directive and we will learn in later chapters.

Component based directives

Component can be used as directives. Every component has **Input** and **Output** option to pass between component and its parent HTML elements.

```
<component-selector-name [input-reference]="input-value"> ... </component-
selector-name>
```

For example,

```
<list-item [items]="fruits"> ... </list-item>
```

Here, **list-item** is a component and **items** is the input option. We will learn how to create component and advanced usages in the later chapters.

Before moving to this topic, let's create a sample application (**directive-app**) in Angular 8 to work out the learnings.

Open command prompt and create new Angular application using below command:

```
cd /go/to/workspace
ng new directive-app
cd directive-app
```

Create a **test** component using Angular CLI as mentioned below:

```
ng generate component test
```

The above create a new component and the output is as follows:

```
CREATE src/app/test/test.component.scss (0 bytes)
CREATE src/app/test/test.component.html (19 bytes)
CREATE src/app/test/test.component.spec.ts (614 bytes)
CREATE src/app/test/test.component.ts (262 bytes)
UPDATE src/app/app.module.ts (545 bytes)
```

Run the application using below command:

```
ng serve
```

DOM Overview

Let us have a look at DOM model in brief. DOM is used to define a standard for accessing documents. Generally, HTML DOM model is constructed as a tree of objects. It is a standard object model to access html elements.

We can use DOM model in Angular 8 for the below reasons:

- We can easily navigate document structures with DOM elements.
- We can easily add html elements.
- We can easily update elements and its contents.

Structural directives

Structural directives change the structure of **DOM** by adding or removing elements. It is denoted by ***** sign with three pre-defined directives **NgIf, NgFor** and **NgSwitch**. Let's understand one by one in brief.

NgIf directive

NgIf directive is used to display or hide data in your application based on the condition becomes true or false. We can add this to any tag in your template.

Let us try **ngIf** directive in our **directive-app** application.

Add the below tag in **test.component.html.**

```
<p>test works!</p>
<div *ngIf="true">Display data</div>
```

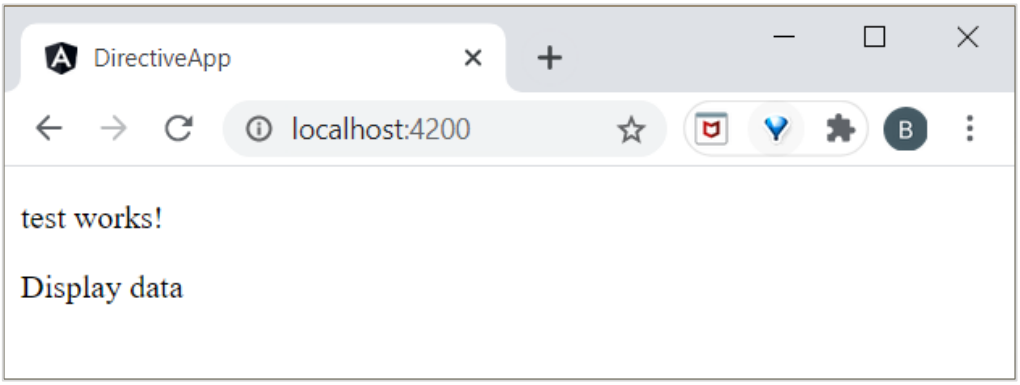
Add the test component in your **app.component.html** file as follows:

```
<app-test></app-test>
```

Start your server (if not started already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



If you set the condition **ngIf="false"** then, contents will be hidden.

ngIfElse directive

ngIfElse is similar to **ngIf** except, it provides option to render content during failure scenario as well.

Let's understand how **ngIfElse** works by doing a sample.

Add the following code in **test.component.ts** file.

```
export class TestComponent implements OnInit {

  isLogIn : boolean = false;

  isLogOut : boolean = true;

}
```

Add the following code in **test.component.html** file as follows:

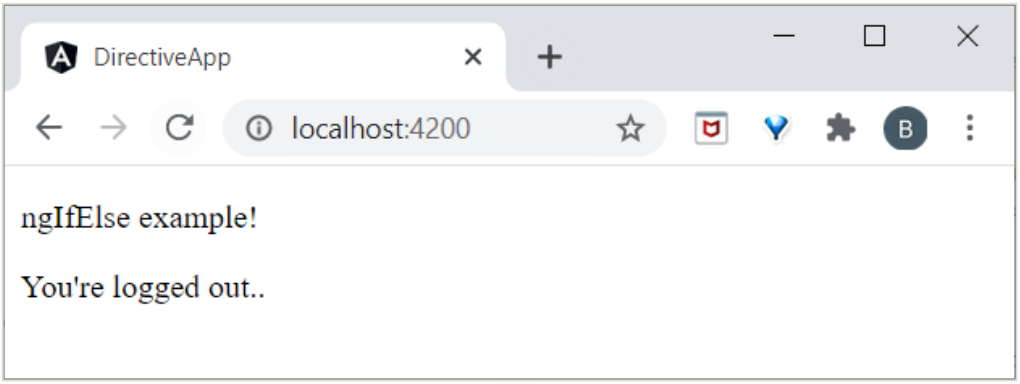
```
<p>ngIfElse example!</p>
<div *ngIf="isLogIn; else isLogOut">
    Hello you are logged in
</div>
```

```
<ng-template #isLogOut>
    You're logged out..
</ng-template>
```

Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



Here, **isLogOut** value is assigned as **true**, so it goes to **else** block and renders **ng-template**. We will learn **ng-template** later in this chapter.

ngFor directive

ngFor is used to repeat a portion of elements from the list of items.

Let's understand how **ngFor** works by doing a sample.

Add the list in **test.component.ts** file as shown below:

```
list = [1,2,3,4,5];
```

Add **ngFor** directive in **test.component.html** as shown below:

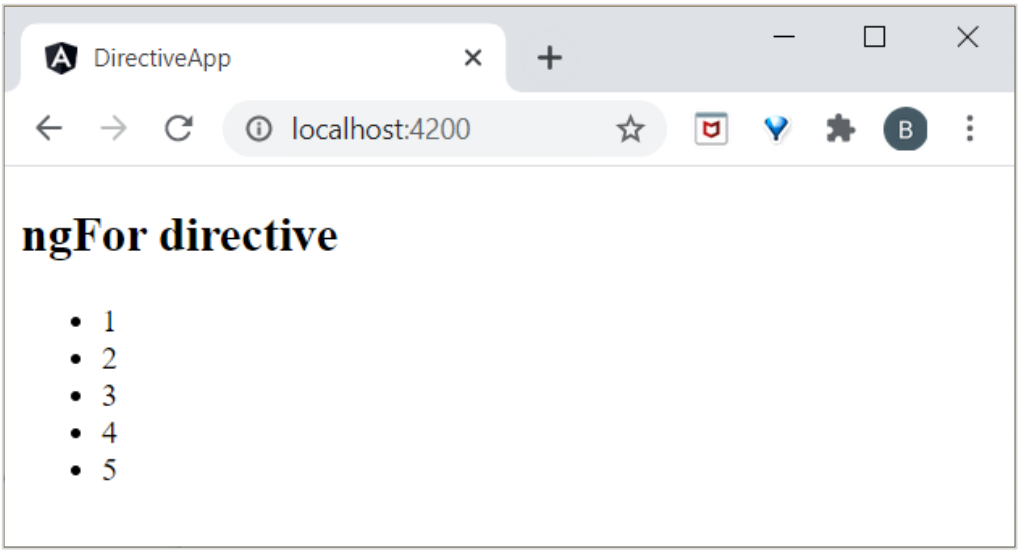
```
<h2>ngFor directive</h2>

<ul>
    <li *ngFor="let l of list">

        {{l}}
    </li>
</ul>
```

Here, the **let** keyword creates a local variable and it can be referenced anywhere in your template. The let / creates a template local variable to get the list elements.

Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



trackBy

Sometimes, **ngFor** performance is low with large lists. For example, when adding new item or remove any item in the list may trigger several DOM manipulations. To iterate over large objects collection, we use **trackBy**.

It is used to track when elements are added or removed. It is performed by trackBy method. It has two arguments index and element. Index is used to identity each element uniquely. Simple example is defined below.

Let's understand how trackBy works along with **ngFor** by doing a sample.

Add the below code in **test.component.ts** file.

```
export class TestComponent
{
  studentArr: any[] = [
    {
      "id": 1,
      "name": "student1"
    },
    {
      "id": 2,

      "name": "student2"
    },
    {
      "id": 3,
      "name": "student3"


    },
```

```
    {
      "id": 4,
      "name": "student4"
    }
  ];

   trackByData(index:number, studentArr:any): number {
    return studentArr.id;
  }
```

Here,

We have created, **trackByData()** method to access each student element in a unique way based on the id.

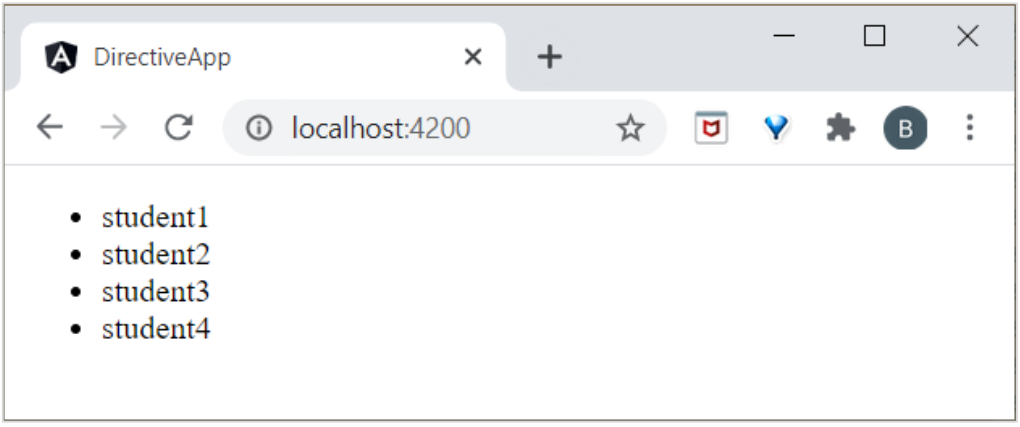
Add the below code in **test.component.html** file to define trackBy method inside ngFor.

```
<ul>
    <li *ngFor="let std of studentArr; trackBy: trackByData">
      {{std.name}}
    </li>
</ul>
```

Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



Here, the application will print the student names. Now, the application is tracking student objects using the student id instead of object references. So, DOM elements are not affected.

NgSwitch directive

NgSWitch is used to check multiple conditions and keep the DOM structure as simple and easy to understand.

Let us try **ngSwitch** directive in our **directive-app** application.

Add the following code in **test.component.ts** file.

```
export class TestComponent implements OnInit {
    logInName = 'admin';
}
```

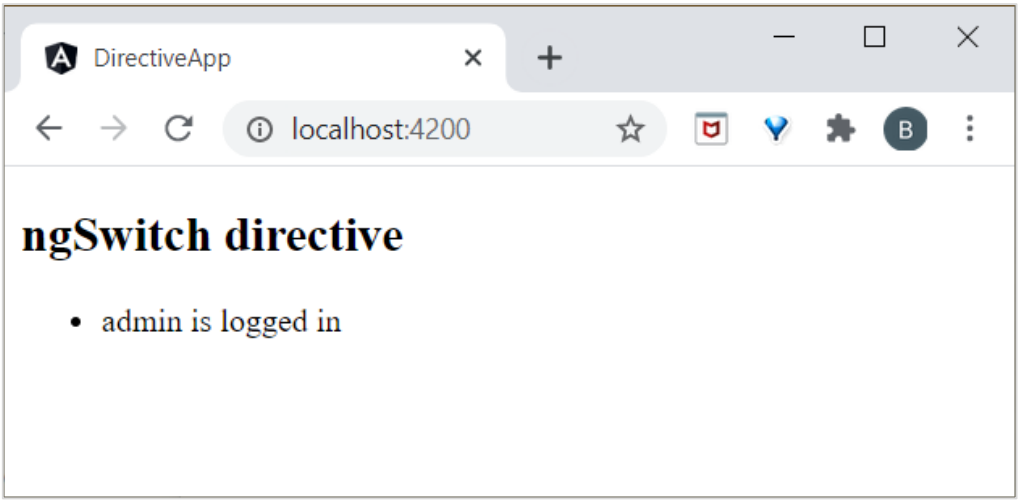
Add the following code in **test.component.html** file as follows:

```
<h2>ngSwitch directive</h2>
<ul [ngSwitch]="logInName">
    <li *ngSwitchCase="'user'">
        <p>User is logged in..</p>
    </li>
    <li *ngSwitchCase="'admin'">
        <p>admin is logged in</p>
    </li>
    <li *ngSwitchDefault>
        <p>Please choose login name</p>
    </li>
</ul>
```

Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



Here, we have defined **logInName** as **admin.** So, it matches second SwitchCase and prints above admin related message.

Attribute directives

Attribute directives performs the appearance or behavior of DOM elements or components. Some of the examples are NgStyle, NgClass and NgModel. Whereas, NgModel is two-way attribute data binding explained in previous chapter.

ngStyle

ngStyle directive is used to add dynamic styles. Below example is used to apply *blue* color to the paragraph.

Let us try **ngStyle** directive in our **directive-app** application.

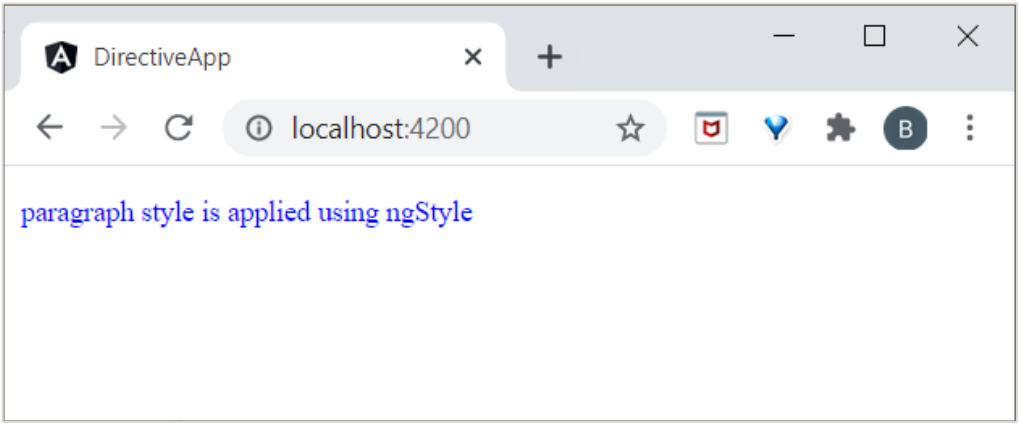
Add below content in **test.component.html** file.

```
<p [ngStyle]="{'color': 'blue', 'font-size': '14px'}">
    paragraph style is applied using ngStyle
</p>
```

Start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



ngClass

ngClass is used to add or remove CSS classes in HTML elements.

Let us try **ngClass** directive in our **directive-app** application.

Create a class **User** using the below command:

```
ng g class User
```

You could see the following response:

```
CREATE src/app/user.spec.ts (146 bytes)
CREATE src/app/user.ts (22 bytes)
```

Move to **src/app/user.ts** file and add the below code:

```
export class User {
    userId : number;
    userName : string;
}
```

Here, we have created two property **userId** and **userName** in the **User** class.

Open **test.component.ts** file and add the below changes:

```
import { User } from '../user';

export class TestComponent implements OnInit {
    users: User[] = [
        {
          "userId": 1,
          "userName": 'User1'
        },
        {
          "userId": 2,
          "userName": 'User2'
        },
    ];
}
```

Here, we have declared a local variable, **users** and initialise with 2 users object.

Open **test.component.css** file and add below code:

```
.highlight
{
    color: red;
}
```

Open your **test.component.html** file and add the below code:

```
<div class="container">
  <br/>
  <div *ngFor="let user of users" [ngClass]="{
    'highlight':user.userName === 'User1'
  }">
    {{ user.userName }}
  </div>
</div>
```

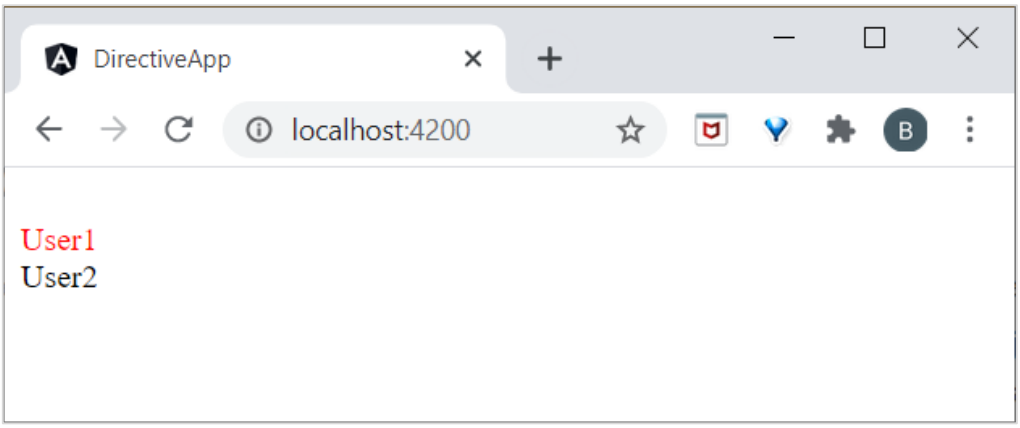
Here,

We have applied, **ngClass** for **User1** so it will highlight the **User1.**

Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



Custom directives

Angular provides option to extend the angular directive with user defined directives and it is called **Custom directives**. Let us learn how to create custom directive in this chapter.

Let us try to create custom directive in our **directive-app** application.

Angular CLI provides a below command to create custom directive.

```
ng generate directive customstyle
```

After executing this command, you could see the below response:

```
CREATE src/app/customstyle.directive.spec.ts (244 bytes)
CREATE src/app/customstyle.directive.ts (151 bytes)
UPDATE src/app/app.module.ts (1115 bytes)
```

Open **app.module.ts**. The directive will be configured in the **AppModule** through **declarations** meta data.

```
import { CustomstyleDirective } from './customstyle.directive';

@NgModule({
  declarations: [
    AppComponent,
    TestComponent,
    CustomstyleDirective
  ]
})
```

Open **customstyle.directive.ts** file and add the below code:

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appCustomstyle]'
})
export class CustomstyleDirective {
```

```
    constructor(el: ElementRef) {
       el.nativeElement.style.fontSize = '24px';
    }
}
```

Here, **constructor** method gets the element using **CustomStyleDirective** as **el.** Then, it accesses el's style and set its font size as **24px** using CSS property.

Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:

ng-template

ng-template is used to create dynamic and reusable templates. It is a virtual element. If you compile your code with **ng-template** then is converted as comment in DOM.

For example,

Let's add a below code in **test.component.html** page.

```
<h3>ng-template</h3>

<ng-template>ng-template tag is a virtual element</ng-template>
```

If you run the application, then it will print only **h3** element. Check your page source, template is displayed in comment section because it is a virtual element so it does not render anything. We need to use **ng-template** along with Angular directives.

Normally, directive emits the HTML tag it is associated. Sometimes, we don't want the tag but only the content. For example, in the below example, *li* will be emitted.

```
<li *ngFor="let item in list">{{ item }}</li>
```

We can use **ng-template** to safely skip the **li** tag.

ng-template with structural directive

ng-template should always be used inside **ngIf, ngFor** or **ngSwitch** directives to render the result.

Let's assume simple code.

```
<ng-template [ngIf]=true>
    <div><h2>ng-template works!</h2></div>
</ng-template>
```

Here, if **ngIf** condition becomes true, it will print the data inside **div** element. Similarly, you can use **ngFor** and **ngSwitch** directives as well.

NgForOf directive

ngForOf is also a structural directive used to render an item in a collection. Below example is used to show **ngForOf** directive inside **ng-template**.

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

@Component({
    selector: 'app-test',
    template: `
    <div>
    <ng-template ngFor let-item [ngForOf]="Fruits" let-i="index">
    <p>{{i}}</p>
    </ng-template>
    </div>`
  ,
  styleUrls: ['./test.component.css']
})
export class TestComponent implements OnInit {
    Fruits = ["mango","apple","orange","grapes"];
    ngOnInit()
    {

    }
}
```

If you run the application, it will show the index of each elements as shown below:

```
0
1
2
3
```

Component directives

Component directives are based on component. Actually, each component can be used as directive. Component provides **@Input** and **@Output** decorator to send and receive information between parent and child components.

Let us try use component as directive in our **directive-app** application.

Create a new **ChildComponent** using below command:

```
ng generate component child

CREATE src/app/child/child.component.html (20 bytes)
CREATE src/app/child/child.component.spec.ts (621 bytes)
CREATE src/app/child/child.component.ts (265 bytes)
CREATE src/app/child/child.component.css (0 bytes)
UPDATE src/app/app.module.ts (466 bytes)
```

Open **child.component.ts** and add below code:

```
@Input() userName: string;
```

Here, we are setting a input property for **ChildComponent.**

Open **child.component.html** and add below code:

```
<p>child works!</p>
<p>Hi {{ userName }}</p>
```

Here, we are using the value **userName** to welcome the user.

Open **test.component.ts** and add below code:

```
name: string = 'Peter';
```

Open **test.component.html** and add below code:

```
<h1>Test component</h1>

<app-child [userName]="name"><app-child>
```

Here, we are using **AppComponent** inside the **TestComponent** as a directive with input property.

Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:

[](images/directive-app/component_as_directive.PNG"

Working example

Let us add a new component in our **ExpenseManager** application to list the expense entries.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Start the application.

```
ng serve
```

Create a new component, **ExpenseEntryListComponent** using below command:

```
ng generate component ExpenseEntryList
```

Output

The output is as follows:

```
CREATE src/app/expense-entry-list/expense-entry-list.component.html (33 bytes)
CREATE src/app/expense-entry-list/expense-entry-list.component.spec.ts (700
bytes)
CREATE src/app/expense-entry-list/expense-entry-list.component.ts (315 bytes)
CREATE src/app/expense-entry-list/expense-entry-list.component.css (0 bytes)
UPDATE src/app/app.module.ts (548 bytes)
```

Here, the command creates the ExpenseEntryList Component and update the necessary code in **AppModule.**

Import **ExpenseEntry** into **ExpenseEntryListComponent** component **(src/app/expense-entry-list/expense-entry-list.component)**

```
import { ExpenseEntry } from '../expense-entry';
```

Add a method, **getExpenseEntries()** to return list of expense entry (mock items) in **ExpenseEntryListComponent** (**src/app/expense-entry-list/expense-entry-list.component**)

```
getExpenseEntries() : ExpenseEntry[] {
        let mockExpenseEntries : ExpenseEntry[] = [
                { id: 1,
                        item: "Pizza",
                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "Mcdonald",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },

                { id: 1,
                        item: "Pizza",
                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "KFC",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },

                { id: 1,
```

```
                        item: "Pizza",

                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "Mcdonald",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },

                { id: 1,
                        item: "Pizza",
                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "KFC",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },

                { id: 1,
                        item: "Pizza",
                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "KFC",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },
        ];

        return mockExpenseEntries;
}
```

Declare a local variable, **expenseEntries** and load the mock list of expense entries as mentioned below:

```
title: string;
expenseEntries: ExpenseEntry[];
constructor() { }

ngOnInit() {
        this.title = "Expense Entry List";
        this.expenseEntries = this.getExpenseEntries();
}
```

Open the template file (**src/app/expense-entry-list/expense-entry-list.component.html**) and show the mock entries in a table.

```
<!-- Page Content -->
```

```
<div class="container">
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">

            <div class="container" style="padding-left: 0px; padding-right:
0px;">
                <div class="row">
                    <div class="col-sm" style="text-align: left;">
                        {{ title }}
                    </div>
                    <div class="col-sm" style="text-align: right;">
                        <button type="button" class="btn btn-
primary">Edit</button>
                    </div>
                </div>
            </div>
            <div class="container box" style="margin-top: 10px;">
                                <table class="table table-striped">
                                        <thead>
                                        <tr>
                                                <th>Item</th>
                                                <th>Amount</th>
                                                <th>Category</th>
                                                <th>Location</th>
                                                <th>Spent On</th>
                                        </tr>
                                </thead>
                                <tbody>
                                        <tr *ngFor="let entry of
expenseEntries">
                                                <th scope="row">{{ entry.item
}}</th>
                                                <th>{{ entry.amount }}</th>
                                                <td>{{ entry.category }}</td>
                                                <td>{{ entry.location }}</td>
                                                <td>{{ entry.spendOn | date:
'short' }}</td>
                                        </tr>
                                </tbody>
                        </table>
            </div>
        </div>
    </div>
</div>
```

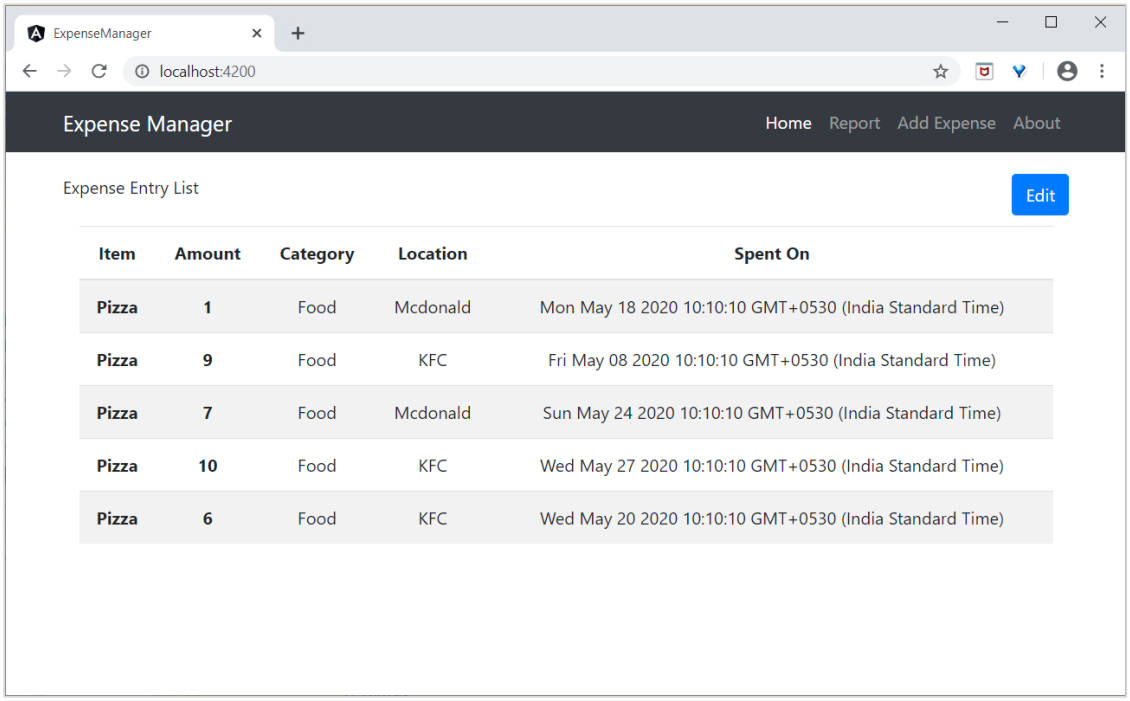
Here,

- Used bootstrap table. **table** and **table-striped** will style the table according to Boostrap style standard.
- Used **ngFor** to loop over the **expenseEntries** and generate table rows.

Open **AppComponent** template, **src/app/app.component.html** and include **ExpenseEntryListComponent** and remove **ExpenseEntryComponent** as shown below:

```
...
<app-expense-entry-list></app-expense-entry-list>
```

Finally, the output of the application is as shown below.



8. Angular 8 — Pipes

Pipes are referred as filters. It helps to transform data and manage data within interpolation, denoted by **{{ | }}.** It accepts data, arrays, integers and strings as inputs which are separated by '|' symbol. This chapter explains about pipes in detail.

Adding parameters

Create a date method in your **test.component.ts** file.

```
export class TestComponent
{
    presentDate = new Date();
}
```

Now, add the below code in your **test.component.html** file.

```
<div>
    Today's date :- {{presentDate}}
</div>
```

Now, run the application, it will show the following output:

```
Today's date :- Mon Jun 15 2020 10:25:05 GMT+0530 (IST)
```

Here,

Date object is converted into easily readable format.

Add Date pipe

Let's add date pipe in the above html file.

```
<div>
    Today's date :- {{presentDate | date }}
</div>
```

You could see the below output:

```
Today's date :- Jun 15, 2020
```

Parameters in Date

We can add parameter in pipe using **:** character. We can show short, full or formatted dates using this parameter. Add the below code in **test.component.html** file.

```
<div>
```

```
    short date :- {{presentDate | date:'shortDate' }} <br/>

    Full date :- {{presentDate | date:'fullDate' }} <br/>
    Formatted date:- {{presentDate | date:'M/dd/yyyy'}} <br/>
    Hours and minutes:- {{presentDate | date:'h:mm'}}
</div>
```

You could see the below response on your screen:

```
short date :- 6/15/20
Full date :- Monday, June 15, 2020
Formatted date:- 6/15/2020
Hours and minutes:- 12:00
```

Chained pipes

We can combine multiple pipes together. This will be useful when a scenario associates with more than one pipe that has to be applied for data transformation.

In the above example, if you want to show the date with uppercase letters, then we can apply both **Date** and **Uppercase** pipes together.

Add the code in **test.component.html** file.

```
<div>
    Date with uppercase :- {{presentDate | date:'fullDate' | uppercase}} <br/>
    Date with lowercase :- {{presentDate | date:'medium' | lowercase}} <br/>
</div>
```

You could see the below response on your screen:

```
Date with uppercase :- MONDAY, JUNE 15, 2020
Date with lowercase :- jun 15, 2020, 12:00:00 am
```

Here,

Date, Uppercase and Lowercase are pre-defined pipes. Let's understand other types of built-in pipes in next section.

Built-in Pipes

Angular 8 supports the following built-in pipes. We will discuss one by one in brief.

AsyncPipe

If data comes in the form of observables, then **Async pipe** subscribes to an observable and returns the transmitted values.

```
import { Observable, Observer } from 'rxjs';


export class TestComponent implements OnInit {

  timeChange = new Observable<string>((observer: Observer<string>) => {

    setInterval(() => observer.next(new Date().toString()), 1000);
  });

}
```

Here,

The **Async** pipe performs subscription for time changing in every one seconds and returns the result whenever gets passed to it. Main advantage is that, we don't need to call subscribe on our **timeChange** and don't worry about unsubscribe, if the component is removed.

Add the below code inside your **test.component.html.**

```
<div>
    Seconds changing in Time: {{ timeChange | async }}
</div>
```

Now, run the application, you could see the seconds changing on your screen.

CurrencyPipe

It is used to convert the given number into various countries currency format. Consider the below code in **test.component.ts** file.

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

@Component({
  selector: 'app-test',
  template: `
  <div style="text-align:center">
    <h3> Currency Pipe</h3>
    <p>{{ price | currency:'EUR':true}}</p>
    <p>{{ price | currency:'INR' }}</p>
  </div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent implements OnInit {

  price : number = 20000;

  ngOnInit() {
```

```
  }

}
```

You could see the following output on your screen:

```
Currency Pipe




€20,000.00

₹20,000.00
```

SlicePipe

Slice pipe is used to return a slice of an array. It takes index as an argument. If you assign only start index, means it will print till the end of values. If you want to print specific range of values, then we can assign start and end index.

We can also use negative index to access elements. Simple example is shown below:

test.component.ts

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

@Component({
  selector: 'app-test',
  template: `
  <div>
   <h3>Start index:- {{Fruits | slice:2}}</h3>
    <h4>Start and end index:- {{Fruits | slice:1:4}}</h4>
    <h5>Negative index:- {{Fruits | slice:-2}}</h5>
    <h6>Negative start and end index:- {{Fruits | slice:-4:-2}}</h6>
  </div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent implements OnInit {

  Fruits = ["Apple","Orange","Grapes","Mango","Kiwi","Pomegranate"];

  ngOnInit() {
  }

}
```

Now run your application and you could see the below output on your screen:

```
Start index:- Grapes,Mango,Kiwi,Pomegranate
```

```
Start and end index:- Orange,Grapes,Mango

Negative index:- Kiwi,Pomegranate

Negative start and end index:- Grapes,Mango
```

Here,

- **{{Fruits | slice:2}}** means it starts from second index value Grapes to till the end of value.
- **{{Fruits | slice:1:4}}** means starts from 1 to end-1 so the result is one to third index values.
- **{{Fruits | slice:-2}}** means starts from -2 to till end because no end value is specified. Hence the result is Kiwi, Pomegranate.
- **{{Fruits | slice:-4:-2}}** means starts from negative index -4 is Grapes to end-1 which is -3 so the result of index[-4,-3] is Grapes, Mango.

DecimalPipe

It is used to format decimal values. It is also considered as CommonModule. Let's understand a simple code in **test.component.ts** file,

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

@Component({
  selector: 'app-test',
  template: `
  <div style="text-align:center">
        <h3>Decimal Pipe</h3>
        <p> {{decimalNum1 | number}} </p>
        <p> {{decimalNum2 | number}} </p>
    </div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent implements OnInit {

  decimalNum1: number = 8.7589623;
  decimalNum2: number = 5.43;
  ngOnInit() {
  }

}
```

You could see the below output on your screen:

```
Decimal Pipe

8.759

5.43
```

Formatting values

We can apply string format inside number pattern. It is based on the below format:

```
number:"{minimumIntegerDigits}.{minimumFractionDigits} -
{maximumFractionDigits}"
```

Let's apply the above format in our code,

```
@Component({
  template: `
  <div style="text-align:center">
        <p> Apply formatting:- {{decimalNum1 | number:'3.1'}} </p>
        <p> Apply formatting:- {{decimalNum1 | number:'2.1-4'}} </p>
    </div>
  `,
})
```

Here,

{{decimalNum1 | number:'3.1'}} means three decimal place and minimum of one fraction but no constraint about maximum fraction limit. It returns the following output:

```
Apply formatting:- 008.759
```

{{decimalNum1 | number:'2.1-4'}} means two decimal places and minimum one and maximum of four fractions allowed so it returns the below output:

```
Apply formatting:- 08.759
```

PercentPipe

It is used to format number as percent. Formatting strings are same as **DecimalPipe** concept. Simple example is shown below:

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

@Component({
  selector: 'app-test',
  template: `
  <div style="text-align:center">
        <h3>Decimal Pipe</h3>
        <p> {{decimalNum1 | percent:'2.2'}} </p>

    </div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent  {
  decimalNum1: number = 0.8178;

}
```

You could see the below output on your screen:

```
Decimal Pipe

81.78%
```

JsonPipe

It is used to transform a JavaScript object into a JSON string. Add the below code in **test.component.ts** file as follows:

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

@Component({
  selector: 'app-test',
  template: `
  <div style="text-align:center">
  <p ngNonBindable>{{ jsonData }}</p> (1)
  <p>{{ jsonData }}</p>
  <p ngNonBindable>{{ jsonData | json }}</p>
  <p>{{ jsonData | json }}</p>
</div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent {
  jsonData = { id: 'one', name: { username: 'user1' }}
}
```

Now, run the application, you could see the below output on your screen:

```
{{ jsonData }}

(1)
[object Object]

{{ jsonData | json }}

{ "id": "one", "name": { "username": "user1" } }
```

Creating custom pipe

As we have seen already, there is a number of pre-defined Pipes available in Angular 8 but sometimes, we may want to transform values in custom formats. This section explains about creating custom Pipes.

Create a custom Pipe using the below command:

```
ng g pipe digitcount
```

After executing the above command, you could see the response:

```
CREATE src/app/digitcount.pipe.spec.ts (203 bytes)
CREATE src/app/digitcount.pipe.ts (213 bytes)
UPDATE src/app/app.module.ts (744 bytes)
```

Let's create a logic for counting digits in a number using Pipe. Open **digitcount.pipe.ts** file and add the below code:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'digitcount'
})
export class DigitcountPipe implements PipeTransform {

  transform(val : number) : number {
    return val.toString().length;

  }
}
```

Now, we have added logic for count number of digits in a number. Let's add the final code in **test.component.ts** file as follows:

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

@Component({
  selector: 'app-test',
  template: `
  <div>
  <p> DigitCount Pipe </p>
  <h1>{{ digits | digitcount }}</h1>
</div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent implements OnInit {
  digits : number = 100;
  ngOnInit() {
  }
}
```

Now, run the application, you could see the below response:

```
DigitCount Pipe

3
```

Working example

Let us use the pipe in the our **ExpenseManager** application.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Start the application.

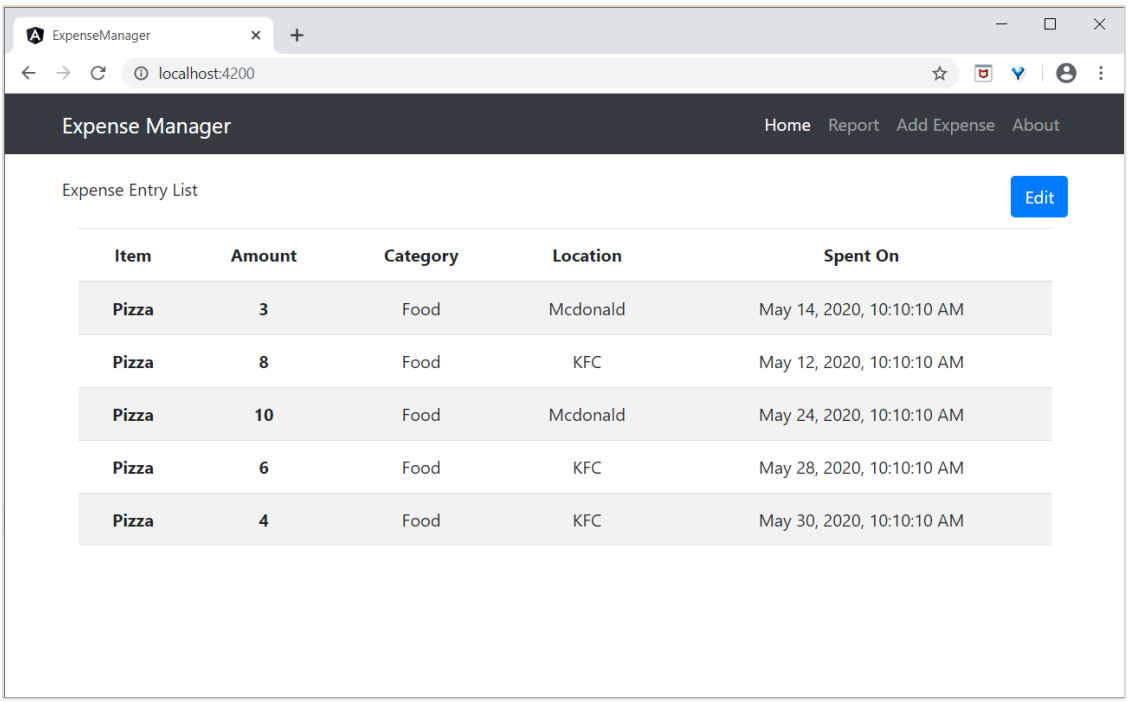
```
ng serve
```

Open **ExpenseEntryListComponent's** template, **src/app/expense-entry-list/expense-entry-list.component.html** and include pipe in **entry.spendOn** as mentioned below:

```
<td>{{ entry.spendOn | date: 'short' }}</td>
```

Here, we have used the date pipe to show the spend on date in the short format.

Finally, the output of the application is as shown below:



9. Angular 8 — Reactive Programming

Reactive programming is a programming paradigm dealing with data streams and the propagation of changes. Data streams may be static or dynamic. An example of static data stream is an array or collection of data. It will have an initial quantity and it will not change. An example for dynamic data stream is event emitters. Event emitters emit the data whenever the event happens. Initially, there may be no events but as the time moves on, events happens and it will gets emitted.

Reactive programming enables the data stream to be emitted from one source called **Observable** and the emitted data stream to be caught by other sources called **Observer** through a process called subscription. This Observable / Observer pattern or simple **Observer** pattern greatly simplifies complex change detection and necessary updating in the context of the programming.

JavaScript does not have the built-in support for Reactive Programming. **RxJs** is a JavaScript Library which enables reactive programming in JavaScript. Angular uses **Rxjs** library extensively to do below mentioned advanced concepts:

- Data transfer between components.
- HTTP client.
- Router.
- Reactive forms.

Let us learn reactive programming using **RxJs** library in this chapter.

Observable

As learn earlier, **Observable** are data sources and they may be static or dynamic. **Rxjs** provides lot of method to create **Observable** from common JavaScript Objects. Let us see some of the common methods.

of - Emit any number of values in a sequence and finally emit a complete notification.

```
const numbers$ = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Here,

- **numbers$** is an **Observable** object, which when subscribed will emit 1 to 10 in a sequence.
- **Dollar sign ($)** at the end of the variable is to identify that the variable is Observable.

range - Emit a range of number in sequence.

```
const numbers$ = range(1,10)
```

from - Emit array, promise or iterable.

```
const numbers$ = from([1,2,3,4,5,6,7,8,9,10]);
```

ajax - Fetch a url through AJAX and then emit the response.

```
const api$ = ajax({
        url: 'https://httpbin.org/delay/1',
        method: 'POST',
        headers: {
        'Content-Type': 'application/text'
        },
        body: "Hello"
});
```

Here,

https://httpbin.org is a free REST API service which will return the supplied body content in the JSON format as specified below:

```
{
  "args": {},
  "data": "Hello",
  "files": {},
  "form": {},
  "headers": {
    "Accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/
*;q=0.8,application/signed-exchange;v=b3;q=0.9",
    "Accept-Encoding": "gzip, deflate, br",
    "Accept-Language": "en-US,en;q=0.9",
    "Host": "httpbin.org",
    "Sec-Fetch-Dest": "document",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-Site": "none",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/83.0.4103.106 Safari/537.36",
    "X-Amzn-Trace-Id": "Root=1-5eeef468-015d8f0c228367109234953c"
  },
  "origin": "ip address",
  "url": "https://httpbin.org/delay/1"
}
```

fromEvent - Listen to an HTML element's event and then emit the event and its property whenever the listened event fires.

```
const clickEvent$ = fromEvent(document.getElementById('counter'), 'click');
```

Angular internally uses the concept extensively to provide data transfer between components and for reactive forms.

Subscribing process

Subscribing to an **Observable** is quite easy. Every Observable object will have a method, **subscribe** for the subscription process. **Observer** need to implement three callback function to subscribe to the Observable object. They are as follows:

- **next** - Receive and process the value emitted from the *Observable*
- **error** - Error handling callback
- **complete** - Callback function called when all data from *Observable* are emitted.

Once the three callback functions are defined, Observable's subscribe method has to be called as specified below:

```
const numbers$ = from([1,2,3,4,5,6,7,8,9,10]);

// observer
const observer = {
        next: (num: number) => { this.numbers.push(num); this.val1 += num },
                error: (err: any) => console.log(err),
                complete: () => console.log("Observation completed")
};
numbers$.subscribe(observer);
```

Here,

- **next** method get the emitted number and then push it into the local variable, **this.numbers.**
- **next** method also adding the number to local variable, **this.val1.**
- **error** method just writes the error message to console.
- **complete** method also writes the completion message to console.

We can skip **error** and **complete** method and write only the **next** method as shown below:

```
number$.subscribe((num: number) => { this.numbers.push(num); this.val1 += num;
});
```

Operations

Rxjs library provides some of the operators to process the data stream. Some of the important **operators** are as follows:

filter - Enable to filter the data stream using callback function.

```
const filterFn = filter( (num : number) =>  num > 5 );
const filteredNumbers$ = filterFn(numbers$);
filteredNumbers$.subscribe( (num : number) => { this.filteredNumbers.push(num);
this.val2 += num } );
```

map - Enables to map the data stream using callback function and to change the data stream itself.

```
const mapFn = map( (num : number) => num + num );
const mappedNumbers$ = mappedFn(numbers$);
```

pipe - Enable two or more operators to be combined.

```
const filterFn = filter( (num : number) =>  num > 5 );
const mapFn = map( (num : number) => num + num );
const processedNumbers$ = numbers$.pipe(filterFn, mapFn);
processedNumbers$.subscribe( (num : number) => {
this.processedNumbers.push(num); this.val3 += num } );
```

Let us create a sample application to try out the reaction programming concept learned in this chapter.

Create a new application, **reactive** using below command:

```
ng new reactive
```

Change the directory to our newly created application.

```
cd reactive
```

Run the application.

```
ng serve
```

Change the **AppComponent** component code (**src/app/app.component.ts**) as specified below:

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

import { Observable, of, range, from, fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { filter, map, catchError } from 'rxjs/operators';

@Component({
        selector: 'app-root',
        templateUrl: './app.component.html',
        styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
        title = 'Reactive programming concept';

        numbers : number[] = [];
        val1 : number = 0;

        filteredNumbers : number[] = [];
        val2 : number = 0;

        processedNumbers : number[] = [];
        val3 : number = 0;

        apiMessage : string;
        counter : number = 0;

        ngOnInit() {
                // Observable stream of data Observable<number>
```

```
                // const numbers$ = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
                // const numbers$ = range(1,10);
                const numbers$ = from([1,2,3,4,5,6,7,8,9,10]);

                // observer
                const observer = {
                        next: (num: number) => { this.numbers.push(num);
this.val1 += num },
                                error: (err: any) => console.log(err),
                                complete: () => console.log("Observation
completed")
                };
                numbers$.subscribe(observer);


                const filterFn = filter( (num : number) =>  num > 5 );
                const filteredNumbers = filterFn(numbers$);
                filteredNumbers.subscribe( (num : number) => {
this.filteredNumbers.push(num); this.val2 += num } );

                const mapFn = map( (num : number) => num + num );
                const processedNumbers$ = numbers$.pipe(filterFn, mapFn);
                processedNumbers$.subscribe( (num : number) => {
this.processedNumbers.push(num); this.val3 += num } );

                const api$ = ajax({
                        url: 'https://httpbin.org/delay/1',
                        method: 'POST',
                        headers: {
                        'Content-Type': 'application/text'
                        },
                        body: "Hello"
                });

                api$.subscribe(res => this.apiMessage = res.response.data );

                const clickEvent$ =
fromEvent(document.getElementById('counter'), 'click');
                clickEvent$.subscribe( () => this.counter++ );
        }
}
```

Here,

- Used of, range, from, ajax and fromEvent methods to created Observable.
- Used filter, map and pipe operator methods to process the data stream.
- Callback functions catch the emitted data, process it and then store it in component's local variables.

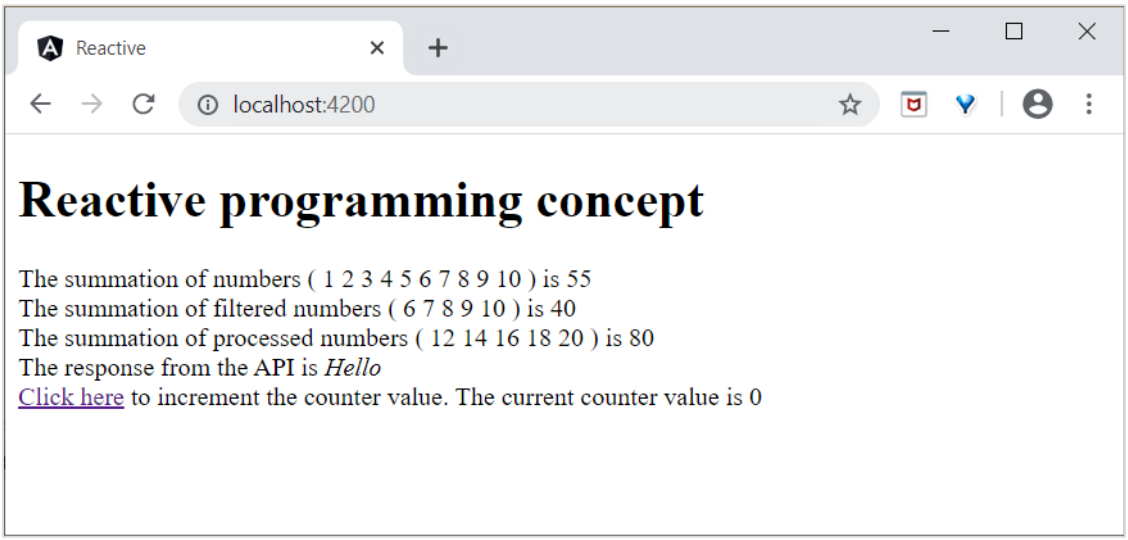
Change the **AppComponent** template (**src/app/app.component.html**) as specified below:

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

<div>
        The summation of numbers ( <span *ngFor="let num of numbers"> {{ num }}
</span> ) is {{ val1 }}
</div>

<div>
        The summation of filtered numbers ( <span *ngFor="let num of
filteredNumbers"> {{ num }} </span> ) is {{ val2 }}
</div>

<div>
        The summation of processed numbers ( <span *ngFor="let num of
processedNumbers"> {{ num }} </span> ) is {{ val3 }}
</div>

<div>
        The response from the API is <em>{{ apiMessage }}</em>
</div>

<div>
        <a id="counter" href="#">Click here</a> to increment the counter value.
The current counter value is {{ counter }}
<div>
```

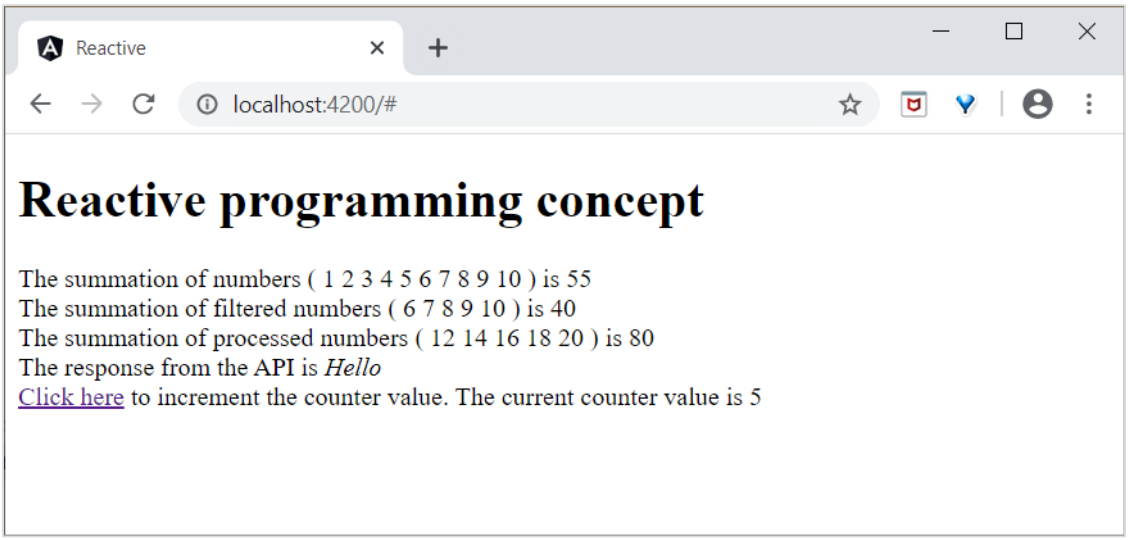
Here,

Shown all the local variable processed by **Observer** callback functions.

Open the browser, http://localhost:4200.



Click the **Click here** link for five times. For each event, the event will be emitted and forward to the **Observer**. Observer callback function will be called. The callback function increment the counter for every click and the final result will be as shown below:



10. Angular 8 — Services and Dependency Injection

As learned earlier, **Services** provides specific functionality in an Angular application. In a given Angular application, there may be one or more services can be used. Similarly, an Angular component may depend on one or more services.

Also, Angular services may depend on another services to work properly. Dependency resolution is one of the complex and time consuming activity in developing any application. To reduce the complexity, Angular provides **Dependency Injection** pattern as one of the core concept.

Let us learn, how to use Dependency Injection in Angular application in this chapter.

Create Angular service

An Angular service is plain Typescript class having one or more methods (functionality) along with **@Injectable** decorator. It enables the normal Typescript class to be used as service in Angular application.

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

@Injectable()
export class DebugService {
  constructor() { }
}
```

Here, **@Injectable** decorator converts a plain Typescript class into Angular service.

Register Angular service

To use **Dependency Injection**, every service needs to be registered into the system. Angular provides multiple option to register a service. They are as follows:

- ModuleInjector @ root level
- ModuleInjector @ platform level
- ElementInjector using providers meta data
- ElementInjector using viewProviders meta data
- NullInjector

ModuleInjector @ root

ModuleInjector enforces the service to used only inside a specific module. **ProvidedIn** meta data available in **@Injectable** has to be used to specify the module in which the service can be used.

The value should refer to the one of the registered Angular Module (decorated with **@NgModule**). **root** is a special option which refers the root module of the application. The sample code is as follows:

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

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

ModuleInjector @ platform

Platform Injector is one level higher than **ModuleInject** and it is only in advanced and rare situation. Every Angular application starts by executing **PreformBrowserDynamic().bootstrap** method (see **main.js**), which is responsible for bootstrapping root module of Angular application.

PreformBrowserDynamic() method creates an injector configured by **PlatformModule**. We can configure platform level services using **platformBrowser()** method provided by **PlatformModule**.

NullInjector

NullInjector is one level higher than platform level **ModuleInjector** and is in the top level of the hierarchy. We could not able to register any service in the **NullInjector**. It resolves when the required service is not found anywhere in the hierarchy and simply throws an error.

ElementInjector using providers

ElementInjector enforces the service to be used only inside some particular components. providers and **ViewProviders** meta data available in **@Component** decorator is used to specify the list of services to be visible for the particular component. The sample code to use providers is as follows:

ExpenseEntryListComponent

```
// import statement
import { DebugService } from '../debug.service';

// component decorator
@Component({
        selector: 'app-expense-entry-list',
        templateUrl: './expense-entry-list.component.html',
        styleUrls: ['./expense-entry-list.component.css'],
        providers: [DebugService]
})
```

Here, **DebugService** will be available only inside the **ExpenseEntryListComponent** and its view. To make DebugService in other component, simply use **providers** decorator in necessary component.

ElementInjector using viewProviders

viewProviders is similar to **provider** except it does not allow the service to be used inside the component's content created using **ng-content** directive.

ExpenseEntryListComponent

```
// import statement
import { DebugService } from '../debug.service';

// component decorator
@Component({
        selector: 'app-expense-entry-list',
        templateUrl: './expense-entry-list.component.html',
        styleUrls: ['./expense-entry-list.component.css'],
        viewProviders: [DebugService]
})
```

Parent component can use a child component either through its view or content. Example of a parent component with child and content view is mentioned below:

Parent component view / template

```
<div>
    child template in view
    <child></child>
</div>
<ng-content></ng-content>
```

child component view / template

```
<div>
    child template in view
</div>
```

Parent component usage in a template (another component)

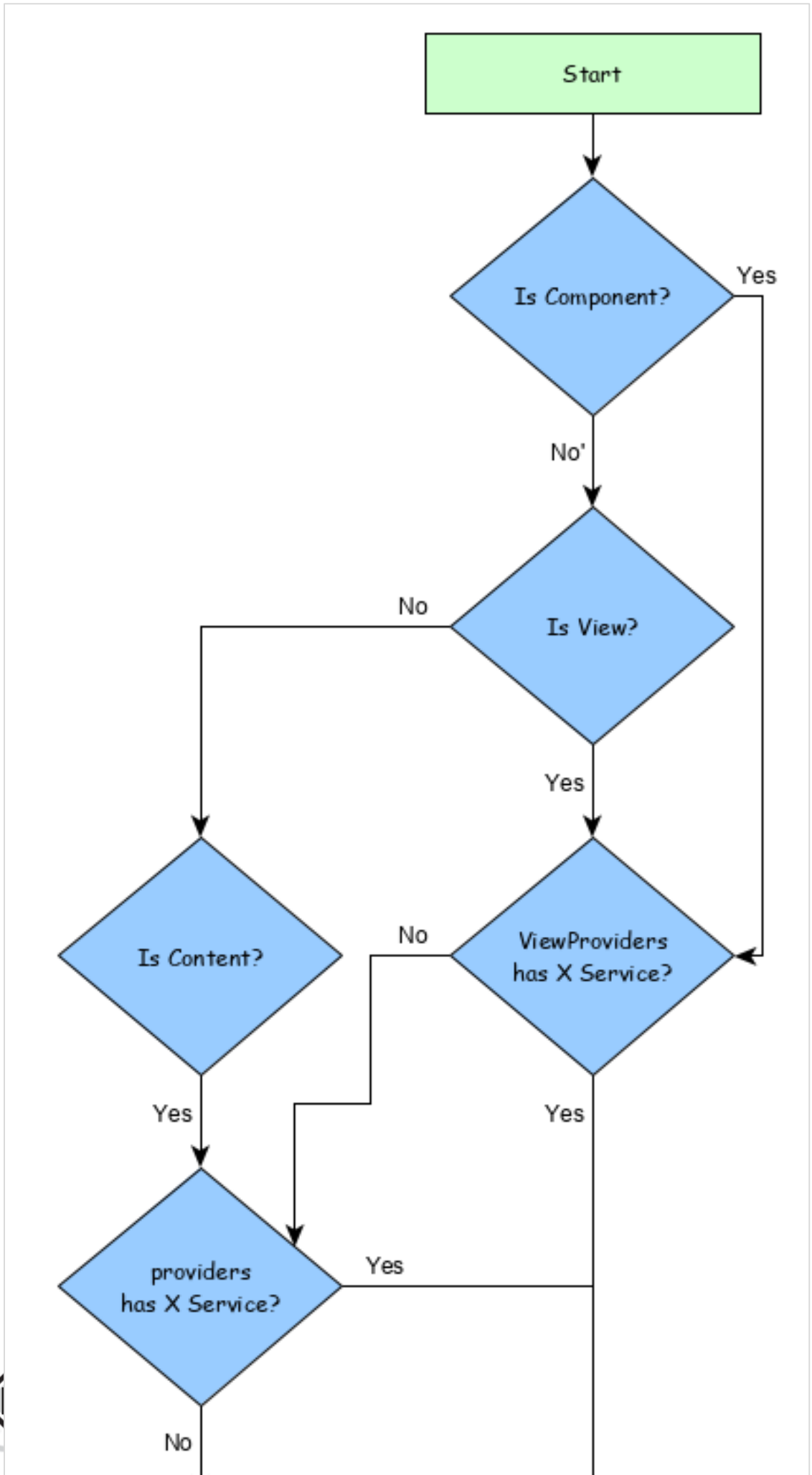
```
<parent><!-- child template in content --><child></child></parent>
```

Here,

- **child** component is used in two place. One inside the parent's view. Another inside parent content.
- Services will be available in child component, which is placed inside parent's view.
- Services will not be available in child component, which is placed inside parent's content.

Resolve Angular service

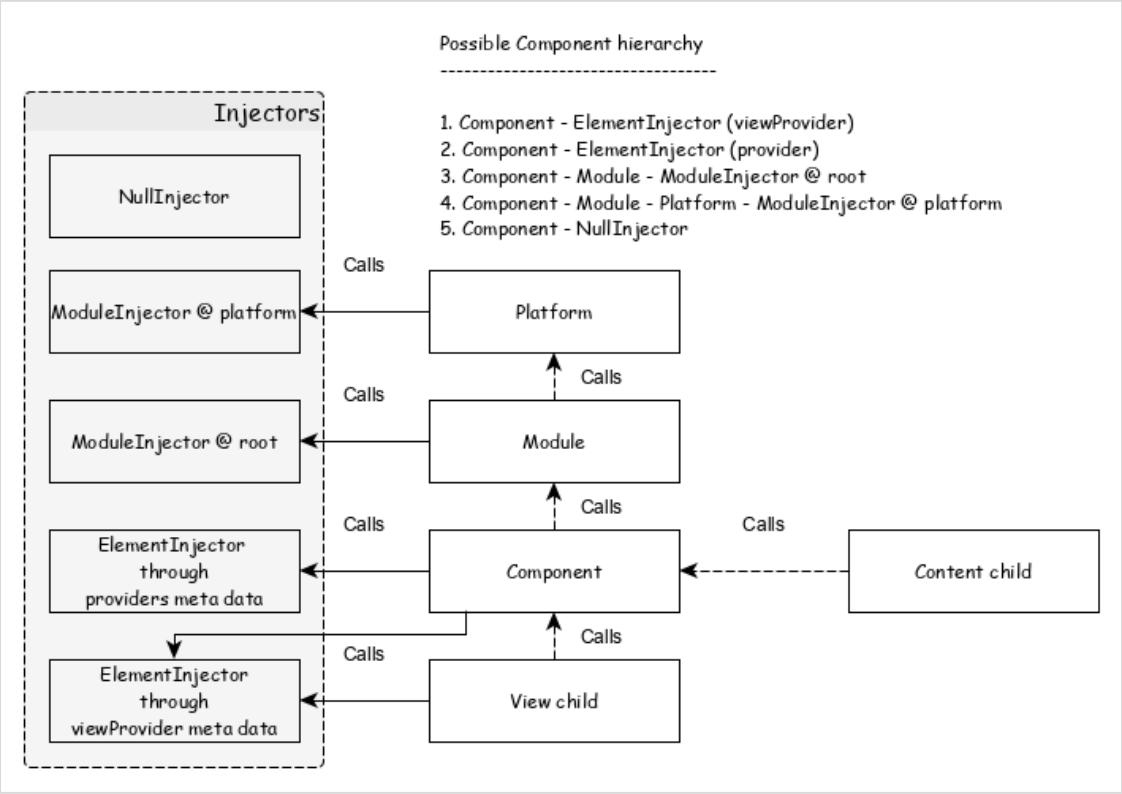
Let us see how a component can resolve a service using the below flow diagram.


Start
Is Component?
Yes
No'
Is View?
No
Yes
Is Content?
ViewProviders
has X Service?
No
Yes
Yes
providers
has X Service?
Yes
No

Here,

- First, component tries to find the service registered using **viewProviders** meta data.
- If not found, component tries to find the service registered using **providers** meta data.
- If not found, Component tries to find the service registered using **ModuleInjector**
- If not found, component tries to find the service registered using **PlatformInjector**
- If not found, component tries to find the service registered using **NullInjector**, which always throws error.

The hierarchy of the Injector along with work flow of the resolving the service is as follows:



Resolution Modifier

As we learn in the previous chapter, the resolution of the service starts from component and stops either when a service is found or **NUllInjector** is reached. This is the default resolution and it can be changed using **Resolution Modifier**. They are as follows:

Self()

Self() start and stops the search for the service in its current **ElementInjector** itself.

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

constructor(@Self() public debugService: DebugService) {}
```

SkipSelf()

SkipSelf() is just opposite to Self(). It skips the current **ElementInjector** and starts the search for service from its parent ElementInjector.

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

constructor(@SkipSelf() public debugService: DebugService) {}
```

Host()

Host() stop the search for the service in its host **ElementInjector**. Even if service available up in the higher level, it stops at host.

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

constructor(@Host() public debugService: DebugService) {}
```

Optional()

@Optional does not throws the error when the search for the service fails.

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

constructor(@Optional() private debugService?: DebugService) {
  if (this.debugService) {
    this.debugService.info("Debugger initialized");
  }
}
```

Dependency Injector Providers

Dependency Injector providers serves two purpose. First, it helps in setting a token for the service to be registered. The token will be used to refer and call the service. Second, it helps in creating the service from the given configuration.

As learned earlier, the simplest provider is as follows:

```
providers: [ DebugService ]
```

Here, **DebugService** is both token as well as the class, with which the service object has to be created. The actual form of the provider is as follows:

```
providers: [ { provides: DebugService, useClass: DebugService }]
```

Here, **provides** is the token and **useClass** is the class reference to create the service object.

Angular provides some more providers and they are as follows:

Aliased class providers

The purpose of the providers is to reuse the existing service.

```
providers: [ DebugService,
                { provides: AnotherDebugService, userClass: DebugService }]
```

Here, only one instance of **DebugService** service will be created.

Value providers

The purpose of the Value providers is to supply the value itself instead of asking the DI to create an instance of the service object. It may use existing object as well. The only restriction is that the object should be in the shape of referenced service.

```
export class MyCustomService {
    name = "My Custom Service"
}

[{ provide: MyService, useValue: { name: 'instance of MyCustomService' }]
```

Here, DI provider just return the instance set in **useValue** option instead of creating a new service object.

Non-class dependency providers

It enables string, function or object to be used in Angular DI.

Let us see a simple example.

```
// Create the injectable token
import { InjectionToken } from '@angular/core';
export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');

// Create value
export const MY_CONFIG: AppConfig = {
    title: 'Dependency Injection'
};

// congfigure providers
providers: [{ provide: APP_CONFIG, useValue: HERO_DI_CONFIG }]

// inject the service
constructor(@Inject(APP_CONFIG) config: AppConfig) {
```

Factory providers

Factory Providers enables complex service creation. It delegates the creation of the object to an external function. Factory providers has option to set the dependency for factory object as well.

```
{ provide: MyService, useFactory: myServiceFactory, deps: [DebugService] };
```

Here, **myServiceFactory** returns the instance of **MyService**.

Angular Service usage

Now, we know how to create and register Angular Service. Let us see how to use the Angular Service inside a component. Using an Angular service is as simple as setting the type of parameters of the constructor as the token of the service providers.

```
export class ExpenseEntryListComponent implements OnInit {
```

```
  title = 'Expense List';

  constructor(private debugService : DebugService) {}

  ngOnInit() {
      this.debugService.info("Angular Application starts");
  }
}
```

Here,

- **ExpenseEntryListComponent** constructor set a parameter of type DebugService.
- **Angular Dependency Injector** (DI) will try to find any service registered in the application with type DebugService. If found, it will set an instance of DebugService to ExpenseEntryListComponent component. If not found, it will throw an error.

Add a debug service

Let us add a simple **Debug** service, which will help us to print the debugging information during application development.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Start the application.

```
ng serve
```

Run the below command to generate an Angular service, **DebugService.**

```
ng g service debug
```

This will create two Typescript files (debug service & its test) as specified below:

```
CREATE src/app/debug.service.spec.ts (328 bytes)
CREATE src/app/debug.service.ts (134 bytes)
```

Let us analyse the content of the **DebugService** service.

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

@Injectable({
  providedIn: 'root'
})
export class DebugService {

  constructor() { }
}
```

Here,

- **@Injectable** decorator is attached to DebugService class, which enables the DebugService to be used in Angular component of the application.
- **providerIn** option and its value, root enables the DebugService to be used in all component of the application.

Let us add a method, **Info,** which will print the message into the browser console.

```
  info(message : String) : void {
          console.log(message);
  }
```

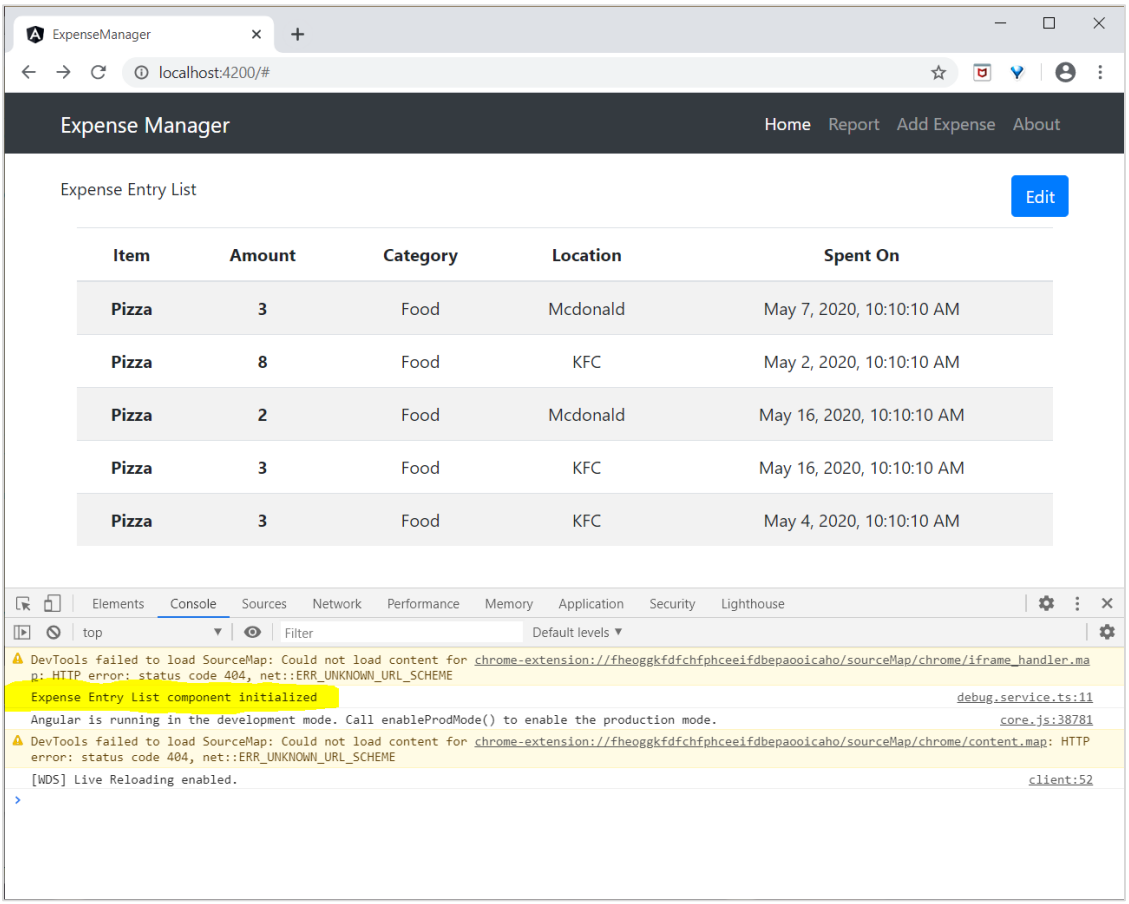
Let us initialise the service in the **ExpenseEntryListComponent** and use it to print message.

```
import { Component, OnInit } from '@angular/core';
import { ExpenseEntry } from '../expense-entry';
import { DebugService } from '../debug.service';

@Component({
        selector: 'app-expense-entry-list',
        templateUrl: './expense-entry-list.component.html',
        styleUrls: ['./expense-entry-list.component.css']
})
export class ExpenseEntryListComponent implements OnInit {
        title: string;
        expenseEntries: ExpenseEntry[];
        constructor(private debugService: DebugService) { }

        ngOnInit() {
        this.debugService.info("Expense Entry List component initialized");
                this.title = "Expense Entry List";
                this.expenseEntries = this.getExpenseEntries();
        }

        // other coding
}
```

Here,

- DebugService is initialised using constructor parameters. Setting an argument (debugService) of type DebugService will trigger the dependency injection to create a new DebugService object and set it into the ExpenseEntryListComponent component.
- Calling the info method of DebugService in the ngOnInit method prints the message in the browser console.

The result can be viewed using developer tools and it looks similar as shown below:



Let us extend the application to understand the scope of the service.

Let us a create a **DebugComponent** by using below mentioned command.

```
ng generate component debug

CREATE src/app/debug/debug.component.html (20 bytes)
CREATE src/app/debug/debug.component.spec.ts (621 bytes)
CREATE src/app/debug/debug.component.ts (265 bytes)
CREATE src/app/debug/debug.component.css (0 bytes)
UPDATE src/app/app.module.ts (392 bytes)
```

Let us remove the DebugService in the root module.

```
// src/app/debug.service.ts

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

@Injectable()
export class DebugService {
        constructor() {
        }
```

```
        info(message : String) : void {
        console.log(message);
        }
}
```

Register the DebugService under ExpenseEntryListComponent component.

```
// src/app/expense-entry-list/expense-entry-list.component.ts
@Component({
        selector: 'app-expense-entry-list',
        templateUrl: './expense-entry-list.component.html',
        styleUrls: ['./expense-entry-list.component.css']
        providers: [DebugService]
})
```

Here, we have used providers meta data (**ElementInjector**) to register the service.

Open **DebugComponent** (src/app/debug/debug.component.ts) and import **DebugService** and set an instance in the constructor of the component.

```
import { Component, OnInit } from '@angular/core';
import { DebugService } from '../debug.service';

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

  constructor(private debugService: DebugService) { }

  ngOnInit() {
          this.debugService.info("Debug component gets service from Parent");
  }
}
```

Here, we have not registered **DebugService**. So, DebugService will not be available if used as parent component. When used inside a parent component, the service may available from parent, if the parent has access to the service.

Open **ExpenseEntryListComponent** template (src/app/expense-entry-list/expense-entry-list.component.html) and include a content section as shown below:

```
// existing content
<app-debug></app-debug>
<ng-content></ng-content>
```

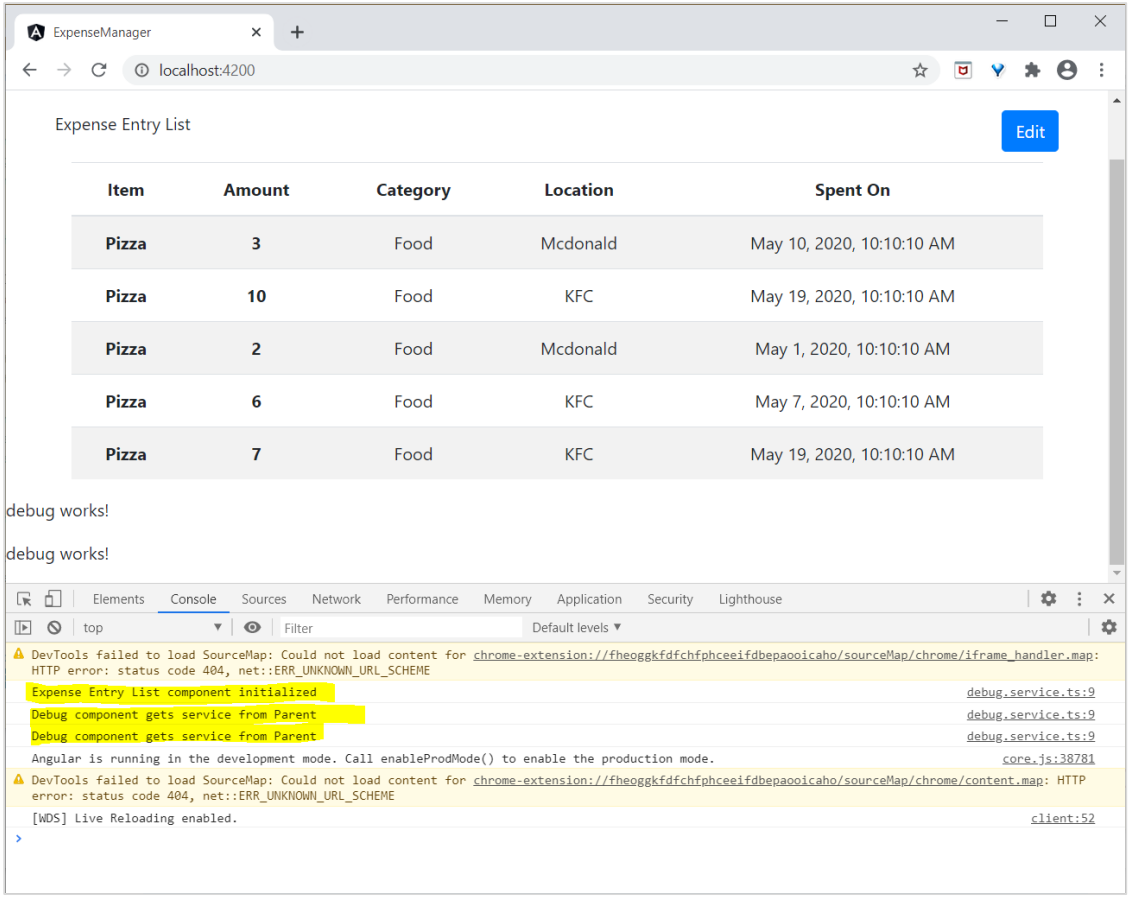
Here, we have included a content section and **DebugComponent** section.

Let us include the debug component as a content inside the **ExpenseEntryListComponent** component in the **AppComponent** template. Open AppComponent template and change **app-expense-entry-list** as below:

```
// navigation code

<app-expense-entry-list>
        <app-debug></app-debug>
</app-expense-entry-list>
```

Here, we have included the **DebugComponent** as content.

Let us check the application and it will show **DebugService** template at the end of the page as shown below:

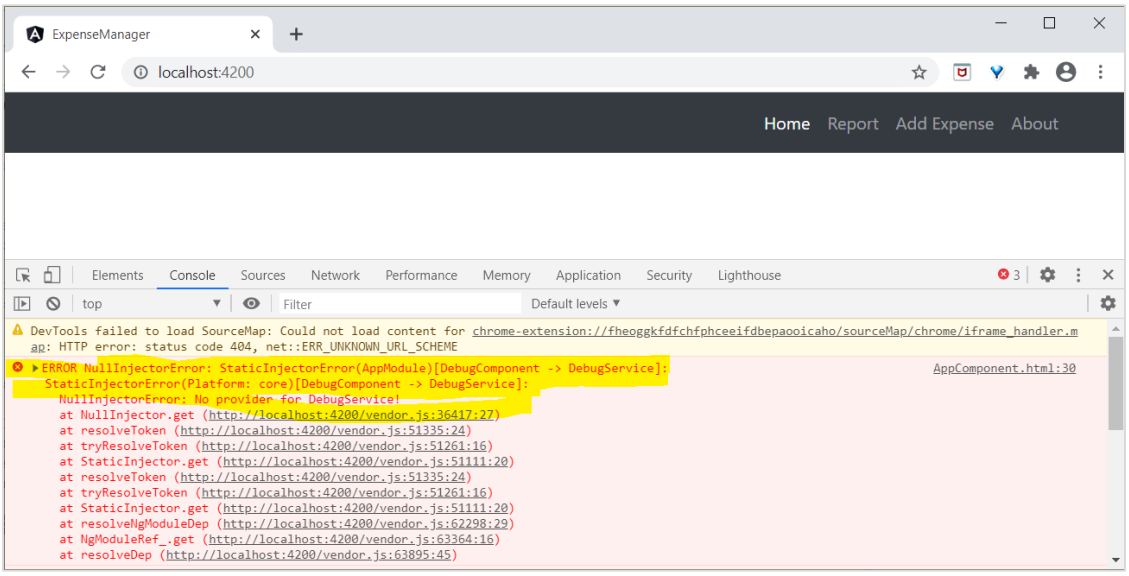


Also, we could able to see two debug information from debug component in the console. This indicate that the debug component gets the service from its parent component.

Let us change how the service is injected in the **ExpenseEntryListComponent** and how it affects the scope of the service. Change providers injector to **viewProviders** injection. viewProviders does not inject the service into the content child and so, it should fail.

```
viewProviders: [DebugService]
```

Check the application and you will see that the one of the debug component (used as content child) throws error as shown below:



Let us remove the debug component in the templates and restore the application.

Open **ExpenseEntryListComponent** template (src/app/expense-entry-list/expense-entry-list.component.html) and remove below content:

```
<app-debug></app-debug>
<ng-content></ng-content>
```

Open **AppComponent** template and change **app-expense-entry-list** as below:

```
// navigation code
<app-expense-entry-list> </app-expense-entry-list>
```

Change the **viewProviders** setting to **providers** in **ExpenseEntryListComponent.**

```
providers: [DebugService]
```

Rerun the application and check the result.

11. Angular 8 — Http Client Programming

Http client programming is a must needed feature in every modern web application. Nowadays, lot of application exposes their functionality through REST API (functionality over HTTP protocol). With this in mind, Angular Team provides extensive support to access HTTP server. Angular provides a separate module, **HttpClientModule** and a service, **HttpClient** to do HTTP programming.

Let us learn how to how to use **HttpClient** service in this chapter. Developer should have a basic knowledge in Http programming to understand this chapter.

Expense REST API

The prerequisite to do Http programming is the basic knowledge of Http protocol and REST API technique. Http programming involves two part, server and client. Angular provides support to create client side application. **Express** a popular web framework provides support to create server side application.

Let us create an **Expense Rest API** using *express* framework and then access it from our **ExpenseManager** application using Angular HttpClient service.

Open a command prompt and create a new folder, **express-rest-api**.

```
cd /go/to/workspace
mkdir express-rest-api
cd expense-rest-api
```

Initialise a new node application using below command:

```
npm init
```

npm init will ask some basic questions like project name (express-rest-api), entry point (server.js), etc., as mentioned below:

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (expense-rest-api)
version: (1.0.0)
description: Rest api for Expense Application
entry point: (index.js) server.js
test command:
```

```
git repository:
keywords:
author:
license: (ISC)
About to write to \path\to\workspace\expense-rest-api\package.json:

{
  "name": "expense-rest-api",
  "version": "1.0.0",
  "description": "Rest api for Expense Application",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes) yes
```

Install **express, sqlite** and **cors** modules using below command:

```
npm install express sqlite3 cors
```

Create a new file **sqlitedb.js** and place below code:

```
var sqlite3 = require('sqlite3').verbose()
const DBSOURCE = "expensedb.sqlite"

let db = new sqlite3.Database(DBSOURCE, (err) => {
    if (err) {
        console.error(err.message)
        throw err
    }else{
        console.log('Connected to the SQLite database.')
        db.run(`CREATE TABLE expense (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            item text,
            amount real,
            category text,
            location text,
            spendOn text,
            createdOn text
            )`,
                (err) => {
                if (err) {
                        console.log(err);
                }else{
                        var insert = 'INSERT INTO expense (item, amount,
category, location, spendOn, createdOn) VALUES (?,?,?,?,?,?)'
```

```
                        db.run(insert, ['Pizza', 10, 'Food', 'KFC', '2020-05-26
10:10', '2020-05-26 10:10'])
                        db.run(insert, ['Pizza', 9, 'Food', 'Mcdonald', '2020-
05-28 11:10', '2020-05-28 11:10'])
                        db.run(insert, ['Pizza', 12, 'Food', 'Mcdonald', '2020-
05-29 09:22', '2020-05-29 09:22'])
                        db.run(insert, ['Pizza', 15, 'Food', 'KFC', '2020-06-06
16:18', '2020-06-06 16:18'])
                        db.run(insert, ['Pizza', 14, 'Food', 'Mcdonald', '2020-
06-01 18:14', '2020-05-01 18:14'])
                }
                });
    }
});

module.exports = db
```

Here, we are creating a new sqlite database and load some sample data.

Open **server.js** and place below code:

```
var express = require("express")
var cors = require('cors')
var db = require("./sqlitedb.js")

var app = express()
app.use(cors());

var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

var HTTP_PORT = 8000
app.listen(HTTP_PORT, () => {
    console.log("Server running on port %PORT%".replace("%PORT%",HTTP_PORT))
});

app.get("/", (req, res, next) => {
    res.json({"message":"Ok"})
});

app.get("/api/expense", (req, res, next) => {
    var sql = "select * from expense"
    var params = []
    db.all(sql, params, (err, rows) => {
        if (err) {
          res.status(400).json({"error":err.message});
          return;
        }
        res.json(rows)
      });
```

```
});

app.get("/api/expense/:id", (req, res, next) => {
    var sql = "select * from expense where id = ?"
    var params = [req.params.id]
    db.get(sql, params, (err, row) => {
        if (err) {
          res.status(400).json({"error":err.message});
          return;
        }
        res.json(row)
      });
});

app.post("/api/expense/", (req, res, next) => {
    var errors=[]
    if (!req.body.item){
        errors.push("No item specified");
    }
    var data = {
        item : req.body.item,
        amount: req.body.amount,
        category: req.body.category,
        location : req.body.location,
        spendOn: req.body.spendOn,
        createdOn: req.body.createdOn,
    }
    var sql = 'INSERT INTO expense (item, amount, category, location, spendOn,
createdOn) VALUES (?,?,?,?,?,?)'
    var params =[data.item, data.amount, data.category, data.location,
data.spendOn, data.createdOn]
    db.run(sql, params, function (err, result) {
        if (err){
            res.status(400).json({"error": err.message})
            return;
        }
        data.id = this.lastID;
        res.json(data);
    });
})

app.put("/api/expense/:id", (req, res, next) => {
    var data = {
        item : req.body.item,
        amount: req.body.amount,
        category: req.body.category,
        location : req.body.location,
        spendOn: req.body.spendOn
    }
    db.run(
        `UPDATE expense SET
           item = ?,
```

```
            amount = ?,
            category = ?,
            location = ?,

            spendOn = ?
            WHERE id = ?`,
                [data.item, data.amount, data.category, data.location,
data.spendOn, req.params.id],
        function (err, result) {
            if (err){
                console.log(err);
                res.status(400).json({"error": res.message})
                return;
            }
            res.json(data)
    });
})

app.delete("/api/expense/:id", (req, res, next) => {
    db.run(
        'DELETE FROM expense WHERE id = ?',
        req.params.id,
        function (err, result) {
            if (err){
                res.status(400).json({"error": res.message})
                return;
            }
            res.json({"message":"deleted", changes: this.changes})
    });
})

app.use(function(req, res){
    res.status(404);
});
```

Here, we create a basic CURD rest api to select, insert, update and delete expense entry.

Run the application using below command:

```
npm run start
```

Open a browser, enter **http://localhost:8000/** and press enter. You will see below response:

```
{
  "message": "Ok"
}
```

It confirms our application is working fine.

Change the url to **http://localhost:8000/api/expense** and you will see all the expense entries in JSON format.

```
[
  {
    "id": 1,

    "item": "Pizza",
    "amount": 10,
    "category": "Food",
    "location": "KFC",
    "spendOn": "2020-05-26 10:10",
    "createdOn": "2020-05-26 10:10"
  },
  {
    "id": 2,
    "item": "Pizza",
    "amount": 14,
    "category": "Food",
    "location": "Mcdonald",
    "spendOn": "2020-06-01 18:14",
    "createdOn": "2020-05-01 18:14"
  },
  {
    "id": 3,
    "item": "Pizza",
    "amount": 15,
    "category": "Food",
    "location": "KFC",
    "spendOn": "2020-06-06 16:18",
    "createdOn": "2020-06-06 16:18"
  },
  {
    "id": 4,
    "item": "Pizza",
    "amount": 9,
    "category": "Food",
    "location": "Mcdonald",
    "spendOn": "2020-05-28 11:10",
    "createdOn": "2020-05-28 11:10"
  },
  {
    "id": 5,
    "item": "Pizza",
    "amount": 12,
    "category": "Food",
    "location": "Mcdonald",
    "spendOn": "2020-05-29 09:22",
    "createdOn": "2020-05-29 09:22"
  }
]
```


Finally, we created a simple CURD REST API for expense entry and we can access the REST API from our Angular application to learn **HttpClient** module.

Configure Http client

Let us learn how to configure **HttpClient** service in this chapter.

HttpClient service is available inside the **HttpClientModule** module, which is available inside the @angular/common/http package.

To register **HttpClientModule** module:

Import the HttpClientModule in **AppComponent**

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

Include **HttpClientModule** in **imports** meta data of **AppComponent.**

```
@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ]
})
export class AppModule {}
```

Create expense service

Let us create a new service **ExpenseEntryService** in our **ExpenseManager** application to interact with **Expense REST API**. ExpenseEntryService will get the latest expense entries, insert new expense entries, modify existing expense entries and delete the unwanted expense entries.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Start the application.

```
ng serve
```

Run the below command to generate an Angular service, **ExpenseService.**

```
ng generate service ExpenseEntry
```

This will create two Typescript files (expense entry service & its test) as specified below:

```
CREATE src/app/expense-entry.service.spec.ts (364 bytes)
CREATE src/app/expense-entry.service.ts (141 bytes)
```

Open **ExpenseEntryService** (src/app/expense-entry.service.ts) and import **ExpenseEntry, throwError** and **catchError** from rxjs library and import **HttpClient, HttpHeaders** and **HttpErrorResponse** from @angular/common/http package.

```
import { Injectable } from '@angular/core';
import { ExpenseEntry } from './expense-entry';
import { throwError } from 'rxjs';



import { catchError } from 'rxjs/operators';
import { HttpClient, HttpHeaders, HttpErrorResponse } from
'@angular/common/http';
```

Inject the **HttpClient** service into our service.

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

Create a variable, **expenseRestUrl** to specify the **Expense Rest API** endpoints.

```
private expenseRestUrl = 'http://localhost:8000/api/expense';
```

Create a variable, **httpOptions** to set the Http Header option. This will be used during the Http Rest API call by Angular **HttpClient** service.

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

The complete code is as follows:

```
import { Injectable } from '@angular/core';
import { ExpenseEntry } from './expense-entry';
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
import { HttpClient, HttpHeaders, HttpErrorResponse } from
'@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class ExpenseEntryService {
        private expenseRestUrl = 'api/expense';
        private httpOptions = {
                headers: new HttpHeaders( { 'Content-Type': 'application/json'
})
        };

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

HTTP GET

HttpClient provides **get()** method to fetch data from a web page. The main argument is the **target web url**. Another optional argument is the **option** object with below format:

```
{
    headers?: HttpHeaders | {[header: string]: string | string[]},
    observe?: 'body' | 'events' | 'response',

    params?: HttpParams|{[param: string]: string | string[]},
    reportProgress?: boolean,
    responseType?: 'arraybuffer'|'blob'|'json'|'text',
    withCredentials?: boolean,
}
```

Here,

- **headers:** HTTP Headers of the request, either as string, array of string or array of HttpHeaders.
- **observe:** Process the response and return the specific content of the response. Possible values are **body, response** and **events**. The default option of **observer** is **body.**
- **params:** HTTP parameters of the request, either as string, array of string or array of **HttpParams.**
- **reportProgress:** Whether to report the progress of the process or not (true or false).
- **responseType:** Refers the format of the response. Possible values are **arraybuffer, blob, json** and **text.**
- **withCredentials:** Whether the request has credentials or not (true or false).

All options are optional.

get() method returns the response of the request as **Observable**. The returned Observable emit the data when the response is received from the server.

The sample code to use **get()** method is as follows:

```
httpClient.get(url, options)
    .subscribe( (data) => console.log(data) );
```

Typed Response

get() method has an option to return observables, which emits typed response as well. The sample code to get typed response (ExpenseEntry) is as follows:

```
httpClient.get<T>(url, options)
    .subscribe( (data: T) => console.log(data) );
```

Handling errors

Error handling is one of the important aspect in the HTTP programming. Encountering error is one of the common scenario in HTTP programming.

Errors in HTTP Programming can be categories into two groups:

- Client side issues can occur due to network failure, misconfiguration, etc., If client side error happens, then the **get()** method throws **ErrorEvent** object.

- Server side issues can occur due to wrong url, server unavailability, server programming errors, etc.,

Let us write a simple error handling for our **ExpenseEntryService** service.

```
private httpErrorHandler (error: HttpErrorResponse) {
        if (error.error instanceof ErrorEvent) {
        console.error("A client side error occurs. The error message is " +
error.message);
        } else {
        console.error(
                "An error happened in server. The HTTP status code is "  +
error.status + " and the error returned is " + error.message);
        }

        return throwError("Error occurred. Pleas try again");
}
```

The error function can be called in **get()** as specified below:

```
httpClient.get(url, options)
    .pipe(catchError(this.httpErrorHandler)
    .subscribe( (data) => console.log(data) )
```

Handle failed request

As we mentioned earlier, errors can happen and one way is to handle it. Another option is to try for certain number of times. If the request failed due to network issue or the HTTP server is temporarily offline, the next request may succeed.

We can use **rxjs** library's **retry** operator in this scenario as specified below:

```
httpClient.get(url, options)
    .pipe(
        retry(5),
        catchError(this.httpErrorHandler))
    .subscribe( (data) => console.log(data) )
```

Fetch expense entries

Let us do the actual coding to fetch the expenses from **Expense Rest API** in our **ExpenseManager** application.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Start the application.

```
ng serve
```

Add **getExpenseEntries()** and **httpErrorHandler()** method in **ExpenseEntryService** (src/app/expense-entry.service.ts) service.

```
getExpenseEntries() : Observable<ExpenseEntry[]> {
    return this.httpClient.get<ExpenseEntry[]>(this.expenseRestUrl,
this.httpOptions)
                            .pipe(
                                    retry(3),
                                    catchError(this.httpErrorHandler)
                            );
}


getExpenseEntry(id: number) : Observable<ExpenseEntry> {
    return this.httpClient.get<ExpenseEntry>(this.expenseRestUrl + "/" + id,
this.httpOptions)
    .pipe(
        retry(3),
        catchError(this.httpErrorHandler)
    );
}

private httpErrorHandler (error: HttpErrorResponse) {
        if (error.error instanceof ErrorEvent) {
        console.error("A client side error occurs. The error message is " +
error.message);
        } else {
        console.error(
                "An error happened in server. The HTTP status code is "  +
error.status + " and the error returned is " + error.message);
        }

        return throwError("Error occurred. Pleas try again");
}
```

Here,

- **getExpenseEntries()** calls the **get()** method using expense end point and also configures the error handler. Also, it configures **httpClient** to try for maximum of 3 times in case of failure. Finally, it returns the response from server as typed (**ExpenseEntry[]**) Observable object.
- **getExpenseEntry** is similar to getExpenseEntries() except it passes the id of the ExpenseEntry object and gets ExpenseEntry Observable object.

The complete coding of **ExpenseEntryService** is as follows:

```
import { Injectable } from '@angular/core';
import { ExpenseEntry } from './expense-entry';


import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
import { HttpClient, HttpHeaders, HttpErrorResponse } from
'@angular/common/http';

@Injectable({
```

```
        providedIn: 'root'
})
export class ExpenseEntryService {
        private expenseRestUrl = 'http://localhost:8000/api/expense';
        private httpOptions = {
                headers: new HttpHeaders( { 'Content-Type': 'application/json'
})
        };

    constructor(private httpClient : HttpClient) { }

    getExpenseEntries() : Observable<ExpenseEntry[]> {
        return this.httpClient.get<ExpenseEntry[]>(this.expenseRestUrl,
this.httpOptions)
        .pipe(
                retry(3),
                catchError(this.httpErrorHandler)
        );
    }

    getExpenseEntry(id: number) : Observable<ExpenseEntry> {
        return this.httpClient.get<ExpenseEntry>(this.expenseRestUrl + "/" +
id, this.httpOptions)
        .pipe(
                retry(3),
                catchError(this.httpErrorHandler)
        );
    }

    private httpErrorHandler (error: HttpErrorResponse) {
            if (error.error instanceof ErrorEvent) {
            console.error("A client side error occurs. The error message is " +
error.message);
            } else {
            console.error(
                    "An error happened in server. The HTTP status code is "  +
error.status + " and the error returned is " + error.message);
            }

            return throwError("Error occurred. Pleas try again");
    }
}
```

Open **ExpenseEntryListComponent** (src-entry-list-entry-list.component.ts) and inject **ExpenseEntryService** through constructor as specified below:

```
constructor(private debugService: DebugService, private restService :
ExpenseEntryService ) { }
```

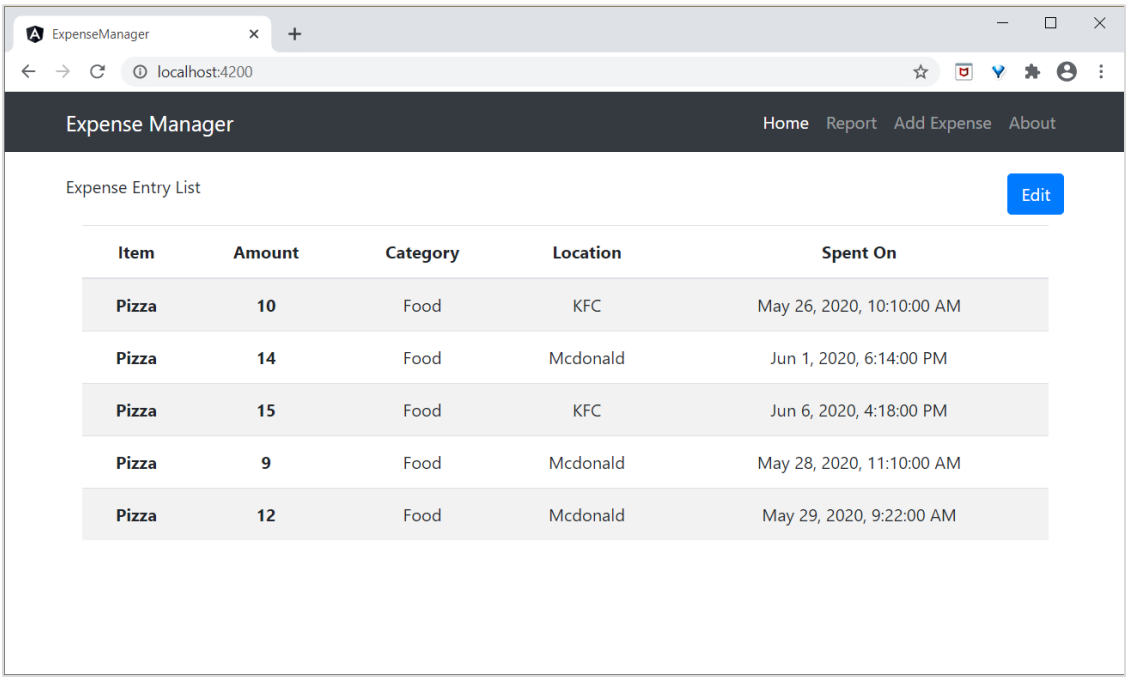
Change the **getExpenseEntries()** function. Call getExpenseEntries() method from **ExpenseEntryService** instead of returning the mock items.

```
getExpenseItems() {
        this.restService.getExpenseEntries()
                .subscribe( data => this.expenseEntries = data );
}
```

The complete **ExpenseEntryListComponent** coding is as follows:

```
import { Component, OnInit } from '@angular/core';
import { ExpenseEntry } from '../expense-entry';
import { DebugService } from '../debug.service';
import { ExpenseEntryService } from '../expense-entry.service';

@Component({
        selector: 'app-expense-entry-list',
        templateUrl: './expense-entry-list.component.html',
        styleUrls: ['./expense-entry-list.component.css'],
        providers: [DebugService]
})
export class ExpenseEntryListComponent implements OnInit {
        title: string;
        expenseEntries: ExpenseEntry[];
        constructor(private debugService: DebugService, private restService :
ExpenseEntryService ) { }

        ngOnInit() {
        this.debugService.info("Expense Entry List component initialized");
                this.title = "Expense Entry List";

                this.getExpenseItems();
        }

        getExpenseItems() {
                this.restService.getExpenseEntries()
                        .subscribe( data => this.expenseEntries = data );
        }
}
```

Finally, check the application and you will see the below response.



HTTP POST

HTTP POST is similar to HTTP GET except that the post request will send the necessary data as posted content along with the request. HTTP POST is used to insert new record into the system.

HttpClient provides **post()** method, which is similar to **get()** except it support extra argument to send the data to the server.

Let us add a new method, **addExpenseEntry()** in our **ExpenseEntryService** to add new expense entry as mentioned below:

```
addExpenseEntry(expenseEntry: ExpenseEntry): Observable<ExpenseEntry> {
    return this.httpClient.post<ExpenseEntry>(this.expenseRestUrl,
expenseEntry, this.httpOptions)
    .pipe(
            retry(3),
            catchError(this.httpErrorHandler)
    );
}
```

HTTP PUT

HTTP PUT is similar to HTTP POST request. HTTP PUT is used to update existing record in the system.

httpClient provides **put()** method, which is similar to **post().**

Update expense entry

Let us add a new method, **updateExpenseEntry()** in our **ExpenseEntryService** to update existing expense entry as mentioned below:

```
updateExpenseEntry(expenseEntry: ExpenseEntry): Observable<ExpenseEntry> {
    return this.httpClient.put<ExpenseEntry>(this.expenseRestUrl + "/" +
expenseEntry.id, expenseEntry, this.httpOptions)
    .pipe(
            retry(3),
            catchError(this.httpErrorHandler)
    );
}
```

HTTP DELETE

HTTP DELETE is similar to http GET request. HTTP DELETE is used to delete entries in the system.

httpclient provides **delete()** method, which is similar to **get().**

Delete expense entry

Let us add a new method, **deleteExpenseEntry()** in our **ExpenseEntryService** to delete existing expense entry as mentioned below:

```
deleteExpenseEntry(expenseEntry: ExpenseEntry | number) :
Observable<ExpenseEntry> {
    const id = typeof expenseEntry == 'number' ? expenseEntry : expenseEntry.id
    const url = `${this.expenseRestUrl}/${id}`;

    return this.httpClient.delete<ExpenseEntry>(url, this.httpOptions)
    .pipe(
            retry(3),
            catchError(this.httpErrorHandler)
    );
}
```

12. Angular 8 — Angular Material

Angular Material provides a huge collection of high-quality and ready-made Angular component based on Material design. Let us learn how to include Angular material in Angular application and use its component.

Configure Angular Material

Let us see how to configure Angular Material in Angular application.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Add Angular material package using below command:

```
ng add @angular/material
```

Angular CLI will ask certain question regarding theme, gesture recognition and browser animations. Select your any theme of your choice and then answer positively for gesture recognition and browser animation.

```
Installing packages for tooling via npm.
Installed packages for tooling via npm.
Choose a prebuilt theme name, or "custom" for a custom theme: Indigo/Pink
[ Preview: https://material.angular.i
o?theme=indigo-pink ]
Set up HammerJS for gesture recognition? Yes
Set up browser animations for Angular Material? Yes
```

Angular material packages each UI component in a separate module. Import all the necessary module into the application through root module (**src/app/app.module.ts**)

```
import { MatTableModule } from '@angular/material/table';
import { MatButtonModule } from '@angular/material/button';
import { MatIconModule } from '@angular/material/icon';

@NgModule({
  imports: [
    MatTableModule,
    MatButtonModule,
    MatIconModule
  ]
})
```

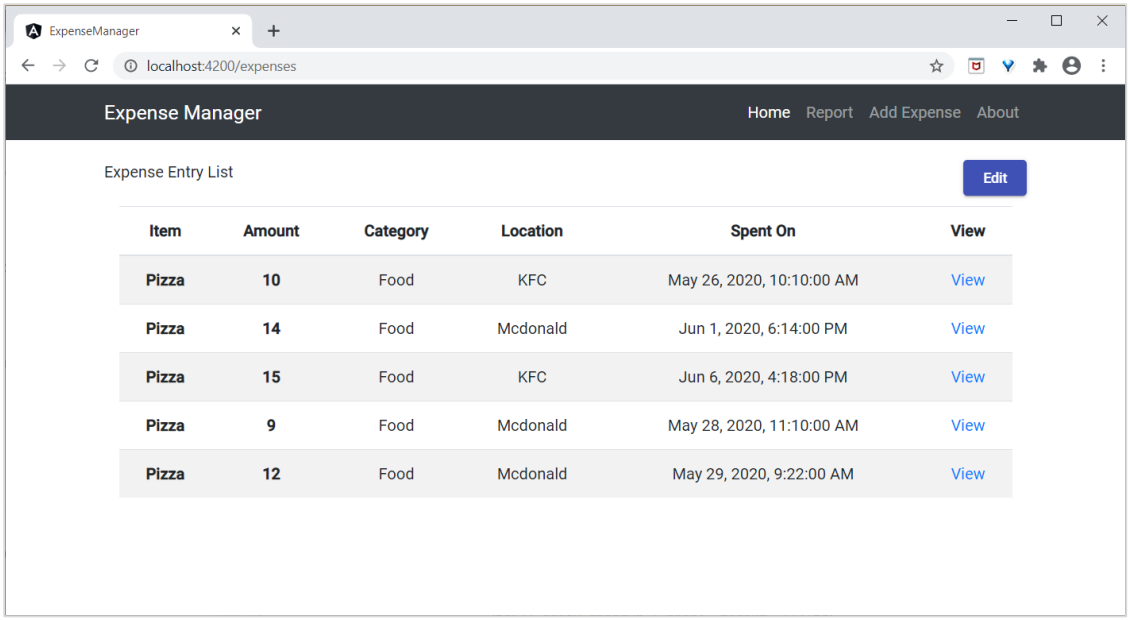
Change the edit button using **ExpenseEntryListComponent** template (**src/app/expense-entry-list/expense-entry-list.component.html**) as specified below:

```
<div class="col-sm" style="text-align: right;">
    <!-- <button type="button" class="btn btn-primary">Edit</button> -->
    <button mat-raised-button color="primary">Edit</button>
</div>
```

Run the application and test the page.

```
ng serve
```

The output of the application is as follows:



Here, the application clearly shows the Angular Material button.

Working example

Some of the important UI elements provided by Angular Material package.

- Form field
- Input
- Checkbox
- Radio button
- Select
- Button
- DatePicker
- List
- Card
- Grid list
- Table
- Paginator
- Tabs

- Toolbar
- Menu
- Dialog
- Snackbar
- Progress bar
- Icon
- Divider

Using material component is quite easy and we will learn one of the frequently used material component, **Material Table** by working on a sample project.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Let us change our **ExpenseEntryListComponent** (src/app/expense-entry-list/expense-entry-list.component.ts) and use Material Table component.

Declare a variable, displayedColumns and assign the list of column to be displayed.

```
displayedColumns: string[] = ['item', 'amount', 'category', 'location',
'spendOn' ];
```

Add material table as specified below in the **ExpenseEntryListComponent** template (**src/app/expense-entry-list/expense-entry-list.component.html**) and remove our existing list.

```
<div class="mat-elevation-z8">
    <table mat-table [dataSource]="expenseEntries">

            <ng-container matColumnDef="item">
                <th mat-header-cell *matHeaderCellDef> Item </th>
                <td mat-cell *matCellDef="let element" style="text-align:
left"> {{element.item}} </td>
            </ng-container>

            <ng-container matColumnDef="amount">
                <th mat-header-cell *matHeaderCellDef > Amount </th>
                <td mat-cell *matCellDef="let element" style="text-align:
left"> {{element.amount}} </td>
            </ng-container>

            <ng-container matColumnDef="category">
                <th mat-header-cell *matHeaderCellDef> Category </th>
                <td mat-cell *matCellDef="let element" style="text-align:
left"> {{element.category}} </td>
            </ng-container>

            <ng-container matColumnDef="location">
                <th mat-header-cell *matHeaderCellDef> Location </th>
                <td mat-cell *matCellDef="let element" style="text-align:
```

```
 left"> {{element.location}} </td>
            </ng-container>

            <ng-container matColumnDef="spendOn">
                <th mat-header-cell *matHeaderCellDef> Spend On </th>
                <td mat-cell *matCellDef="let element" style="text-align:
left"> {{element.spendOn}} </td>
            </ng-container>

            <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
            <tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
    </table>
</div>
```

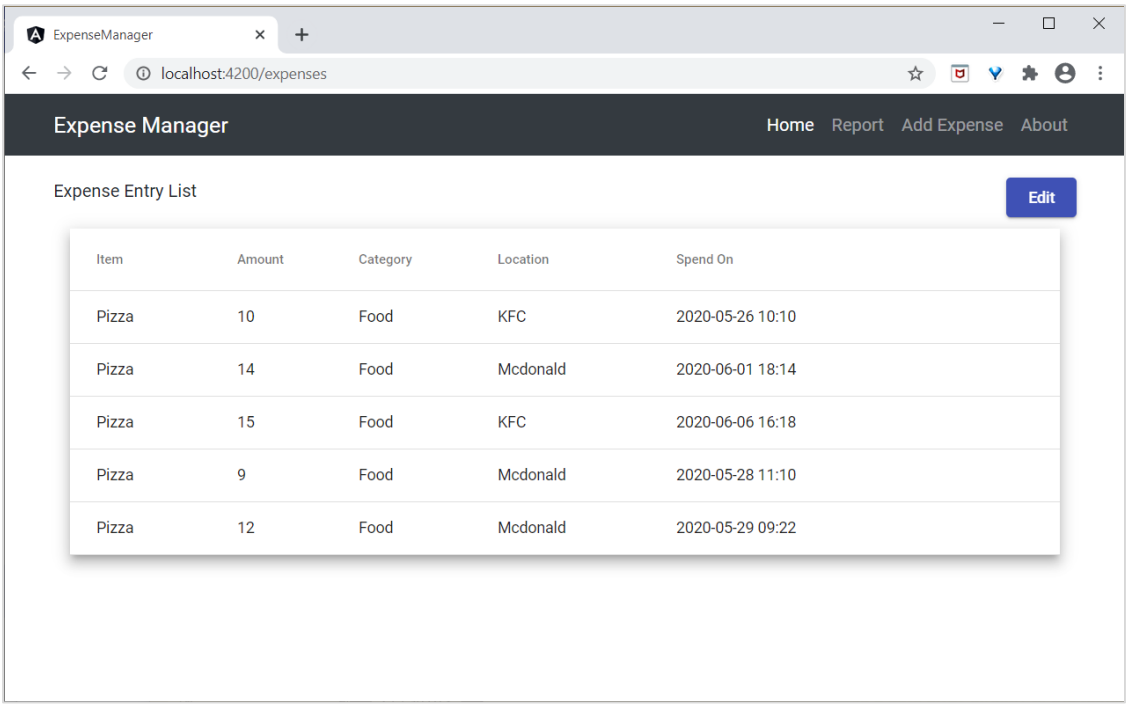
Here,

- **mat-table** property is used to convert the normal table in to material table.
- **[dataSource]** property is used to specify the data source of the table.
- Material table is template based and each column can be designed using separate template. **ng-container** is used to create template.
- **matColumnDef** is used to specify the column of the data source applied to the particular ng-container.
- **mat-header-cell** is used to specify the header text for each column.
- **mat-cell** is used to specify the content of each column.
- **mat-header-row** and **mat-row** is used to specify the order of the column in row.
- We have used only the basic features of the Material table. Material table has many more features such as sorting, pagination, etc.

Run the application.

```
ng serve
```

The output of the application is as follows:



13. Angular 8 — Routing and Navigation

Navigation is one of the important aspect in a web application. Even though a single page application (SPA) does not have multiple page concept, it does moves from one view (list of expenses) to another view (expense details). Providing clear and understandable navigation elements decides the success of an application.

Angular provides extensive set of navigation feature to accommodate simple scenario to complex scenario. The process of defining navigation element and the corresponding view is called **Routing**. Angular provides a separate module, **RouterModule** to set up the navigation in the Angular application. Let us learn the how to do the routing in Angular application in this chapter.

Configure Routing

Angular CLI provides complete support to setup routing during the application creation process as well as during working an application. Let us create a new application with router enabled using below command:

```
ng new routing-app --routing
```

Angular CLI generate a new module, **AppRoutingModuele** for routing purpose. The generated code is as follows:

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


const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Here,

- **Imports RouterModule** and Routes from @angular/router package.
- RouterMoudle provides functionality to configure and execute routing in the application.
- Routes is the type used to setup the navigation rules.
- Routes is the local variable (of type Routes) used to configure the actual navigation rules of the application.
- RouterMoudle.forRoot() method will setup the navigation rules configured in the routes variable.

Angular CLI include the generated **AppRoutingModule** in **AppComponent** as mentioned below:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

Here,

AppComponent imports the **AppRoutingModule** module using imports meta data.

Angular CLI provides option to set routing in the existing application as well. The general command to include routing in an existing application is as follows:

```
ng generate module my-module --routing
```

This will generate new module with routing features enabled. To enable routing feature in the existing module (*AppModule*), we need to include extra option as specified below:

```
ng generate module app-routing --module app --flat
```

Here,

–module app configures the newly created routing module, **AppRoutingModule** in the AppModule module.

Let us configure the routing module in **ExpenseManager** application.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Generate routing module using below command:

```
ng generate module app-routing --module app --flat
```

Output

The output is mentioned below:

```
CREATE src/app/app-routing.module.ts (196 bytes)
UPDATE src/app/app.module.ts (785 bytes)
```

Here,

CLI generate **AppRoutingModule** and then, configures it in **AppModule**

Creating routes

Creating a route is simple and easy. The basic information to create a route is given below:

- Target component to be called.
- The path to access the target component.

The code to create a simple route is mentioned below:

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

Here,

- **Routes** is the variable in the AppRoutingModule.
- **about** is the path and AboutComponent is the target / destination component. When user requests http://localhost:4200/about url, the path matches with about rule and then AboutComponent will be called.

Accessing routes

Let us learn how to use the configured routes in the application.

Accessing the route is a two step process.

Include **router-outlet** tag in the root component template.

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

Use **routerLink** and **routerLinkActive** property in the required place.

```
<a routerLink="/about" routerLinkActive="active">First Component</a>
```

Here,

- **routerLink** set the route to be called using the path.
- **routerLinkActive** set the CSS class to be used when the route is activated.

Sometime, we need to access routing inside the component instead of template. Then, we need to follow below steps:

Inject instance of **Router** and **ActivatedRoute** in the corresponding component.

```
import { Router, ActivatedRoute } from '@angular/router';

constructor(private router: Router, private route: ActivatedRoute)
```

Here,

- **Router** provides the function to do **routing operations**.
- **Route** refers the current **activate route**.

Use router's **navigate** function.

```
this.router.navigate(['about']);
```

Here,

navigate function expects an array with necessary path information.

Using relative path

Route path is similar to web page URL and it supports relative path as well. To access **AboutComponent** from another component, say **HomePageComponent**, simple use *..* notation as in web url or folder path.

```
<a routerLink="../about">Relative Route to about component</a>
```

To access relative path in the component:

```
import { NavigationExtras } from '@angular/router';

this.router.navigate(['about'], { relativeTo: this.route });
```

Here,

relativeTo is available under **NavigationExtras** class.

Route ordering

Route ordering is very important in a route configuration. If same path is configured multiple times, then the first matched path will get called. If the first match fails due to some reason, then the second match will get called.

Redirect routes

Angular route allows a path to get redirected to another path. **redirectTo** is the option to set redirection path. The sample route is as follows:

```
const routes: Routes = [
  { path: '',   redirectTo: '/about' },
];
```

Here,

- **redirectTo** sets about as the redirection path if the actual path matches empty string.

Wildcard routes

Wildcard route will match any path. It is created using ** and will be used to handle non existing path in the application. Placing the wildcard route at the end of the configuration make it called when other path is not matched.

The sample code is as follows:

```
const routes: Routes = [
  { path: 'about', component: AboutComponent },
  { path: '',   redirectTo: '/about', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a
404 page
];
```

Here,

If a non existent page is called, then the first two route gets failed. But, the final wildcard route will succeed and the **PageNotFoundComponent** gets called.

Access Route parameters

In Angular, we can attach extra information in the path using parameter. The parameter can be accessed in the component by using **paramMap** interface. The syntax to create a new parameter in the route is as follows:

```
const routes: Routes = [
  { path: 'about', component: AboutComponent },
  { path: 'item/:id', component: ItemComponent },
  { path: '',   redirectTo: '/about', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a
404 page
];
```

Here, we have attached **id** in the path. **id** can be accessed in the **ItemComponent** using two techniques.

- Using Observable.
- Using snapshot (non-observable option).

Using Observable

Angular provides a special interface, paramMap to access the parameter of the path. parmaMap has following methods:

- **has(name)** - Returns true if the specified name is available in the path (parameter list).
- **get(name)** - Returns the value of the specified name in the path (parameter list).
- **getAll(name)** - Returns the multiple value of the specified name in the path. get() method returns only the first value when multiple values are available.
- **keys** - Returns all parameter available in the path.

Steps to access the parameter using **paramMap** are as follows:

- Import **paramMap** available in **@angular/router** package.
- Use **paramMap** in the **ngOnInit()** to access the parameter and set it to a local variable.

```
ngOnInit() {
    this.route.paramMap.subscribe(params => {
        this.id = params.get('id);
    });
}
```

We can use it directly in the rest service using **pipe** method.

```
this.item$ = this.route.paramMap.pipe(
    switchMap(params => {
      this.selectedId = Number(params.get('id'));
      return this.service.getItem(this.selectedId);
    })
);
```

Using snapshot

snapshot is similar to **Observable** except, it does not support observable and get the parameter value immediately.

```
let id = this.route.snapshot.paramMap.get('id');
```

Nested routing

In general, **router-outlet** will be placed in root component (**AppComponent**) of the application. But, router-outlet can be used in any component. When router-outlet is used in a component other then root component, the routes for the particular component has to be configured as the children of the parent component. This is called **Nested routing**.

Let us consider a component, say **ItemComponent** is configured with **router-outlet** and has two **routerLink** as specified below:

```
<h2>Item Component</h2>

<nav>
  <ul>
    <li><a routerLink="view">View</a></li>
    <li><a routerLink="edit">Edit</a></li>
  </ul>

</nav>

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

The route for the **ItemComponent** has to be configured as **Nested routing** as specified below:

```
const routes: Routes = [
{
    path: 'item',
```

```
    component: ItemComponent,
    children: [
      {
        path: 'view',
        component: ItemViewComponent
      },
      {
        path: 'edit',
        component: ItemEditComponent
      }
    ]
}]
```

Working example

Let us apply the routing concept learned in this chapter in our **ExpenseManager** application.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Generate routing module using below command, if not done before.

```
ng generate module app-routing --module app --flat
```

Output

The output is as follows:

```
CREATE src/app/app-routing.module.ts (196 bytes)
UPDATE src/app/app.module.ts (785 bytes)
```

Here,

CLI generate **AppRoutingModule** and then configures it in **AppModule.**

Update **AppRoutingModule** *(***src/app/app.module.ts***)* as mentioned below:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ExpenseEntryComponent } from './expense-entry/expense-

entry.component';
import { ExpenseEntryListComponent } from './expense-entry-list/expense-entry-
list.component';

const routes: Routes = [
        { path: 'expenses', component: ExpenseEntryListComponent },
        { path: 'expenses/detail/:id', component: ExpenseEntryComponent },
        { path: '', redirectTo: 'expenses', pathMatch: 'full' }
```

```
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Here, we have added route for our expense list and expense details component.

Update **AppComponent** template (**src/app/app.component.html**) to include **router-outlet** and **routerLink.**

```
<!-- Navigation -->
<nav class="navbar navbar-expand-lg navbar-dark bg-dark static-top">
    <div class="container">
        <a class="navbar-brand" href="#">{{ title }}</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarResponsive" aria-controls="navbarResponsive" aria-
expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarResponsive">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item active">
                    <a class="nav-link" href="#">Home
                        <span class="sr-only" routerLink="/">(current)</span>
                    </a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" routerLink="/expenses">Report</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Add Expense</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">About</a>
                </li>
            </ul>
        </div>
    </div>
</nav>

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

Open **ExpenseEntryListComponent** template (src/app/expense-entry-list/expense-entry-list.component.html) and include view option for every expense entries.

```
<table class="table table-striped">
    <thead>
        <tr>
                <th>Item</th>
```

```
                <th>Amount</th>
                <th>Category</th>
                <th>Location</th>
                <th>Spent On</th>
                <th>View</th>
        </tr>
    </thead>
    <tbody>
            <tr *ngFor="let entry of expenseEntries">
                    <th scope="row">{{ entry.item }}</th>
                    <th>{{ entry.amount }}</th>
                    <td>{{ entry.category }}</td>
                    <td>{{ entry.location }}</td>
                    <td>{{ entry.spendOn | date: 'medium' }}</td>
                    <td><a routerLink="../expenses/detail/{{ entry.id
}}">View</a></td>
            </tr>
    </tbody>
</table>
```

Here, we have updated the expense list table and added a new column to show the view option.

Open **ExpenseEntryComponent** *(***src/app/expense-entry/expense-entry.component.ts***)* and add functionality to fetch the current selected expense entry. It can be done by first getting the id through the **paramMap** and then, using the **getExpenseEntry()** method from **ExpenseEntryService.**

```
        this.expenseEntry$ = this.route.paramMap.pipe(
                switchMap(params => {
                        this.selectedId = Number(params.get('id'));
                        return
this.restService.getExpenseEntry(this.selectedId);
                }));

        this.expenseEntry$.subscribe( (data) => this.expenseEntry = data );
```

Update **ExpenseEntryComponent** and add option to go to expense list.

```
    goToList() {
        this.router.navigate(['/expenses']);
    }
```

The complete code of **ExpenseEntryComponent** is as follows:

```
import { Component, OnInit } from '@angular/core';
import { ExpenseEntry } from '../expense-entry';
import { ExpenseEntryService } from '../expense-entry.service';
import { Router, ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
```

```
import { switchMap } from 'rxjs/operators';

@Component({
        selector: 'app-expense-entry',
        templateUrl: './expense-entry.component.html',
        styleUrls: ['./expense-entry.component.css']
})
export class ExpenseEntryComponent implements OnInit {
        title: string;
        expenseEntry$ : Observable<ExpenseEntry>;
        expenseEntry: ExpenseEntry = {} as ExpenseEntry;
        selectedId: number;

        constructor(private restService : ExpenseEntryService,
                                private router : Router, private route :
ActivatedRoute ) { }

        ngOnInit() {
                this.title = "Expense Entry";

        this.expenseEntry$ = this.route.paramMap.pipe(
                switchMap(params => {
                        this.selectedId = Number(params.get('id'));
                        return
this.restService.getExpenseEntry(this.selectedId);
                }));

        this.expenseEntry$.subscribe( (data) => this.expenseEntry = data );
    }

    goToList() {
        this.router.navigate(['/expenses']);
    }
}
```

Open **ExpenseEntryComponent (src/app/expense-entry/expense-entry.component.html)** template and add a new button to navigate back to expense list page.

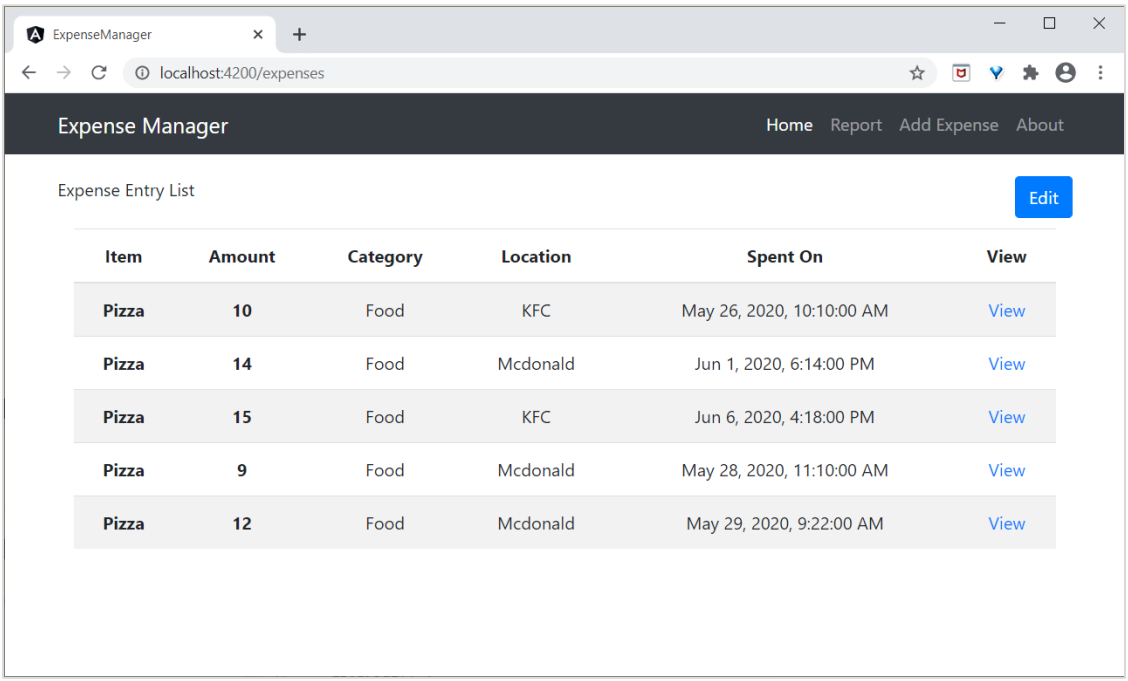
```
<div class="col-sm" style="text-align: right;">
    <button type="button" class="btn btn-primary" (click)="goToList()">Go to
List</button>
     <button type="button" class="btn btn-primary">Edit</button>
</div>
```

Here, we have added **Go to List** button before **Edit** button.

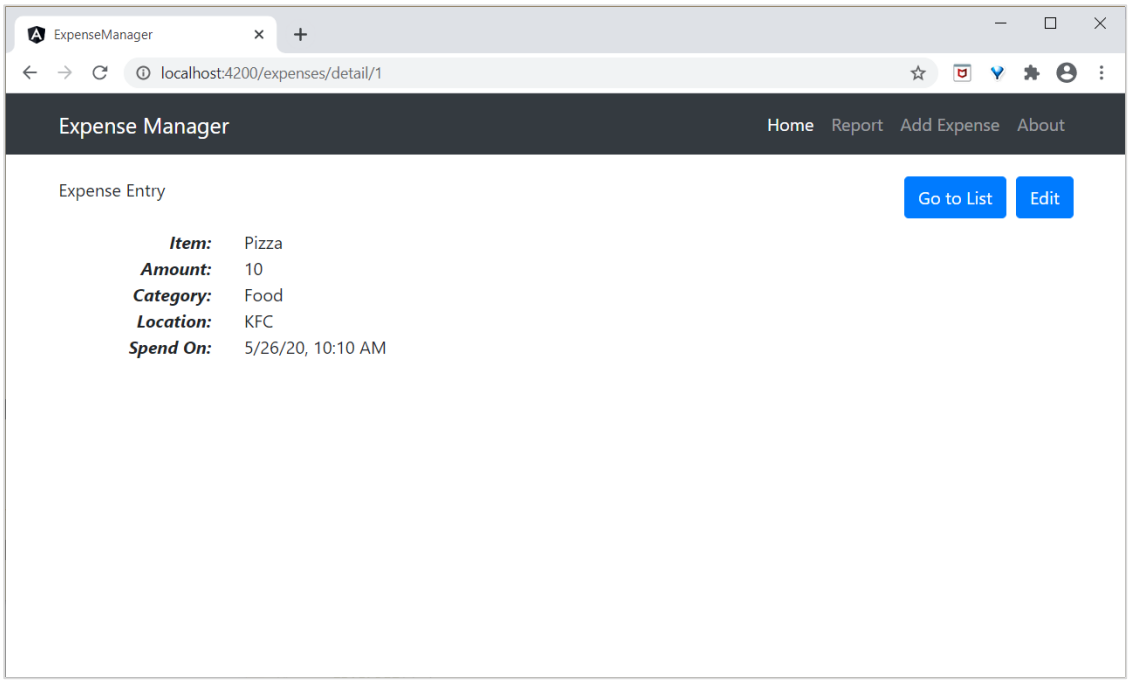
Run the application using below command:

```
ng serve
```

The final output of the application is as follows:



Clicking the view option of the first entry will navigate to details page and show the selected expense entry as shown below:



14. Angular 8 — Animations

Animation gives the web application a refreshing look and rich user interaction. In HTML, animation is basically the transformation of HTML element from one CSS style to another over a specific period of time. For example, an image element can be enlarged by changing its width and height.

If the width and height of the image is changed from initial value to final value in steps over a period of time, say 10 seconds, then we get an animation effect. So, the scope of the animation depends on the feature / property provided by the CSS to style a HTML element.

Angular provides a separate module **BrowserAnimationModule** to do the animation. **BrowserAnimationModule** provides an easy and clear approach to do animation.

Configuring animation module

Let us learn how to configure animation module in this chapter.

Follow below mentioned steps to configure animation module, **BrowserAnimationModule** in an application.

Import **BrowserAnimationModule** in **AppModule.**

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

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

Import animation function in the relevant components.

```
import { state, style, transition, animate, trigger } from
'@angular/animations'
```

Add **animations** metadata property in the relevant component.

```
@Component({
  animations: [
    // animation functionality goes here
  ]
})
export class MyAnimationComponent
```

Concepts

In angular, we need to understand the five core concept and its relationship to do animation.

State

State refers the specific state of the component. A component can have multiple defined state. The state is created using state() method. state() method has two arguments.

- **name** - Unique name of the state.
- **style** - Style of the state defined using style() method.

```
animations: [
    ...
    state('start', style( { width: 200px; } ))
    ...
]
```

Here, **start** is the name of the state.

Style

Style refers the CSS style applied in a particular state. style() method is used to style the particular state of a component. It uses the CSS property and can have multiple items.

```
animations: [
    ...
    state('start', style( { width: 200px; opacity: 1 } ))
    ...
]
```

Here, **start** state defines two CSS property, **width** with value 200px and opacity with value 1.

Transition

Transition refers the transition from one state to another. Animation can have multiple transition. Each transition is defined using **transition()** function. **transition()** takes two argument.

- Specifies the direction between two transition state. For example, **start => end** refers that the initial state is **start** and the final state is **end**. Actually, it is an expression with rich functionality.
- Specifies the animation details using **animate()** function.

```
animations: [
    ...
    transition('start => end', [
        animate('1s')
    ])

    ...
]
```

Here, **transition()** function defines the transition from start state to end state with animation defined in **animate()** method.

Animation

Animation defines the way the transition from one state to another take place. **animation()** function is used to set the animation details. **animate()** takes a single argument in the form of below expression:

```
duration delay easing
```

- **duration** refers the duration of the transition. It is expressed as 1s, 100ms, etc.,
- **delay** refers the delay time to start the transition. It is expressed similar to *duration*
- **easing** refers how do to accelerates / decelerates the transition in the given time duration.

Trigger

Every animation needs a trigger to start the animation. **trigger()** method is used to set all the animation information such as state, style, transition and animation in one place and give it a unique name. The unique name is used further to trigger the animation.

```
animations: [
    trigger('enlarge', [
            state('start', style({
                height: '200px',
                })),
            state('end', style({
                height: '500px',
                })),
            transition('start => end', [
                animate('1s')
            ]),
            transition('end => start', [
                animate('0.5s')
            ])
    ]),
]
```

Here, **enlarge** is the unique name given to the particular animation. It has two state and related styles. It has two transition one from start to end and another from end to start. End to start state do the reverse of the animation.

Trigger can be attached to an element as specified below:

```
<div [@triggerName]="expression">...</div>;
```

For example,

```
<img [@enlarge]="isEnlarge ? 'end' : 'start'">...</img>;
```

Here,

- **@enlarge** trigger is set to image tag and attrached to an expression.

- If **isEnlarge** value is changed to true, then **end** state will be set and it triggers **start => end** transition.
- If **isEnlarge** value is changed to false, then **start** state will be set and it triggers **end => start** transition.

Simple Animation Example

Let us write a new angular application to better understand the animation concept by enlarging an image with animation effect.

Open command prompt and create new angular application.

```
cd /go/to/workspace
ng new animation-app
cd animation-app
```

Configure **BrowserAnimationModule** in the **AppModule** *(*src/app/app.module.ts*)*

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

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

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

Open **AppComponent** *(***src/app/app.component.ts***)* and import necessary animation functions.

```
import { state, style, transition, animate, trigger } from
'@angular/animations';
```

Add animation functionality, which will animate the image during the enlarging / shrinking of the image.

```
@Component({
        selector: 'app-root',
        templateUrl: './app.component.html',
        styleUrls: ['./app.component.css'],
        animations: [
                trigger('enlarge', [
                        state('start', style({
                                height: '150px'
```

```
                        })),
                        state('end', style({
                                height: '250px'
                        })),
                        transition('start => end', [
                                animate('1s 2s')
                        ]),
                        transition('end => start', [
                                animate('1s 2s')
                        ])
                ])
        ]
})
```

Open **AppComponent** template, **src/app/app.component.html** and remove sample code. Then, include a header with application title, image and a button to enlarge / shrink the image.

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

<img src="assets/puppy.jpeg" style="height: 200px" />

<br />

<button>{{ this.buttonText }}</button>
```

Write a function to change the animation expression.

```
export class AppComponent {
        title = 'Animation Application';
        isEnlarge: boolean = false;
        buttonText: string = "Enlarge";

        triggerAnimation() {
                this.isEnlarge = !this.isEnlarge;

                if(this.isEnlarge)
                        this.buttonText = "Shrink";
                else
                        this.buttonText = "Enlarge";
        }
}
```

Attach the animation in the image tag. Also, attach the click event for the button.

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

<img [@enlarge]="isEnlarge ? 'end' : 'start'" src="assets/puppy.jpeg"
style="height: 200px" />

<br />

<button (click)='triggerAnimation()'>{{ this.buttonText }}</button>
```

Add a button and attach the function to trigger the animation.

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

<img [@enlarge]="isEnlarge ? 'end' : 'start'" src="assets/puppy.jpeg"
style="height: 200px" />

<br />

<button (click)='triggerAnimation()'>{{ this.buttonText }}</button>
```

The complete **AppComponent** code is as follows:

```
import { Component } from '@angular/core';
import { state, style, transition, animate, trigger } from
'@angular/animations';

@Component({
        selector: 'app-root',
        templateUrl: './app.component.html',
        styleUrls: ['./app.component.css'],
        animations: [
                trigger('enlarge', [
                        state('start', style({
                                height: '150px'
                        })),
                        state('end', style({
                                height: '250px'
                        })),
                        transition('start => end', [
                                animate('1s 2s')
                        ]),
                        transition('end => start', [
                                animate('1s 2s')
                        ])
                ])
        ]
})
export class AppComponent {
        title = 'Animation Application';
        isEnlarge: boolean = false;
        buttonText: string = "Enlarge";


        triggerAnimation() {
                this.isEnlarge = !this.isEnlarge;

                if(this.isEnlarge)
                        this.buttonText = "Shrink";
                else
                        this.buttonText = "Enlarge";
        }
}
```

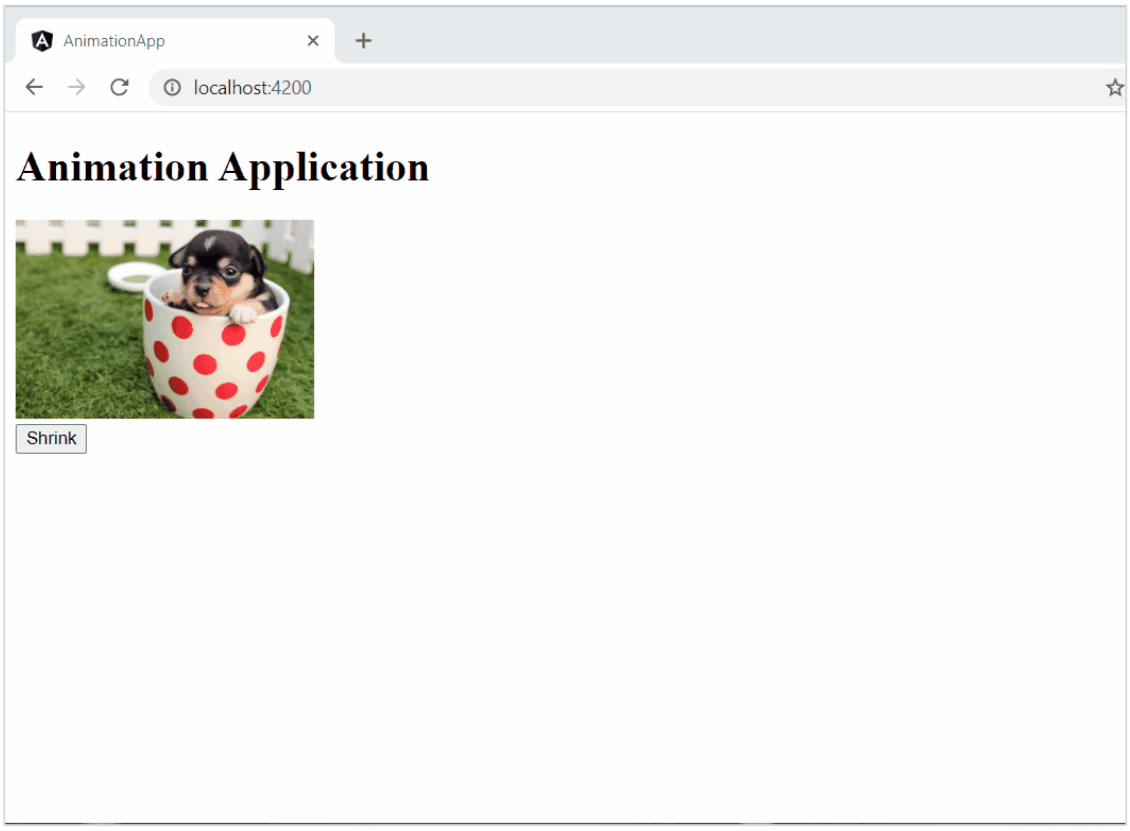
The complete **AppComponent** template code is as follows:

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

<img [@enlarge]="isEnlarge ? 'end' : 'start'" src="assets/puppy.jpeg"
style="height: 200px" />

<br />

<button (click)='triggerAnimation()'>{{ this.buttonText }}</button>
```

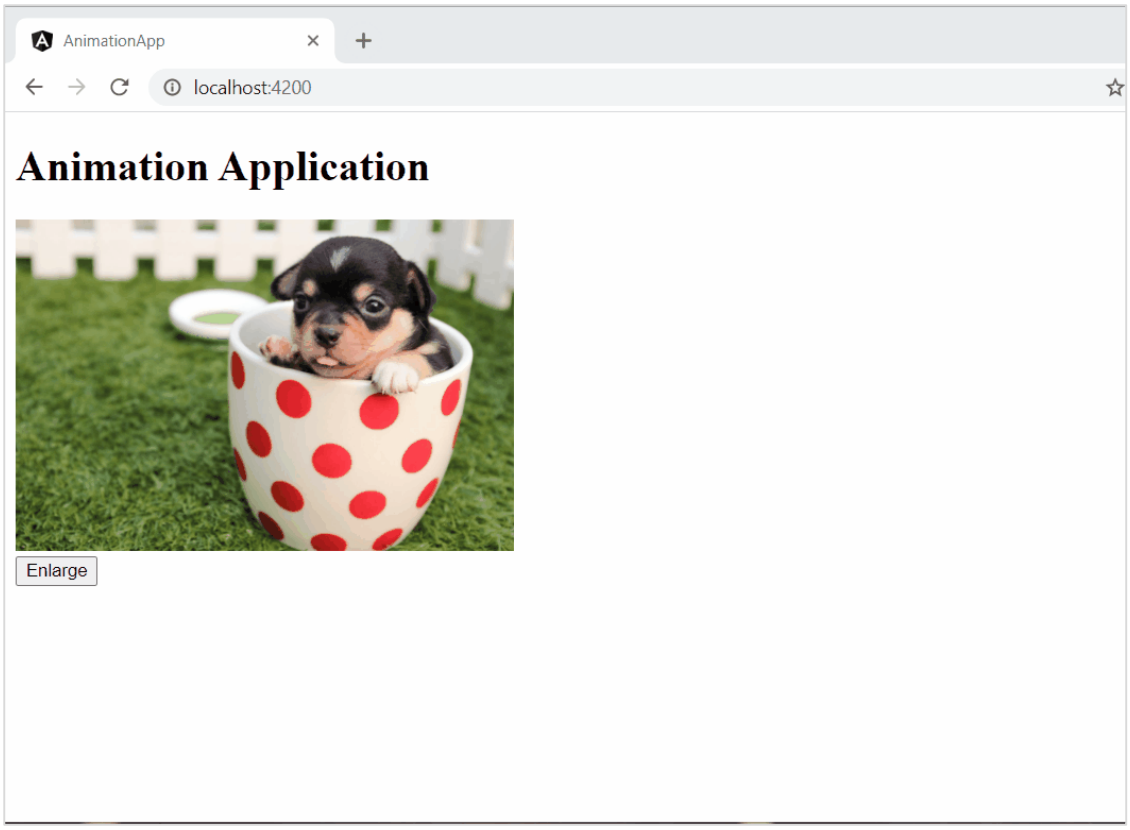
Run the application using below command:

```
ng serve
```

Click the enlarge button, it will enlarge the image with animation. The result will be as shown below:



Click the button again to shrink it. The result will be as shown below:


AnimationApp
localhost:4200
Animation Application
Enlarge

15. Angular 8 — Forms

Forms are used to handle user input data. Angular 8 supports two types of forms. They are **Template driven forms** and **Reactive forms**. This section explains about Angular 8 forms in detail.

Template driven forms

Template driven forms is created using directives in the template. It is mainly used for creating a simple form application. Let's understand how to create template driven forms in brief.

Configure Forms

Before understanding forms, let us learn how to configure forms in an application. To enable template driven forms, first we need to import **FormsModule** in **app.module.ts.** It is given below:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

//import FormsModule here
import { FormsModule } from '@angular/forms';

imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule   //Assign FormsModule
  ],
```

Once, **FormsModule** is imported, the application will be ready for form programming.

Create simple form

Let us create a sample application (**template-form-app**) in Angular 8 to learn the template driven form.

Open command prompt and create new Angular application using below command:

```
cd /go/to/workspace
ng new template-form-app
cd template-form-app
```

Configure **FormsModule** in **AppComponent** as shown below:

```
...
```

```
import { FormsModule } from '@angular/forms';

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

Create a test component using Angular CLI as mentioned below:

```
ng generate component test
```

The above create a new component and the output is as follows:

```
CREATE src/app/test/test.component.scss (0 bytes)
CREATE src/app/test/test.component.html (19 bytes)
CREATE src/app/test/test.component.spec.ts (614 bytes)
CREATE src/app/test/test.component.ts (262 bytes)
UPDATE src/app/app.module.ts (545 bytes)
```

Let's create a simple form to display user entered text.

Add the below code in **test.component.html** file as follows:

```
<form #userName="ngForm" (ngSubmit)="onClickSubmit(userName.value)">
  <input type="text" name="username" placeholder="username" ngModel>
  <br/>
  <br/>
  <input type="submit" value="submit">
</form>
```

Here, we used **ngModel** attribute in **input** text field.

Create **onClickSubmit()** method inside **test.component.ts** file as shown below:

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

@Component({
  selector: 'app-test',
  templateUrl: './test.component.html',
  styleUrls: ['./test.component.scss']
})
```

```
export class TestComponent implements OnInit {

  ngOnInit() {
  }
  onClickSubmit(result) {
    console.log("You have entered : " + result.username);
 }
}
```

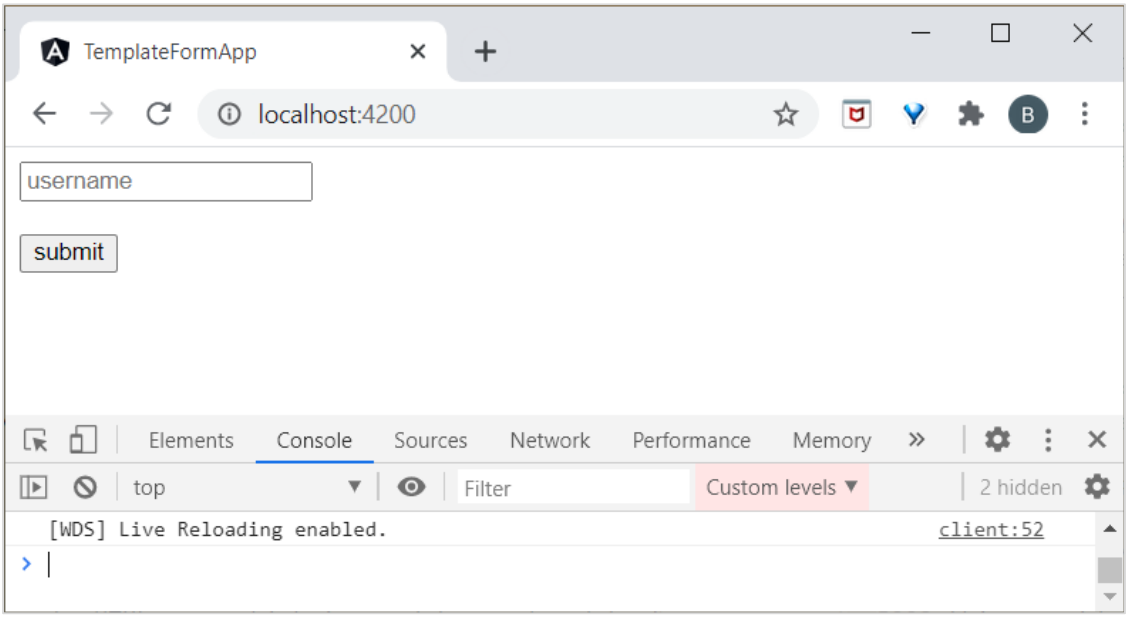
Open **app.component.html** and change the content as specified below:

```
<app-test></app-test>
```

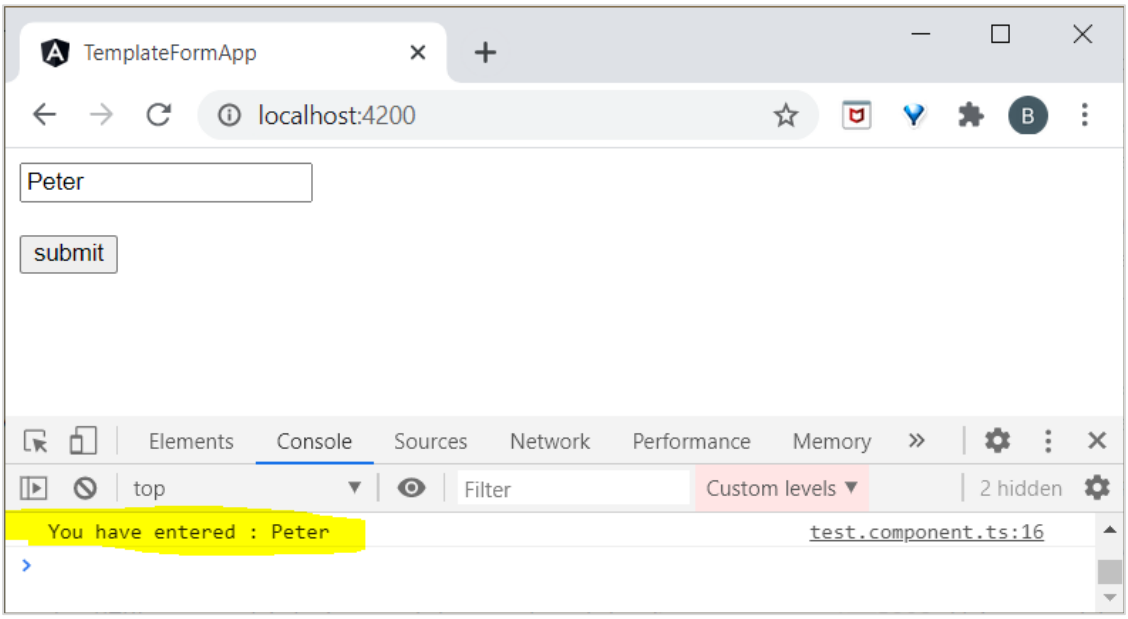
Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



Enter **Peter** in input text field and enter submit. **onClickSubmit** function will be called and user entered text **Peter** will be send as an argument. **onClickSubmit** will print the user name in the console and the output is as follows:



Reactive Forms

Reactive Forms is created inside component class so it is also referred as model driven forms. Every form control will have an object in the component and this provides greater control and flexibility in the form programming. **Reactive Form** is based on structured data model. Let's understand how to use Reactive forms in angular.

Configure Reactive forms

To enable reactive forms, first we need to import **ReactiveFormsModule** in **app.module.ts**. It is defined below:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { TestComponent } from './test/test.component';
import { FormsModule } from '@angular/forms';

//import ReactiveFormsModule here
import { ReactiveFormsModule } from '@angular/forms';

  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule   //Assign here
  ]
```

Create Reactive forms

Before moving to create Reactive forms, we need to understand about the following concepts,

- **FormControl** - Define basic functionality of individual form control
- **FormGroup** - Used to aggregate the values of collection form control
- **FormArray** - Used to aggregate the values of form control into an array
- **ControlValueAccessor** - Acts as an interface between Forms API to HTML DOM elements.

Let us create a sample application (**reactive-form-app**) in Angular 8 to learn the template driven form.

Open command prompt and create new Angular application using below command:

```
cd /go/to/workspace
ng new reactive-form-app
cd reactive-form-app
```

Configure **ReactiveFormsModule** in **AppComponent** as shown below:

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

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

Create a **test** component using Angular CLI as mentioned below:

```
ng generate component test
```

The above create a new component and the output is as follows:

```
CREATE src/app/test/test.component.scss (0 bytes)
CREATE src/app/test/test.component.html (19 bytes)
CREATE src/app/test/test.component.spec.ts (614 bytes)
CREATE src/app/test/test.component.ts (262 bytes)
UPDATE src/app/app.module.ts (545 bytes)
```

Let's create a simple form to display user entered text.

We need to import **FormGroup, FormControl** classes in **TestComponent.**

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

Create **onClickSubmit()** method inside **test.component.ts** file as shown below:

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

@Component({
  selector: 'app-test',
  templateUrl: './test.component.html',
  styleUrls: ['./test.component.css']
})
export class TestComponent implements OnInit {
  userName;
  formdata;
  ngOnInit() {
    this.formdata = new FormGroup({
       userName: new FormControl("Tutorialspoint")
    });
 }
 onClickSubmit(data) {this.userName = data.userName;}
}
```

Here,

- Created an instance of **formGroup** and set it to local variable, formdata.
- Crete an instance of **FormControl** and set it one of the entry in formdata.
- Created a **onClickSubmit()** method, which sets the local variable, **userName** with its argument.

Add the below code in **test.component.html** file.

```
<div>
  <form [formGroup]="formdata" (ngSubmit)="onClickSubmit(formdata.value)" >
     <input type= text"  name="userName" placeholder="userName"
        formControlName = "userName">
     <br/>
     <br/>
     <input type="submit"  value="Click here">
  </form>
</div>
<p> Textbox result is: {{userName}} </p>
```

Here,

- New form is created and set it's **formGroup** property to formdata.
- New input text field is created and set is **formControlName** to **username.**
- **ngSubmit** event property is used in the form and set **onClickSubmit()** method as its value.
- **onClickSubmit()** method gets formdata values as its arguments.

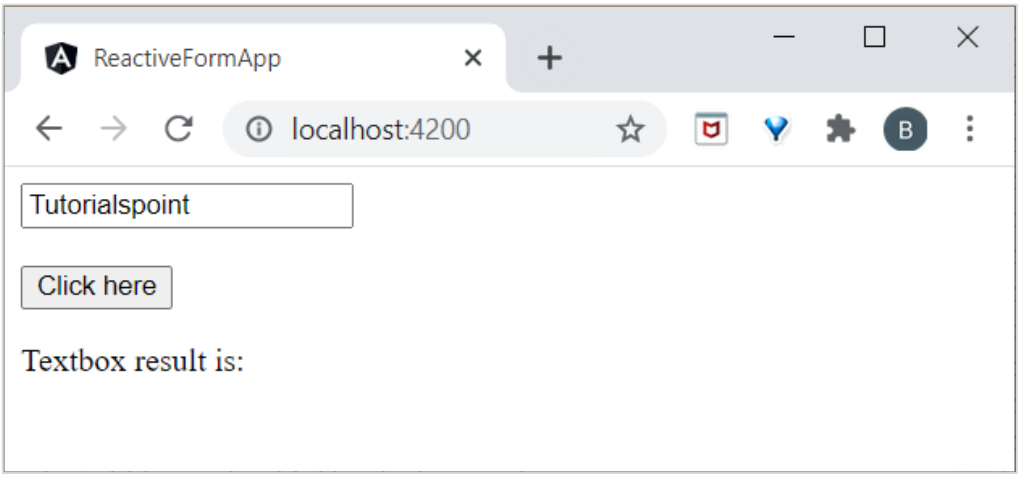
Open **app.component.html** and change the content as specified below:

```
<app-test></app-test>
```

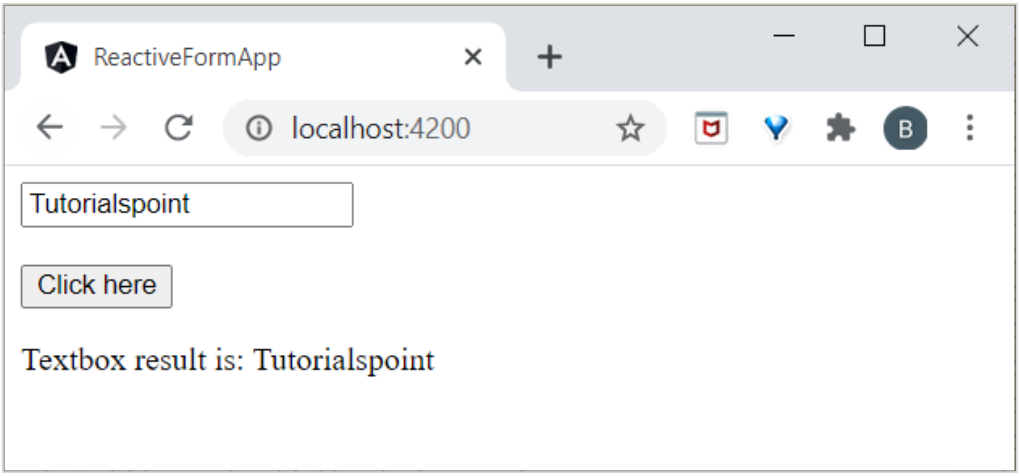
Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now, run your application and you could see the below response:



Enter **Tutorialspoint** in input text field and enter submit. **onClickSubmit** function will be called and user entered text **Peter** will be send as an argument.



We will perform Forms validation in next chapter.

16. Angular 8 — Form Validation

Form validation is an important part of web application. It is used to validate whether the user input is in correct format or not.

RequiredValidator

Let's perform simple required field validation in angular.

Open command prompt and go to **reactive-form-app.**

```
cd /go/to/reactive-form-app
```

Replace the below code in **test.component.ts** file.

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

//import validator and FormBuilder
import { FormGroup, FormControl, Validators, FormBuilder } from
'@angular/forms';

@Component({
        selector: 'app-test',
        templateUrl: './test.component.html',
        styleUrls: ['./test.component.css']
})

export class TestComponent implements OnInit {
        //Create FormGroup
        requiredForm: FormGroup;
        constructor(private fb: FormBuilder) {
                this.myForm();
        }

        //Create required field validator for name
        myForm() {
                this.requiredForm = this.fb.group({
                name: ['', Validators.required ]
                });
        }
        ngOnInit()
        {

        }
}
```

Here,

We have used form builder to handle all the validation. Constructor is used to create a form with the validation rules.

Add the below code inside **test.component.html** file.

```
<div>
   <h2>
     Required Field validation
   </h2>
   <form [formGroup]="requiredForm" novalidate>
         <div class="form-group">
           <label class="center-block">Name:
             <input class="form-control" formControlName="name">
           </label>
         </div>
         <div *ngIf="requiredForm.controls['name'].invalid &&
requiredForm.controls['name'].touched" class="alert alert-danger">
             <div *ngIf="requiredForm.controls['name'].errors.required">
             Name is required.
           </div>
         </div>
   </form>
 <p>Form value: {{ requiredForm.value | json }}</p>
 <p>Form status: {{ requiredForm.status | json }}</p>
 </div>
```

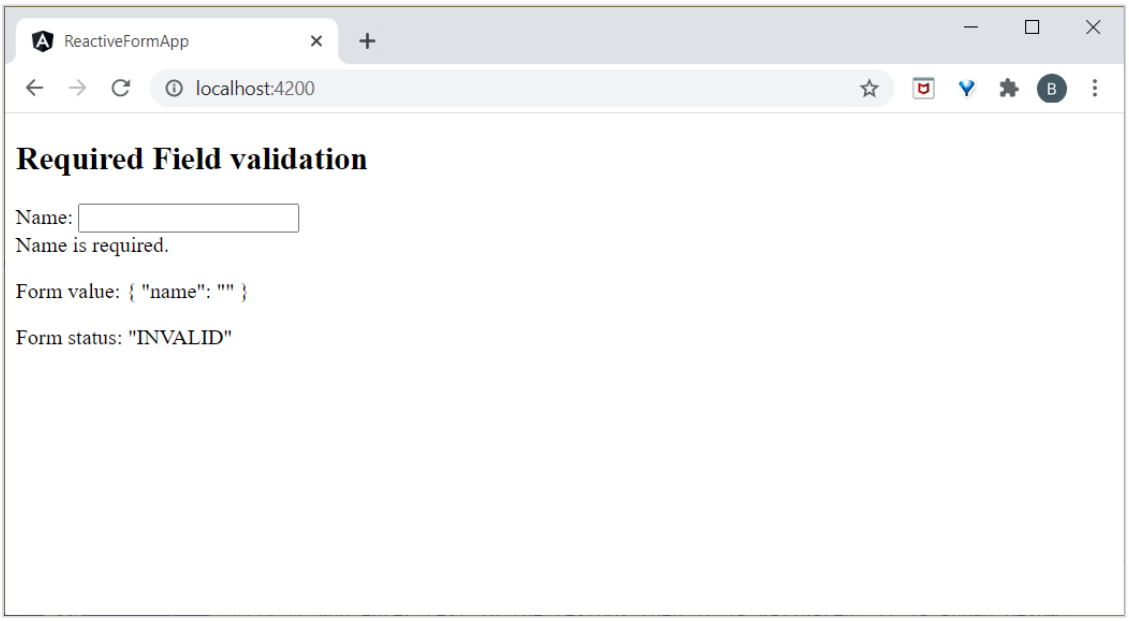
Here,

- **requiredForm** is called global form group object. It is a parent element. Form controls are childrens of requiredForm.
- Conditional statement is used to check, if a user has touched the input field but not enter the values then, it displays the error message.

Finally, start your application (if not done already) using the below command:

```
ng serve
```

Now run your application and put focus on text box. Then, it will use show **Name is required** as shown below:



If you enter text in the textbox, then it is validated and the output is shown below:

PatternValidator

PatternValidator is used to validate regex pattern. Let's perform simple email validation.

Open command prompt and to **reactive-form-app.**

```
cd /go/to/reactive-form-app
```

Replace below code in **test.component.ts** file:

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

import { FormGroup, FormControl, Validators, FormBuilder } from
```

```
'@angular/forms';

@Component({
        selector: 'app-test',
        templateUrl: './test.component.html',
        styleUrls: ['./test.component.css']
})

export class TestComponent implements OnInit {
        requiredForm: FormGroup;
        constructor(private fb: FormBuilder) {
                this.myForm();
        }

        myForm() {
                this.requiredForm = this.fb.group({
                email: ['', [Validators.required,
                        Validators.pattern("^[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-
z]{2,4}$")] ]
                });
        }

        ngOnInit()
        {

        }
}
```

Here,

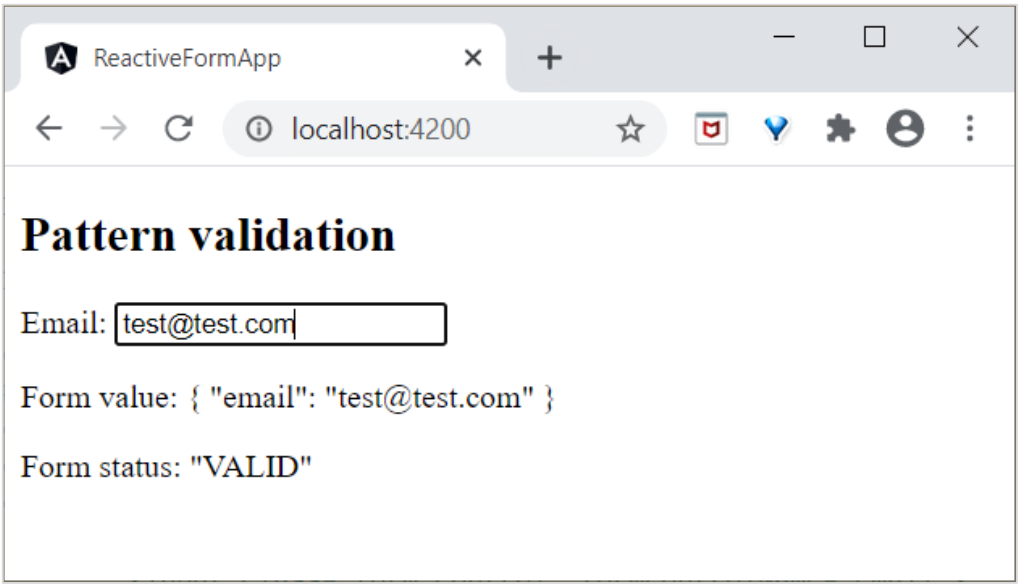
Added email pattern validator inside the Validator.

Update below code in **test.component.html** file:

```
<div>
        <h2>
        Pattern validation
        </h2>
        <form [formGroup]="requiredForm" novalidate>
        <div class="form-group">
                <label class="center-block">Email:
                <input class="form-control" formControlName="email">
                </label>
        </div>
        <div *ngIf="requiredForm.controls['email'].invalid &&
requiredForm.controls['email'].touched" class="alert alert-danger">
            <div *ngIf="requiredForm.controls['email'].errors.required">
                Email is required.
                </div>
        </div>
        </form>
        <p>Form value: {{ requiredForm.value | json }}</p>
        <p>Form status: {{ requiredForm.status | json }}</p>
</div>
```

Here, we have created the email control and called email validator.

Run your application and you could see the below result:



Similarly, you can try yourself to perform other types of validators.

17. Angular 8 — Authentication and Authorization

Authentication is the process matching the visitor of a web application with the pre-defined set of user identity in the system. In other word, it is the process of recognizing the user's identity. Authentication is very important process in the system with respect to security.

Authorization is the process of giving permission to the user to access certain resource in the system. Only the **authenticated user** can be authorised to access a resource.

Let us learn how to do Authentication and Authorization in Angular application in this chapter.

Guards in Routing

In a web application, a resource is referred by url. Every user in the system will be allowed access a set of urls. For example, an administrator may be assigned all the url coming under administration section.

As we know already, URLs are handled by **Routing**. Angular routing enables the urls to be guarded and restricted based on programming logic. So, a url may be denied for a normal user and allowed for an administrator.

Angular provides a concept called **Router Guards** which can be used to prevent unauthorised access to certain part of the application through routing. Angular provides multiple guards and they are as follows:

- **CanActivate** - Used to stop the access to a route.
- **CanActivateChild** - Used to stop the access to a child route.
- **CanDeactivate** - Used to stop ongoing process getting feedback from user. For example, **delete** process can be stop if the user replies in negative.
- **Resolve** - Used to pre-fetch the data before navigating to the route.
- **CanLoad** - Used to load assets.

Working example

Let us try to add login functionality to our application and secure it using **CanActivate** guard.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Start the application.

```
ng serve
```

Create a new service, **AuthService** to authenticate the user.

```
ng generate service auth
```

```
CREATE src/app/auth.service.spec.ts (323 bytes)
CREATE src/app/auth.service.ts (133 bytes)
```

Open **AuthService** and include below code.

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

import { Observable, of } from 'rxjs';
import { tap, delay } from 'rxjs/operators';

@Injectable({
        providedIn: 'root'
})
export class AuthService {

        isUserLoggedIn: boolean = false;

        login(userName: string, password: string): Observable<boolean> {
                console.log(userName);
                console.log(password);
                this.isUserLoggedIn = userName == 'admin' && password ==
'admin';
                localStorage.setItem('isUserLoggedIn', this.isUserLoggedIn ?
"true" : "false");

        return of(this.isUserLoggedIn).pipe(
                delay(1000),
                tap(val => {
                        console.log("Is User Authentication is successful: " +
val);
                })
        );
        }

        logout(): void {
        this.isUserLoggedIn = false;
                localStorage.removeItem('isUserLoggedIn');
        }

        constructor() { }
}
```

Here,

- We have written two methods, **login** and **logout.**
- The purpose of the **login** method is to validate the user and if the user successfully validated, it stores the information in **localStorage** and then returns true.
- Authentication validation is that the user name and password should be **admin.**
- We have not used any backend. Instead, we have simulated a delay of 1s using Observables.
- The purpose of the **logout** method is to invalidate the user and removes the information stored in **localStorage**.

Create a **login** component using below command:

```
ng generate component login

CREATE src/app/login/login.component.html (20 bytes)
CREATE src/app/login/login.component.spec.ts (621 bytes)
CREATE src/app/login/login.component.ts (265 bytes)
CREATE src/app/login/login.component.css (0 bytes)
UPDATE src/app/app.module.ts (1207 bytes)
```

Open **LoginComponent** and include below code:

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

import { FormGroup, FormControl } from '@angular/forms';
import { AuthService } from '../auth.service';
import { Router } from '@angular/router';

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

        userName: string;
        password: string;
        formData: FormGroup;

        constructor(private authService : AuthService, private router : Router)
{ }

        ngOnInit() {
                this.formData = new FormGroup({
                        userName: new FormControl("admin"),
                        password: new FormControl("admin"),
                });
        }

        onClickSubmit(data: any) {
                this.userName = data.userName;
                this.password = data.password;

                console.log("Login page: " + this.userName);
                console.log("Login page: " + this.password);

                this.authService.login(this.userName, this.password)
                        .subscribe( data => {
                                console.log("Is Login Success: " + data);

                     if(data) this.router.navigate(['/expenses']);
                });
        }
}
```

Here,

- Used reactive forms.
- Imported **AuthService** and **Router** and configured it in constructor.
- Created an instance of **FormGroup** and included two instance of **FormControl**, one for **user name** and another for **password.**
- Created a **onClickSubmit** to validate the user using **authService** and if successful, navigate to expense list.

Open **LoginComponent** template and include below template code.

```
<!-- Page Content -->
<div class="container">
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">
            <div class="container box" style="margin-top: 10px; padding-left:
0px; padding-right: 0px;">
                <div class="row">
                    <div class="col-12" style="text-align: center;">
                                                <form [formGroup]="formData"
(ngSubmit)="onClickSubmit(formData.value)"
                                                        class="form-signin">
                                                <h2 class="form-signin-
heading">Please sign in</h2>
                                                <label for="inputEmail"
class="sr-only">Email address</label>
                                                <input type="text"
id="username" class="form-control"

formControlName="userName" placeholder="Username" required autofocus>
                                                <label for="inputPassword"
class="sr-only">Password</label>
                                                <input type="password"
id="inputPassword" class="form-control"

formControlName="password" placeholder="Password" required>
                                                <button class="btn btn-lg btn-
primary btn-block" type="submit">Sign in</button>
                                                </form>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

Here,

Created a reactive form and designed a login form.

Attached the **onClickSubmit** method to the form submit action.

Open **LoginComponent** style and include below CSS Code.

```
.form-signin {
        max-width: 330px;
```

```
        padding: 15px;
        margin: 0 auto;
}

input {
        margin-bottom: 20px;
}
```

Here, some styles are added to design the login form.

Create a logout component using below command:

```
ng generate component logout

CREATE src/app/logout/logout.component.html (21 bytes)
CREATE src/app/logout/logout.component.spec.ts (628 bytes)
CREATE src/app/logout/logout.component.ts (269 bytes)
CREATE src/app/logout/logout.component.css (0 bytes)
UPDATE src/app/app.module.ts (1368 bytes)
```

Open **LogoutComponent** and include below code.

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

import { AuthService } from '../auth.service';
import { Router } from '@angular/router';

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

        constructor(private authService : AuthService, private router: Router)
{ }

        ngOnInit() {
                this.authService.logout();
                this.router.navigate(['/']);
        }

}
```

Here,

- Used logout method of AuthService.
- Once the user is logged out, the page will redirect to home page (/).

Create a guard using below command:

```
ng generate guard expense
```

```
CREATE src/app/expense.guard.spec.ts (364 bytes)
CREATE src/app/expense.guard.ts (459 bytes)
```

Open **ExpenseGuard** and include below code:

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router,
UrlTree } from '@angular/router';
import { Observable } from 'rxjs';

import { AuthService } from './auth.service';

@Injectable({
        providedIn: 'root'
})
export class ExpenseGuard implements CanActivate {

        constructor(private authService: AuthService, private router: Router)
{}

        canActivate(
        next: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): boolean | UrlTree {
                let url: string = state.url;

                    return this.checkLogin(url);
                }

                checkLogin(url: string): true | UrlTree {
                        console.log("Url: " + url)
                        let val: string =
localStorage.getItem('isUserLoggedIn');

                        if(val != null && val == "true"){
                                if(url == "/login")
                                        this.router.parseUrl('/expenses');
                                else
                                        return true;
                        } else {
                                return this.router.parseUrl('/login');
                        }
                }
}
```

Here,

- **checkLogin** will check whether the localStorage has the user information and if it is available, then it returns true.
- If the user is logged in and goes to login page, it will redirect the user to **expenses** page
- If the user is not logged in, then the user will be redirected to **login** page.

Open **AppRoutingModule (src/app/app-routing.module.ts)** and update below code:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ExpenseEntryComponent } from './expense-entry/expense-
entry.component';
import { ExpenseEntryListComponent } from './expense-entry-list/expense-entry-
list.component';
import { LoginComponent } from './login/login.component';
import { LogoutComponent } from './logout/logout.component';

import { ExpenseGuard } from './expense.guard';

const routes: Routes = [
        { path: 'login', component: LoginComponent },
        { path: 'logout', component: LogoutComponent },
        { path: 'expenses', component: ExpenseEntryListComponent, canActivate:
[ExpenseGuard]},
        { path: 'expenses/detail/:id', component: ExpenseEntryComponent,
canActivate: [ExpenseGuard]},
        { path: '', redirectTo: 'expenses', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Here,

- Imported **LoginComponent** and **LogoutComponent**.
- Imported ExpenseGuard.
- Created two new routes, login and logout to access LoginComponent and LogoutComponent respectively.
- Add new option canActivate for ExpenseEntryComponent and ExpenseEntryListComponent.

Open **AppComponent** template and add two login and logout link.

```
<div class="collapse navbar-collapse" id="navbarResponsive">
    <ul class="navbar-nav ml-auto">
        <li class="nav-item active">
            <a class="nav-link" href="#">Home
                <span class="sr-only" routerLink="/">(current)</span>


            </a>
        </li>
        <li class="nav-item">
            <a class="nav-link" routerLink="/expenses">Report</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="#">Add Expense</a>
        </li>
        <li class="nav-item">
```

```
            <a class="nav-link" href="#">About</a>
        </li>
        <li class="nav-item">
                        <div *ngIf="isUserLoggedIn; else isLogOut">
                                <a class="nav-link"
routerLink="/logout">Logout</a>
                        </div>

                        <ng-template #isLogOut>
                                        <a class="nav-link"
routerLink="/login">Login</a>
                        </ng-template>
        </li>
    </ul>
</div>
```

Open **AppComponent** and update below code:

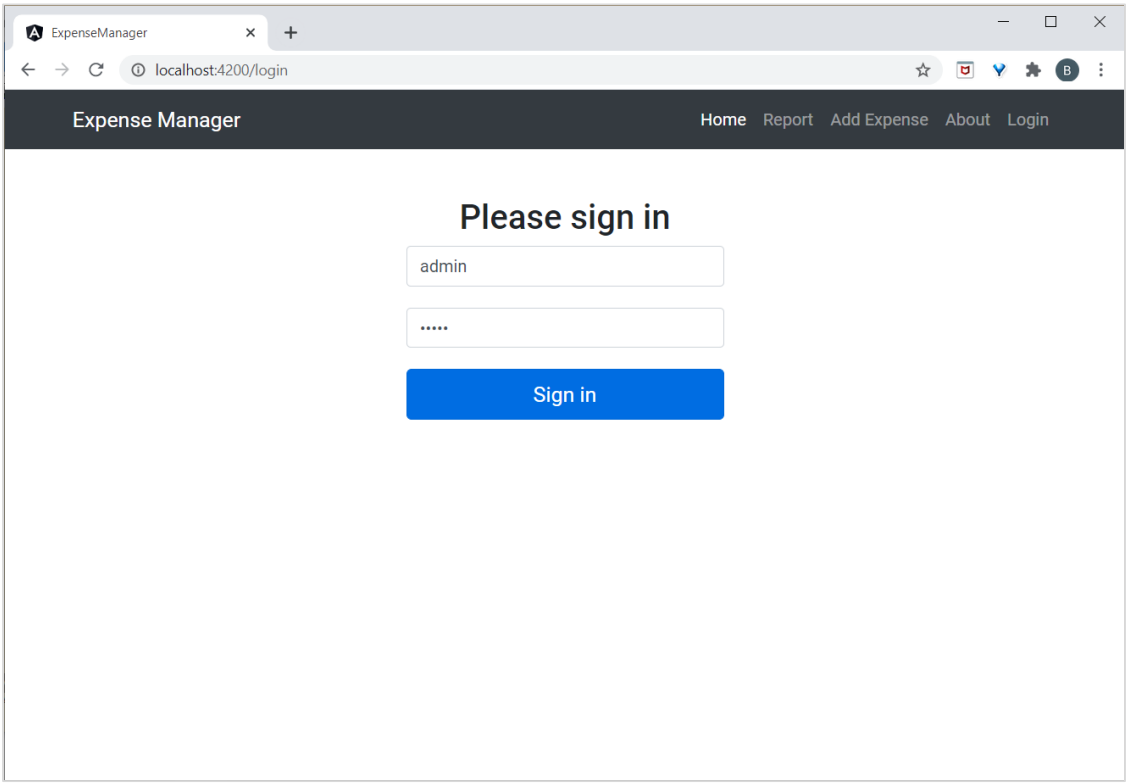
```
import { Component } from '@angular/core';

import { AuthService } from './auth.service';

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

        title = 'Expense Manager';
        isUserLoggedIn = false;

        constructor(private authService: AuthService) {}

        ngOnInit() {
                let storeData = localStorage.getItem("isUserLoggedIn");
                console.log("StoreData: " + storeData);

                if( storeData != null && storeData == "true")
                        this.isUserLoggedIn = true;
                else



                        this.isUserLoggedIn = false;
        }
}
```

Here, we have added the logic to identify the user status so that we can show **login / logout** functionality.

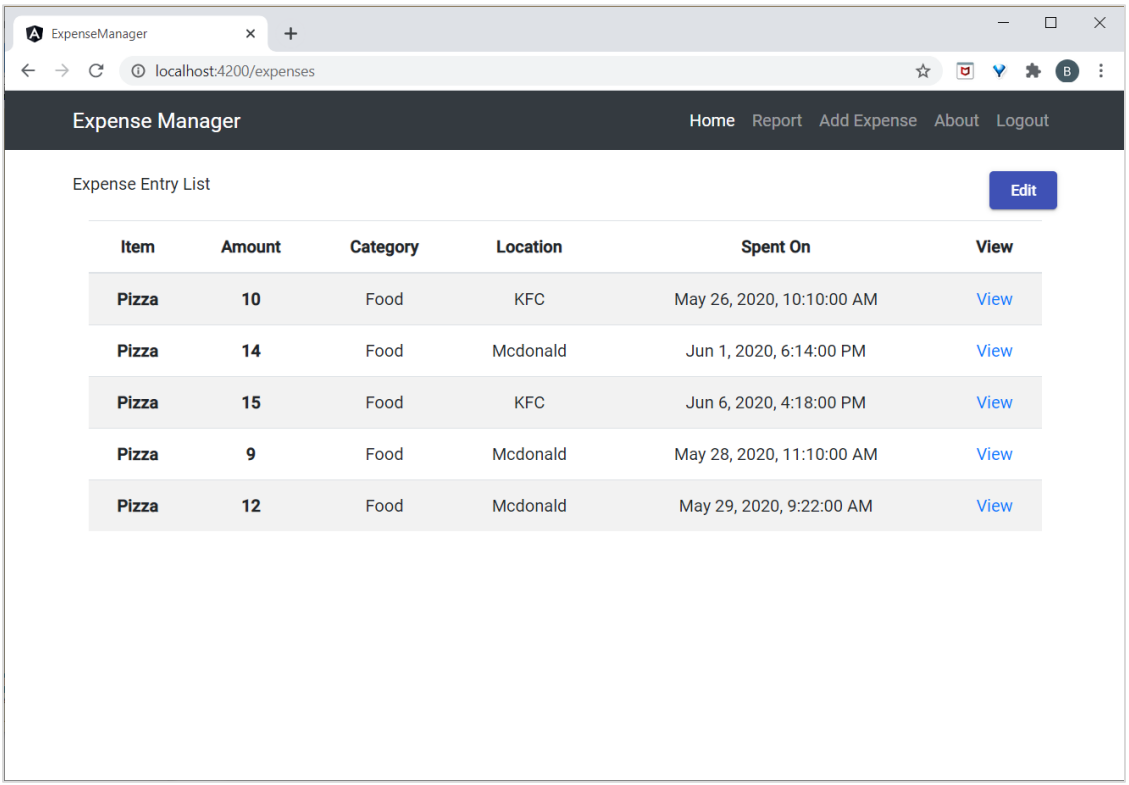
Open **AppModule (src/app/app.module.ts)** and configure **ReactiveFormsModule.**

```
import { ReactiveFormsModule } from '@angular/forms';

imports: [
    ReactiveFormsModule
]
```

Now, run the application and the application opens the login page.



Enter **admin** and **admin** as **username** and **password** and then, click submit. The application process the login and redirects the user to expense list page as shown below:



Finally, your can click logout and exit the application.

18. Angular 8 — Web Workers

Web workers enables JavaScript application to run the CPU-intensive in the background so that the application main thread concentrate on the smooth operation of UI. Angular provides support for including Web workers in the application. Let us write a simple Angular application and try to use web workers.

Create a new Angular application using below command:

```
cd /go/to/workspace
ng new web-worker-sample
```

Run the application using below command:

```
cd web-worker-sample
npm run start
```

Add new web worker using below command:

```
ng generate web-worker app
```

The output of the above command is as follows:

```
CREATE tsconfig.worker.json (212 bytes)
CREATE src/app/app.worker.ts (157 bytes)
UPDATE tsconfig.app.json (296 bytes)
UPDATE angular.json (3776 bytes)
UPDATE src/app/app.component.ts (605 bytes)
```

Here,

- **app** refers the location of the web worker to be created.
- Angular CLI will generate two new files, **tsconfig.worker.json** and **src/app/app.worker.ts** and update three files, **tsconfig.app.json, angular.json** and **src/app/app.component.ts** file.

Let us check the changes:

```
// tsconfig.worker.json
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "./out-tsc/worker",
    "lib": [
      "es2018",
      "webworker"
    ],
```

```
    "types": []
  },
  "include": [
    "src/**/*.worker.ts"
  ]
}
```

Here,

tsconfig.worker.json extends **tsconfig.json** and includes options to compile web workers.

```
// tsconfig.app.json [only a snippet]
"exclude": [
    "src/test.ts",
    "src/**/*.spec.ts",
    "src/**/*.worker.ts"
]
```

Here,

Basically, it excludes all the worker from compiling as it has separate configuration.

```
// angular.json (only a snippet)
"webWorkerTsConfig": "tsconfig.worker.json"
```

Here,

angular.json includes the web worker configuration file, **tsconfig.worker.json.**

```
// src/app/app.worker.ts
addEventListener('message', ({ data }) => {
  const response = `worker response to ${data}`;
  postMessage(response);
});
```

Here,

A web worker is created. Web worker is basically a function, which will be called when a message event is fired. The web worker will receive the data send by the caller, process it and then send the response back to the caller.

```
// src/app/app.component.ts [only a snippet]
if (typeof Worker !== 'undefined') {
  // Create a new
  const worker = new Worker('./app.worker', { type: 'module' });
  worker.onmessage = ({ data }) => {
    console.log(`page got message: ${data}`);
  };
  worker.postMessage('hello');
} else {
```

```
  // Web Workers are not supported in this environment.
  // You should add a fallback so that your program still executes correctly.
}
```

Here,

- **AppComponent** create a new worker instance, create a callback function to receive the response and then post the message to the worker.

Restart the application. Since the **angular.json** file is changed, which is not watched by Angular runner, it is necessary to restart the application. Otherwise, Angular does not identify the new web worker and does not compile it.

Let us create Typescript class, **src/app/app.prime.ts** to find nth prime numbers.

```
export class PrimeCalculator
{
        static isPrimeNumber(num : number) : boolean {
                if(num == 1) return true;

                let idx : number = 2;
                for(idx = 2; idx < num / 2; idx++)
                {
                        if(num % idx == 0)
                                return false;
                }

                return true;
        }

        static findNthPrimeNumber(num : number) : number {
                let idx : number = 1;
                let count = 0;

                while(count < num) {
                        if(this.isPrimeNumber(idx))
                                count++;

                        idx++;
                        console.log(idx);
                }

                return idx - 1;
        }
}
```

Here,

- **isPrimeNumber** check whether the given number is prime or not.
- **findNthPrimeNumber** finds the nth prime number.

Import the new created prime number class into **src/app/app.worker.ts** and change the logic of the web worker to find nth prime number.

```
/// <reference lib="webworker" />

import { PrimeCalculator } from './app.prime';

addEventListener('message', ({ data }) => {
        // const response = `worker response to ${data}`;
        const response = PrimeCalculator.findNthPrimeNumber(parseInt(data));
        postMessage(response);
});
```

Change **AppComponent** and include two function, **find10thPrimeNumber** and **find10000thPrimeNumber**.

```
import { Component } from '@angular/core';
import { PrimeCalculator } from './app.prime';

@Component({
        selector: 'app-root',
        templateUrl: './app.component.html',
        styleUrls: ['./app.component.css']
})
export class AppComponent {
        title = 'Web worker sample';
        prime10 : number = 0;
        prime10000 : number = 0;

        find10thPrimeNumber() {
                this.prime10 = PrimeCalculator.findNthPrimeNumber(10);
        }

        find10000thPrimeNumber() {
                if (typeof Worker !== 'undefined') {
                        // Create a new
                        const worker = new Worker('./app.worker', { type:
'module' });
                        worker.onmessage = ({ data }) => {
                        this.prime10000 = data;
                        };
                        worker.postMessage(10000);
                } else {
                        // Web Workers are not supported in this environment.
                        // You should add a fallback so that your program still
executes correctly.
                }
        }
}
```

Here,

find10thPrimeNumber is directly using the PrimeCalculator. But, **find10000thPrimeNumber** is delegating the calculation to web worker, which in turn uses PrimeCalculator.

Change the AppComponent template, src/app/app.commands.html and include two option, one to find 10th prime number and another to find the 10000th prime number.

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

<div>
        <a href="#" (click)="find10thPrimeNumber()">Click here</a> to find 10th
prime number
        <div>The 10<sup>th</sup> prime number is {{ prime10 }}</div> <br/>
        <a href="#" (click)="find10000thPrimeNumber()">Click here</a> to find
10000th prime number
        <div>The 10000<sup>th</sup> prime number is {{ prime10000 }}</div>
</div>
```

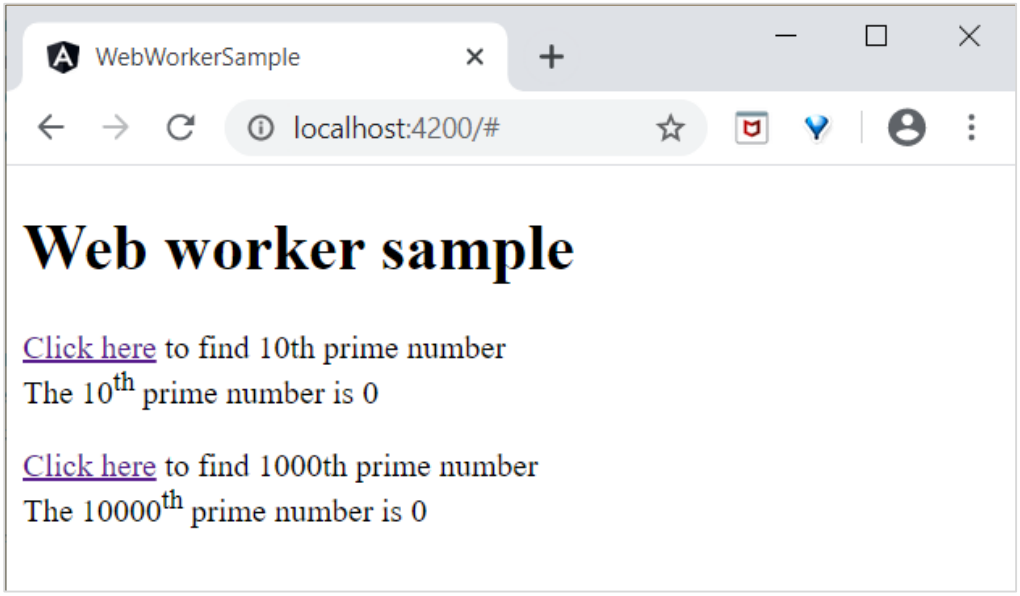
Here,

Finding 10000th prime number will take few seconds, but it will not affect other process as it is uses web workers. Just try to find the 10000th prime number first and then, the 10th prime number.

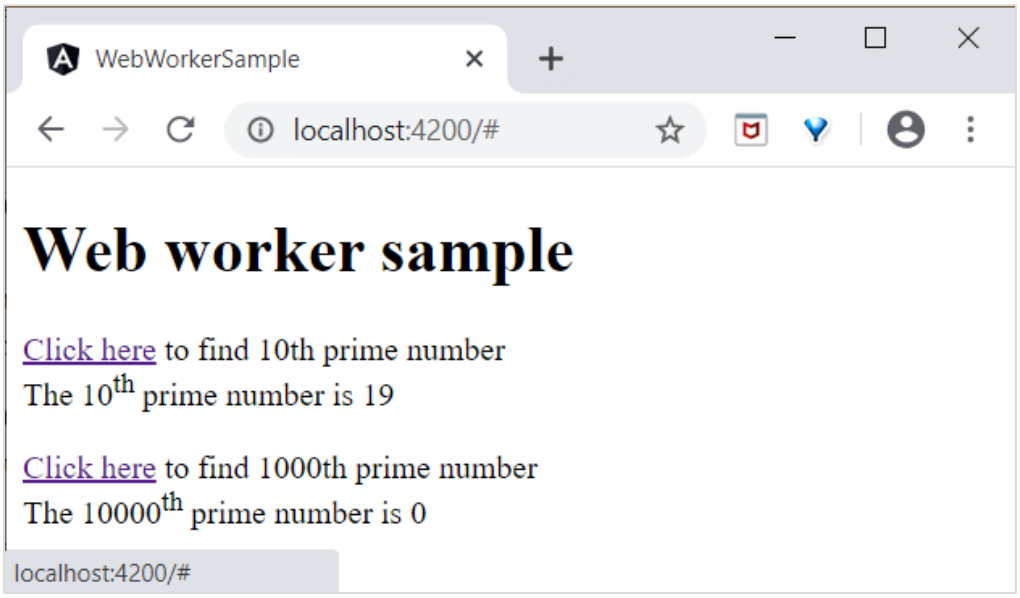
Since, the web worker is calculating 10000th prime number, the UI does not freeze. We can check 10th prime number in the meantime. If we have not used web worker, we could not do anything in the browser as it is actively processing the 10000th prime number.

The result of the application is as follows:

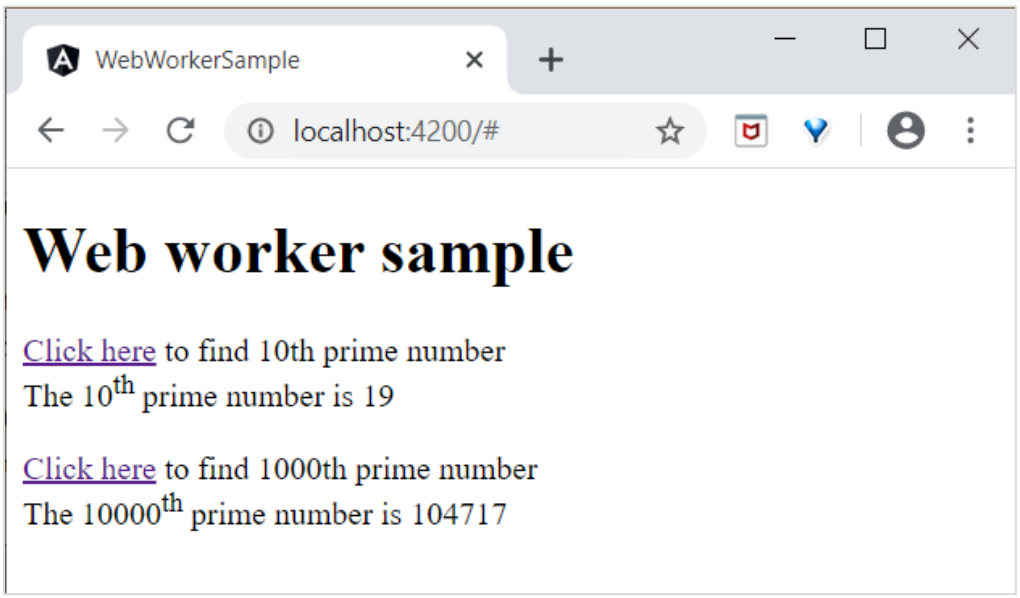
Initial state of the application.



Click and try to find the 10000th prime number and then try to find the 10th prime number. The application finds the 10th prime number quite fast and shows it. The application is still processing in the background to find the 10000th prime number.



Both processes are completed.



Web worker enhances the user experience of web application by doing the complex operation in the background and it is quite easy to do it in Angular Application as well.

19. Angular 8 — Service Workers and PWA

Progressive web apps (PWA) are normal web application with few enhancements and behaves like a native application. PWA apps does not depends on network to work. PWA caches the application and renders it from local cache. It regularly checks the live version of the application and then caches the latest version in the background.

PWA can be installed in the system like native application and shortcut can be shown in the desktop. Clicking the shortcut will open the application in browser with local cache even without any network available in the system.

Angular application can be converted into PWA application. To convert an Angular application, we need to use service worker API. Service worker is actually a proxy server, which sits in between the browser, application and the network.

Service workers is separate from web pages. It does not able to access DOM objects. Instead, Service Workers interact with web pages through **PostMessage** interface.

PWA application has two prerequisites. They are as follows,

- **Browser support** - Even though lot of browser supports the PWA app, IE, Opera mini and few other does not provides the PWA support.
- **HTTPS delivery** - The application needs to be delivered through HTTPS protocol. One exception of the https support is **localhost** for development purpose.

Let us create a new application and convert it into PWA application.

Create a new Angular application using below command:

```
cd /go/to/workspace
ng new pwa-sample
```

Add PWA support using below command:

```
cd pwa-sample
ng add @angular/pwa --project pwa-sample
```

Build the production version of the application,

```
ng build --prod
```

PWA application does not run under Angular development server. Install, a simple web server using below command:

```
npm install -g http-server
```

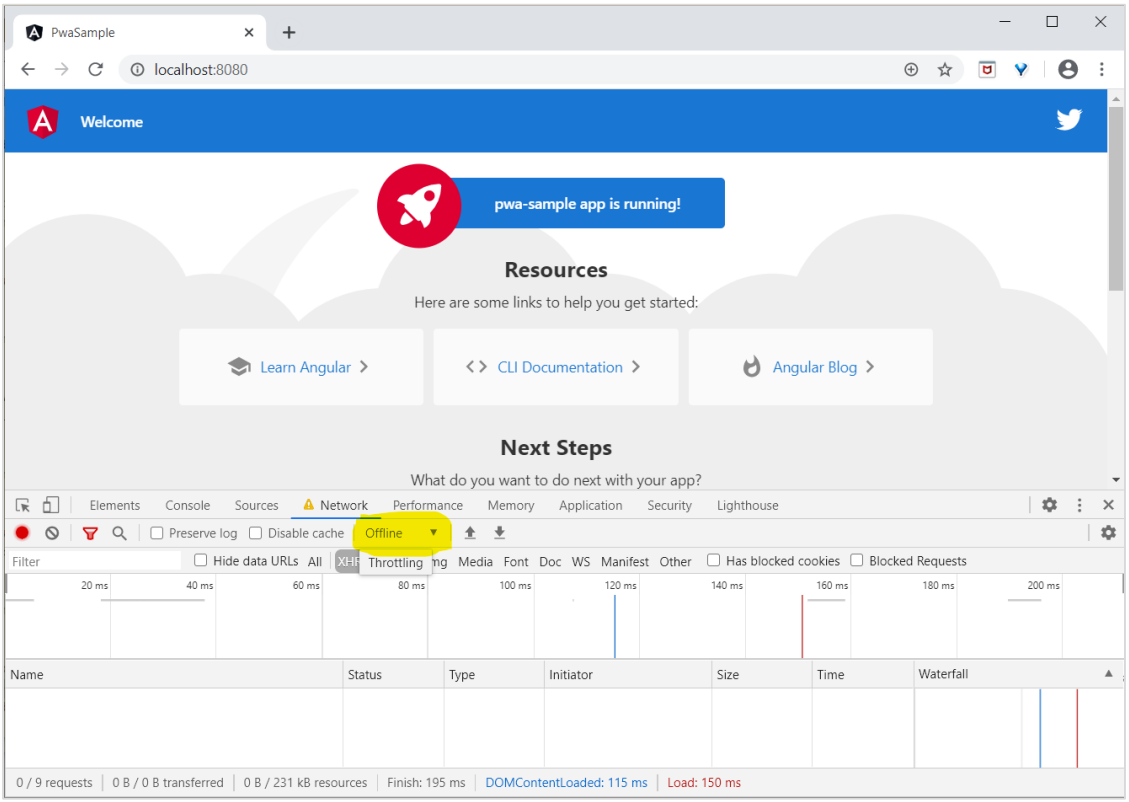
Run the web server and set our production build of the application as root folder.

```
http-server -p 8080 -c-1 dist/pwa-sample
```

Open browser and enter http://localhost:8080.

Now, go to **Developer tools -> Network** and select **Offline** option.

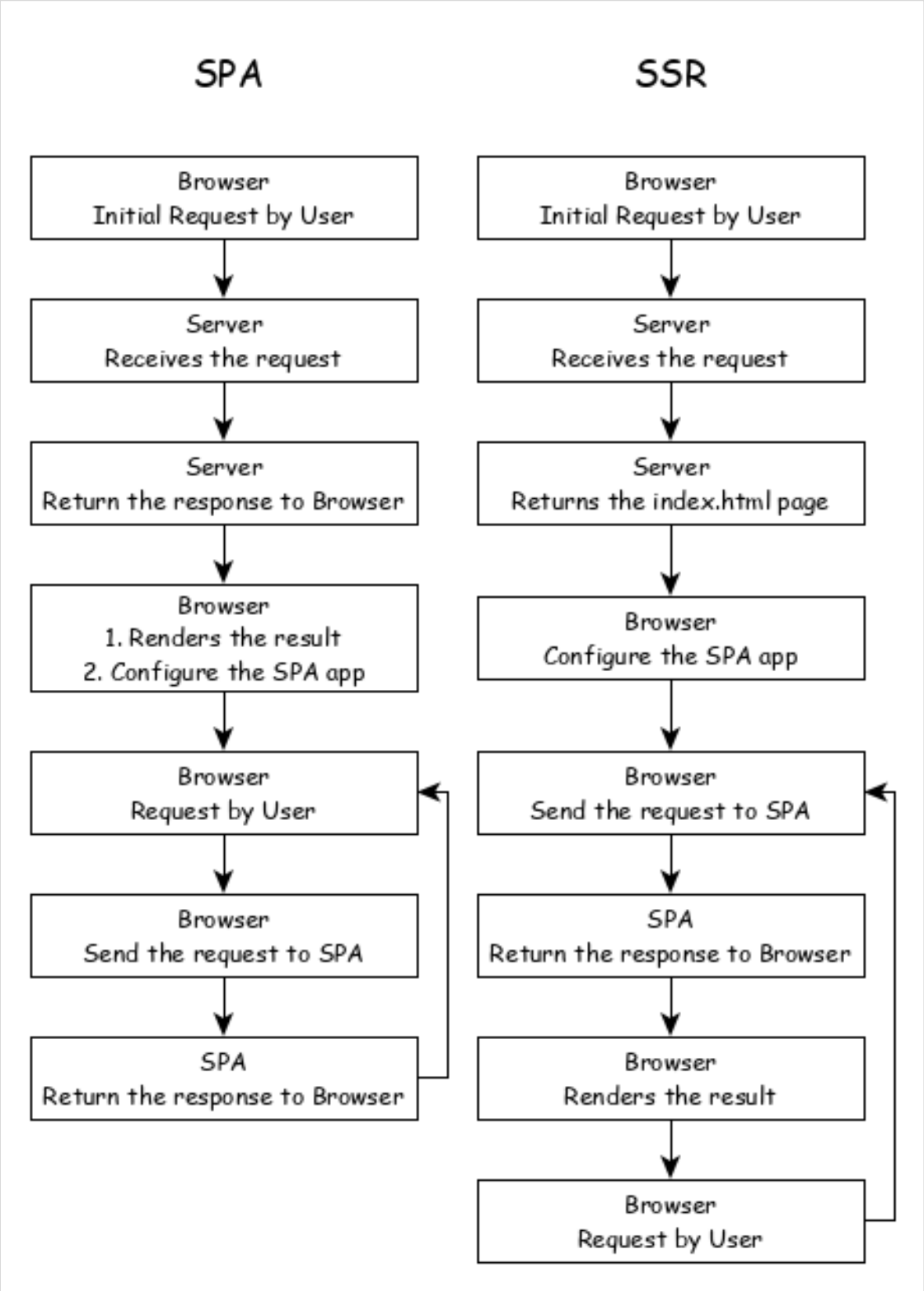
Normal application stops working if network is set to Offline but, PWA application works fine as shown below:



20. Angular 8 — Server Side Rendering

Server side Rendering (SSR) is a modern technique to convert a Single Page Application (SPA) running in the browser into a server based application. Usually, in SPA, the server returns a simple **index.html** file with the reference to the JavaScript based SPA app. The SPA app take over from there, configure the entire application, process the request and then send the final response.

But in SSR supported application, the server as well do all the necessary configuration and then send the final response to the browser. The browser renders the response and start the SPA app. SPA app takeover from there and further request are diverted to SPA app. The flow of SPA and SSR is as shown in below diagram.



Converting a SPA application to SSR provides certain advantages and they are as follows:

- **Speed** - First request is relatively fast. One of the main drawback of SPA is slow initial rendering. Once the application is rendered, SPA app is quite fast. SSR fixes the initial rendering issue.

- **SEO Friendly** - Enables the site to be SEO friendly. Another main disadvantage of SPA is not able to crawled by web crawler for the purpose of SEO. SSR fixes the issue.

Angular Universal

To enable SSR in Angular, Angular should be able to rendered in the server. To make it happen, Angular provides a special technology called **Angular Universal**. It is quite new technology and it is continuously evolving. Angular Universal knows how to render Angular application in the server. We can upgrade our application to **Angular Universal** to support SSR.

21. Angular 8 — Internationalization (i18n)

Internationalization (i18n) is a must required feature for any modern web application. Internationalization enables the application to target any language in the world. Localization is a part of the Internationalization and it enables the application to render in a targeted local language. Angular provides complete support for internationalization and localization feature.

Let us learn how to create a simple hello world application in different language.

Create a new Angular application using below command:

```
cd /go/to/workspace
ng new i18n-sample
```

Run the application using below command:

```
cd i18n-sample
npm run start
```

Change the **AppComponent's** template as specified below:

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

<div>Hello</div>
<div>The Current time is {{ currentDate | date : 'medium' }}</div>
```

Add **localize** module using below command:

```
ng add @angular/localize
```

Restart the application.

LOCALE_ID is the Angular variable to refer the current locale. By default, it is set as en_US. Let us change the locale by using in the provider in AppModule.

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

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [ { provide: LOCALE_ID, useValue: 'hi' } ],
```

```
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Here,

- **LOCALE_ID** is imported from **@angular/core**.
- LOCALE_ID is set to hi through provider so that, the LOCALE_ID will be available everywhere in the application.

Import the locale data from **@angular/common/locales/hi** and then, register it using registerLocaleData method as specified below:

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

import { registerLocaleData } from '@angular/common';
import localeHi from '@angular/common/locales/hi';

registerLocaleData(localeHi);

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = 'Internationzation Sample';
}
```

Create a local variable, **CurrentDate** and set current time using **Date.now().**

```
export class AppComponent {
  title = 'Internationzation Sample';

  currentDate: number = Date.now();
}
```

Change **AppComponent's** template content and include the **currentDate** as specified below:

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

<div>Hello</div>
<div>The Current time is {{ currentDate | date : 'medium' }}</div>
```

Check the result and you will see the date is specified using **hi** locale.

We have changed the date to current locale. Let us change other content as well. To do it, include **i18n** attribute in the relevant tag with format, **title|description@@id.**

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

<h1 i18n="greeting|Greeting a person@@greeting">Hello</h1>
<div>
        <span i18n="time|Specifiy the current time@@currentTime">
                The Current time is {{ currentDate | date : 'medium' }}
        </span>
</div>
```

Here,

- **hello** is simple translation format since it contains complete text to be translated.
- **Time** is little bit complex as it contains dynamic content as well. The format of the text should follow ICU message format for translation.

We can extract the data to be translated using below command:

```
ng xi18n --output-path src/locale
```

Command generates **messages.xlf** file with below content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="greeting" datatype="html">
        <source>Hello</source>
        <context-group purpose="location">
          <context context-
type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">3</context>
        </context-group>
        <note priority="1" from="description">Greeting a person</note>
        <note priority="1" from="meaning">greeting</note>
```

```
      </trans-unit>
      <trans-unit id="currentTime" datatype="html">
        <source>
                The Current time is <x id="INTERPOLATION" equiv-text="{{
currentDate | date : &apos;medium&apos; }}"/>
        </source>
        <context-group purpose="location">
          <context context-
type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">5</context>
        </context-group>
        <note priority="1" from="description">Specifiy the current time</note>
        <note priority="1" from="meaning">time</note>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Copy the file and rename it to **messages.hi.xlf**

Open the file with Unicode text editor. Locate **source** tag and duplicate it with **target** tag and then change the content to **hi** locale. Use google translator to find the matching text. The changed content is as follows:

```
<source>Hello</source>
<target>हेलो</target>
```

```
<source>
      The Current time is <x id="INTERPOLATION" equiv-text="{{ currentDate | date : &apos;medium&apos; }}"/>
</source>
<target>
      वर्तमान तिथि और समय <x id="INTERPOLATION" equiv-text="{{ currentDate | date : &apos;medium&apos; }}"/> है
</target>
```

Open **angular.json** and place below configuration under **build -> configuration**.

```
"hi": {
  "aot": true,
  "outputPath": "dist/hi/",
  "i18nFile": "src/locale/messages.hi.xlf",
  "i18nFormat": "xlf",
  "i18nLocale": "hi",
  "i18nMissingTranslation": "error",
  "baseHref": "/hi/"
},
"en": {
  "aot": true,
  "outputPath": "dist/en/",
  "i18nFile": "src/locale/messages.xlf",
  "i18nFormat": "xlf",
  "i18nLocale": "en",
  "i18nMissingTranslation": "error",
  "baseHref": "/en/"
}
```

Here,

We have used separate setting for **hi** and **en** locale.

Set below content under **serve -> configuration.**

```
"hi": {
  "browserTarget": "i18n-sample:build:hi"
},
"en": {
  "browserTarget": "i18n-sample:build:en"
}
```

We have added the necessary configuration. Stop the application and run below command:

```
npm run start -- --configuration=hi
```

Here,

We have specified that the *hi* configuration has to be used.

Navigate to **http://localhost:4200/hi** and you will see the Hindi localised content.

Finally, we have created a localized application in Angular.

22. Angular 8 — Accessibility

Accessibility support is one of the important feature of every UI based application. Accessibility is a way of designing the application so that, it is accessible for those having certain disabilities as well. Let us learn the support provided by Angular to develop application with good accessibility.

- While using attribute binding, use **attr.** prefix for ARIA attributes.
- Use Angular material component for Accessibility. Some of the useful components are **LiveAnnouncer** and **cdkTrapFocus**
- Use native HTML elements wherever possible because native HTML element provides maximum accessibility features. When creating a component, select native html element matching your use case instead of redeveloping the native functionality.
- Use **NavigationEnd** to track and control the focus of the application as it greatly helps in accessibility.

23. Angular 8 — CLI Commands

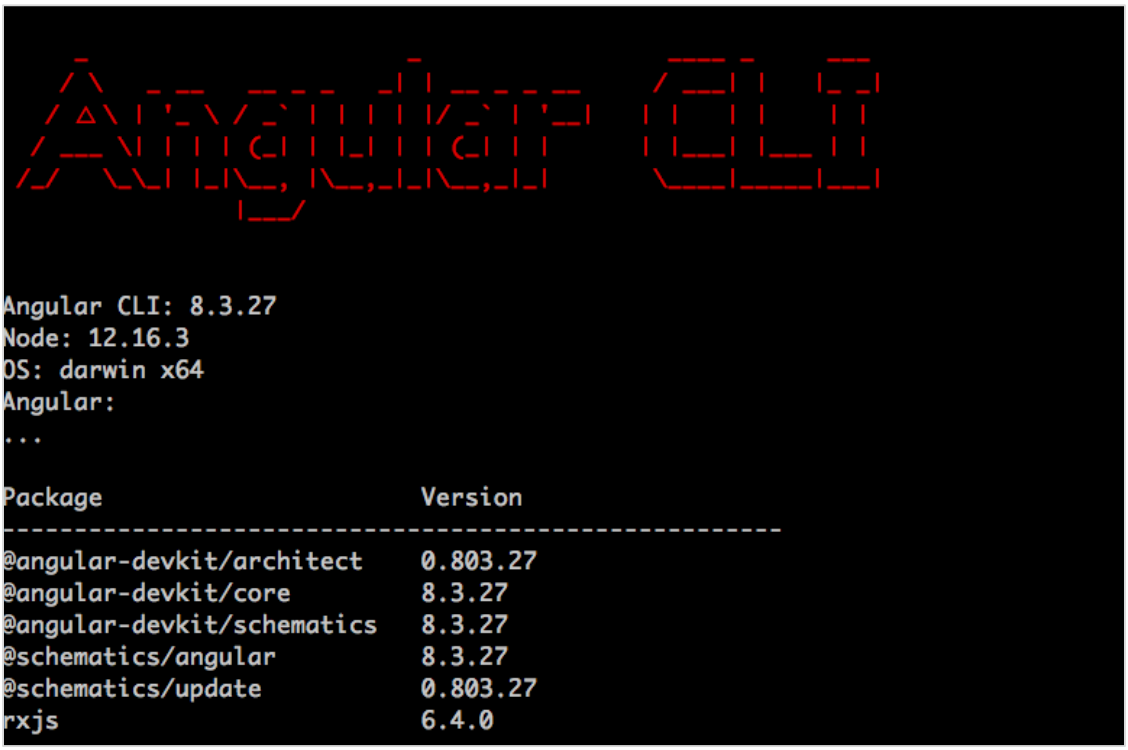
Angular CLI helps developers to create projects easily and quickly. As we know already, Angular CLI tool is used for development and built on top of Node.js, installed from NPM.This chapter explains about Angular 8 CLI commands in detail.

Verify CLI

Before moving to Angular CLI commands, we have to ensure that Angular CLI is installed on your machine. If it is installed, you can verify it by using the below command:

```
ng version
```

You could see the below response:



If CLI is not installed, then use the below command to install it.

```
npm install -g @angular/cli@^8.0.0
```

Let's understand the commands one by one in brief.

New command

To create an application in Angular, use the below syntax:

```
ng new <project-name>
```

Example

If you want to create **CustomerApp** then, use the below code:

```
ng new CustomerApp
```

Generate Command

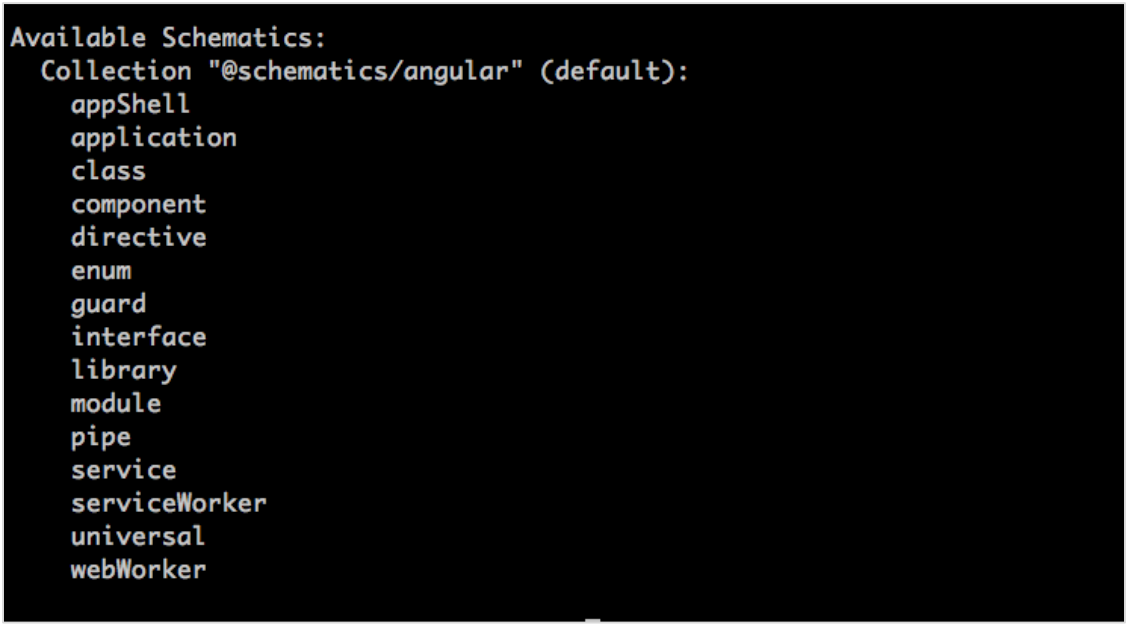
It is used to generate or modify files based on a schematic. Type the below command inside your angular project:

```
ng generate
```

Or, you can simply type generate as g. You can also use the below syntax:

```
ng g
```

It will list out the available schematics:



Let's understand some of the repeatedly used **ng generate** schematics in next section.

Create a component

Components are building block of Angular. To create a component in angular use the below syntax:

```
ng g c <component-name>
```

For example, if user wants to create a **Details** component then use the below code:

```
ng g c Details
```

After using this command, you could see the below response:

```
CREATE src/app/details/details.component.scss (0 bytes)
CREATE src/app/details/details.component.html (22 bytes)
CREATE src/app/details/details.component.spec.ts (635 bytes)
CREATE src/app/details/details.component.ts (274 bytes)
UPDATE src/app/app.module.ts (1201 bytes)
```

Create a class

It is used to create a new class in Angular. It is defined below:

```
ng g class <class-name>
```

If you want to create a customer class, then type the below command:

```
ng g class Customer
```

After using this command, you could see the below response:

```
CREATE src/app/customer.spec.ts (162 bytes)
CREATE src/app/customer.ts (26 bytes)
```

Create a pipe

Pipes are used for filtering the data. It is used to create a custom pipe in Angular. It is defined below:

```
ng g pipe <pipe-name>
```

If you want to create a custom digit counts in a pipe, then type the below command:

```
ng g pipe DigitCount
```

After using this command, you could see the below response:

```
CREATE src/app/digit-count.pipe.spec.ts (204 bytes)
CREATE src/app/digit-count.pipe.ts (213 bytes)
UPDATE src/app/app.module.ts (1274 bytes)
```

Create a directive

It is used to create a new directive in Angular. It is defined below:

```
ng g directive <directive-name>
```

If you want to create a UnderlineText directive, then type the below command:

```
ng g directive UnderlineText
```

After using this command, you could see the below response:

```
CREATE src/app/underline-text.directive.spec.ts (253 bytes)
CREATE src/app/underline-text.directive.ts (155 bytes)
UPDATE src/app/app.module.ts (1371 bytes)
```

Create a module

It is used to create a new module in Angular. It is defined below:

```
ng g module <module-name>
```

If you want to create a user information module, then type the below command:

```
ng g module Userinfo
```

After using this command, you could see the below response:

```
CREATE src/app/userinfo/userinfo.module.ts (194 bytes)
```

Create an interface

It is used to create an interface in Angular. It is given below:

```
ng g interface <interface-name>
```

If you want to create a customer class, then type the below command:

```
ng g interface CustomerData
```

After using this command, you could see the below response:

```
CREATE src/app/customer-data.ts (34 bytes)
```

Create a web worker

It is used to create a new web worker in Angular. It is stated below:

```
ng g webWorker <webWorker-name>
```

If you want to create a customer class, then type the below command:

```
ng g webWorker CustomerWebWorker
```

After using this command, you could see the below response:

```
CREATE tsconfig.worker.json (212 bytes)
CREATE src/app/customer-web-worker.worker.ts (157 bytes)
UPDATE tsconfig.app.json (296 bytes)
UPDATE angular.json (3863 bytes)
```

Create a service

It is used to create a service in Angular. It is given below:

```
ng g service <service-name>
```

If you want to create a customer class, then type the below command:

```
ng g service CustomerService
```

After using this command, you could see the below response:

```
CREATE src/app/customer-service.service.spec.ts (379 bytes)
CREATE src/app/customer-service.service.ts (144 bytes)
```

Create an enum

It is used to create an enum in Angular. It is given below:

```
ng g enum <enum-name>
```

If you want to create a customer class, then type the below command:

```
ng g enum CustomerRecords
```

After using this command, you could see the below response:

```
CREATE src/app/customer-records.enum.ts (32 bytes)
```

Add command

It is used to add support for an external library to your project. It is specified by the below command:

```
ng add [name]
```

Build command

It is used to compile or build your angular app. It is defined below:

```
ng build
```

After using this command, you could see the below response:

```
Generating ES5 bundles for differential loading...
ES5 bundle generation complete.
```

Config command

It is used to retrieve or set Angular configuration values in the angular.json file for the workspace. It is defined below:

```
ng config
```

After using this command, you could see the below response:

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "MyApp": {
      "projectType": "application",
      "schematics": {
        "@schematics/angular:component": {
          "style": "scss"
        }
      },
 ..........................
 ..........................
```

Doc command

It is used to open the official Angular documentation (angular.io) in a browser, and searches for a given keyword.

```
ng doc <keyword>
```

For example, if you search with **component** as **ng g component** then, it will open the documentation.

e2e command

It is used to build and serves an Angular app, then runs end-to-end tests using Protractor. It is stated below:

```
ng e2e <project> [options]
```

Help command

It lists out available commands and their short descriptions. It is stated below:

```
ng help
```

Serve command

It is used to build and serves your app, rebuilding on file changes. It is given below:

```
ng serve
```

Test command

Runs unit tests in a project. It is mentioned below:

```
ng test
```

Update command

Updates your application and its dependencies. It is given below:

```
ng update
```

Version command

Shows Angular CLI version. It is stated below:

```
ng version
```

24. Angular 8 — Testing

Testing is a very important phase in the development life cycle of an application. It ensures an application quality. It needs careful planning and execution.

Unit Test

Unit testing is the easiest method to test an application. It is based on ensuring the correctness of a piece of code or a method of a class. But, it does not reflect the real environment and subsequently. It is the least option to find the bugs.

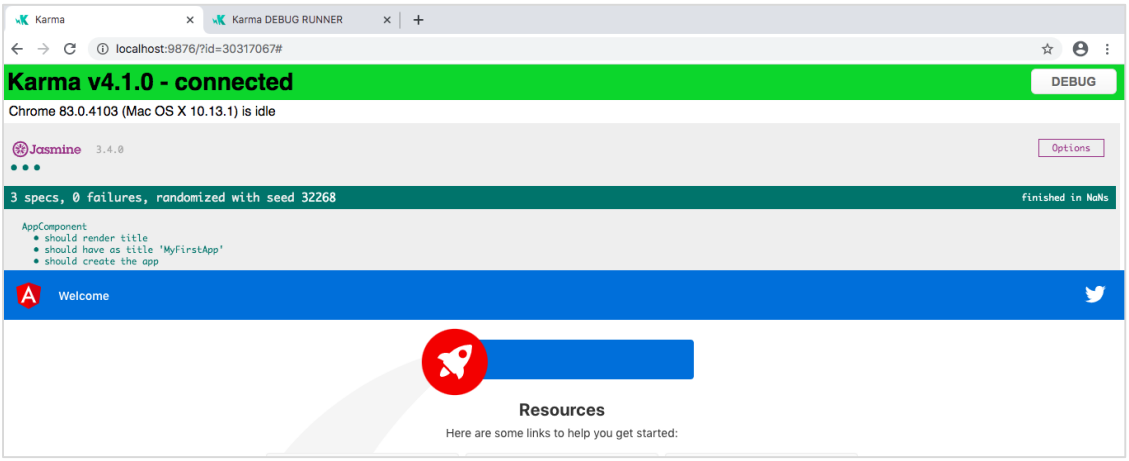
Generally, Angular 8 uses Jasmine and Karma configurations. To perform this, first you need to configure in your project, using the below command:

```
ng test
```

Now, you could see the following response:

```
10% building 2/2 modules 0 active30 06 2020 01:46:39.982:WARN [karma]: No captured browser, open http:/
/localhost:9876/
30 06 2020 01:46:40.121:INFO [karma-server]: Karma v4.1.0 server started at http://0.0.0.0:9876/
30 06 2020 01:46:40.121:INFO [launcher]: Launching browsers Chrome with concurrency unlimited
30 06 2020 01:46:40.169:INFO [launcher]: Starting browser Chrome
30 06 2020 01:46:48.883:WARN [karma]: No captured browser, open http://localhost:9876/
30 06 2020 01:46:49.092:INFO [Chrome 83.0.4103 (Mac OS X 10.13.1)]: Connected on socket Syipx6txxp9ghkj
WAAAA with id 30317067
Chrome 83.0.4103 (Mac OS X 10.13.1): Executed 1 of 3 SUCCESS (0 secs / 0.243 secChrome 83.0.4103 (Mac O
S X 10.13.1): Executed 2 of 3 SUCCESS (0 secs / 0.463 secChrome 83.0.4103 (Mac OS X 10.13.1): Executed
3 of 3 SUCCESS (0 secs / 0.571 secChrome 83.0.4103 (Mac OS X 10.13.1): Executed 3 of 3 SUCCESS (0.685 s
ecs / 0.571 secs)
TOTAL: 3 SUCCESS
TOTAL: 3 SUCCESS
```

Now, Chrome browser also opens and shows the test output in the "Jasmine HTML Reporter". It looks similar to this,

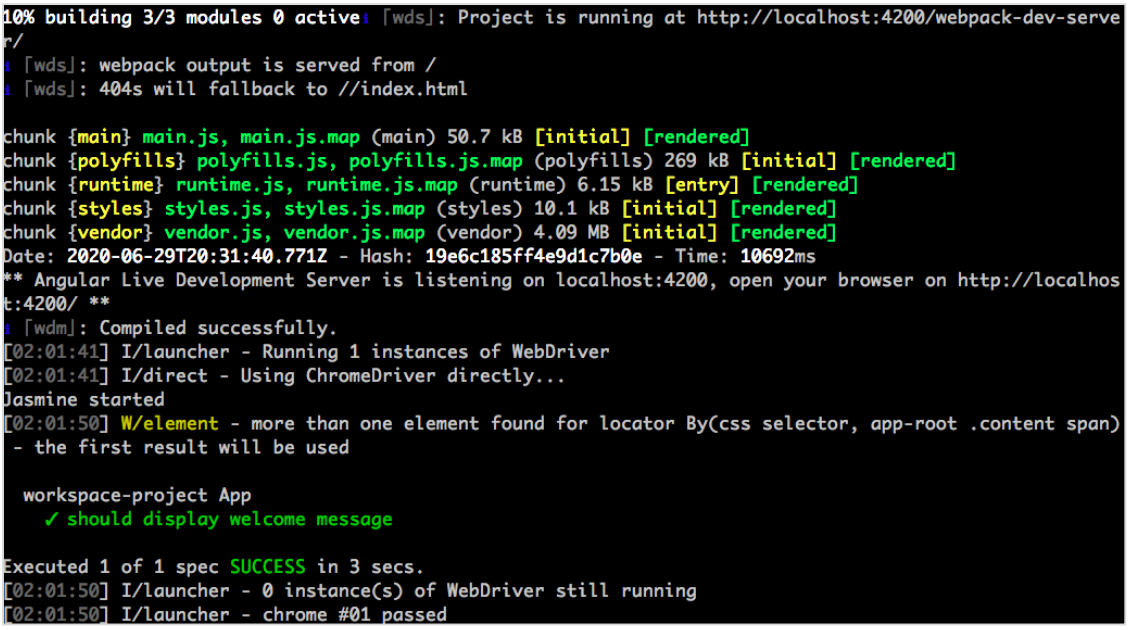


End to End (E2E) Testing

Unit tests are small, simple and fast process whereas, E2E testing phase multiple components are involved and works together which cover flows in the application. To perform e2e test, type the below command:

```
ng e2e
```

You could see the below response:



25. Angular 8 — Ivy Compiler

Ivy Compiler is the latest compiler for Angular application released by Angular Team. Currently, Angular is using **View Engine** compiler to compile Angular application.

In general, Angular compiler has two options to compile an application.

Just In Time (JIT) Compiler

In **Just In Time (JIT)** compilation, the compiler will be bundled along with the application and send to the browser. Angular application will be compiled in the browser and run just before the execution of application.

Eventhough **JIT** provides certain advanced feature, **JIT** slows down the compilation and also the app bundle will be double the size produced by **AOT** compiler as it includes compiler as well.

Ahead Of Time (AOT) Compiler

In **AOT** compilation, the compiler will emit optimised code ready to run inside the browser without any addition step. It will reduce the size of the bundle as well as reduce the compilation time and startup time of the application.

Advantages of Ivy Compiler

Ivy Compiler is the optimised and advanced compiler for Angular. As of Angular 8, it is not yet complete even though it is useable at this stage. Angular Team is recommending the developer to use it in Angular 8.

The main advantages of **Ivy Compiler** are as follows:

- Optimised code.
- Faster build time.
- Reduced bundle size.
- Better performance.

How to use Ivy?

Ivy Compiler can be used in Angular 8 application by changing the project setting as specified below:

Open angular.json and set the aot option (**projects -> -> architect -> build -> configurations -> production**) of the project to true.

```
{
  "projects": {
    "my-existing-project": {
      "architect": {
```

```
        "build": {
          "options": {
            ...
            "aot": true,
          }
        }
      }
    }
  }
}
```

Open **tsconfig.app.json** and set **enableIvy** to true under **angularCompilerOptions**.

```
{
    ...
    "angularCompilerOptions": {
        "enableIvy": true
}
```

Compile and run the application and get benefited by **Ivy Compiler**.

26. Angular 8 — Building with Bazel

Bazel is an advanced build and test tool. It supports lot of features suitable for large projects.

Some of the features of **Bazel** are as follows:

- Support multiple languages.
- Support multiple platforms.
- Support multiple repository.
- Support high-level build language.
- Fast and reliable.

Angular supports building the application using bazel. Let us see how to use bazel to compile Angular application.

First, install **@angular/bazel** package.

```
npm install -g @angular/bazel
```

For existing application, Add **@angular/bazel** as mentioned below:

```
ng add @angular/bazel
```

For new application, use below mentioned command:

```
ng new --collection=@angular/bazel
```

To build an application using bazel, use below command:

```
ng build --leaveBazelFilesOnDisk
```

Here,

leaveBazelFilesOnDisk option will leave the bazel files created during build process, which we can use to build the application directly using bazel.

To build application using bazel directly, install **@bazel/bazelisk** and then, use **bazelisk build** command.

```
npm install -g @bazel/bazelisk
bazelisk build
```

27. Angular 8 — Backward Compatibility

Angular framework provides maximum compatibility with previous versions. If Angular Team deprecate a feature in a release, it will wait for 3 more release to completely remove the feature. Angular Team release a major version for every six months. Every version will have active maintenance period of six months and then Long Term Support (LTS) period for another one year. Angular does not introduce breaking changes during these 18 months. If Angular version deprecate a feature in version 5, then it will probably remove it in version 8 or in next releases.

Angular maintains documentation and guides of all version. For example, Angular documentation for version 7 can be checked @ https://v7.angular.io. Angular also provides a detailed upgrade path through https://update.angular.io/ site.

To update Angular application written from previous version, use below command inside the project directory:

```
ng update @angular/cli@8 @angular/core@8
```

Let us see some of the important changes introduced in Angular 8.

- **HttpModule** module and its associated **Http** service is removed. Use **HttpClient** service from **HttpClientModule** module.
- **/deep/, >>>** and **:ng-deep** component selectors are removed.
- Angular default version of TypeScript is 3.4.
- Node version supported by Angular is v10 and later.
- **@ViewChild()** and **ContentChild()** decorator behaviour is changed from dynaic to static.

Lazy loading string syntax in router module is removed and only function based is supported.

```
loadChildren: './lazy/lazy.module#LazyModule'
loadChildren: () => import('./lazy/lazy.module'
```

28. Angular 8 — Working Example

Here, we will study about the complete step by step working example with regards to Angular 8.

Let us create an Angular application to check our day to day expenses. Let us give **ExpenseManager** as our choice for our new application.

Create an application

Use below command to create the new application.

```
cd /path/to/workspace
ng new expense-manager
```

Here,

new is one of the command of the ng CLI application. It will be used to create new application. It will ask some basic question in order to create new application. It is enough to let the application choose the default choices. Regarding routing question as mentioned below, specify **No.**

```
Would you like to add Angular routing? No
```

Once the basic questions are answered, the **ng CLI application** create a new Angular application under **expense-manager** folder.

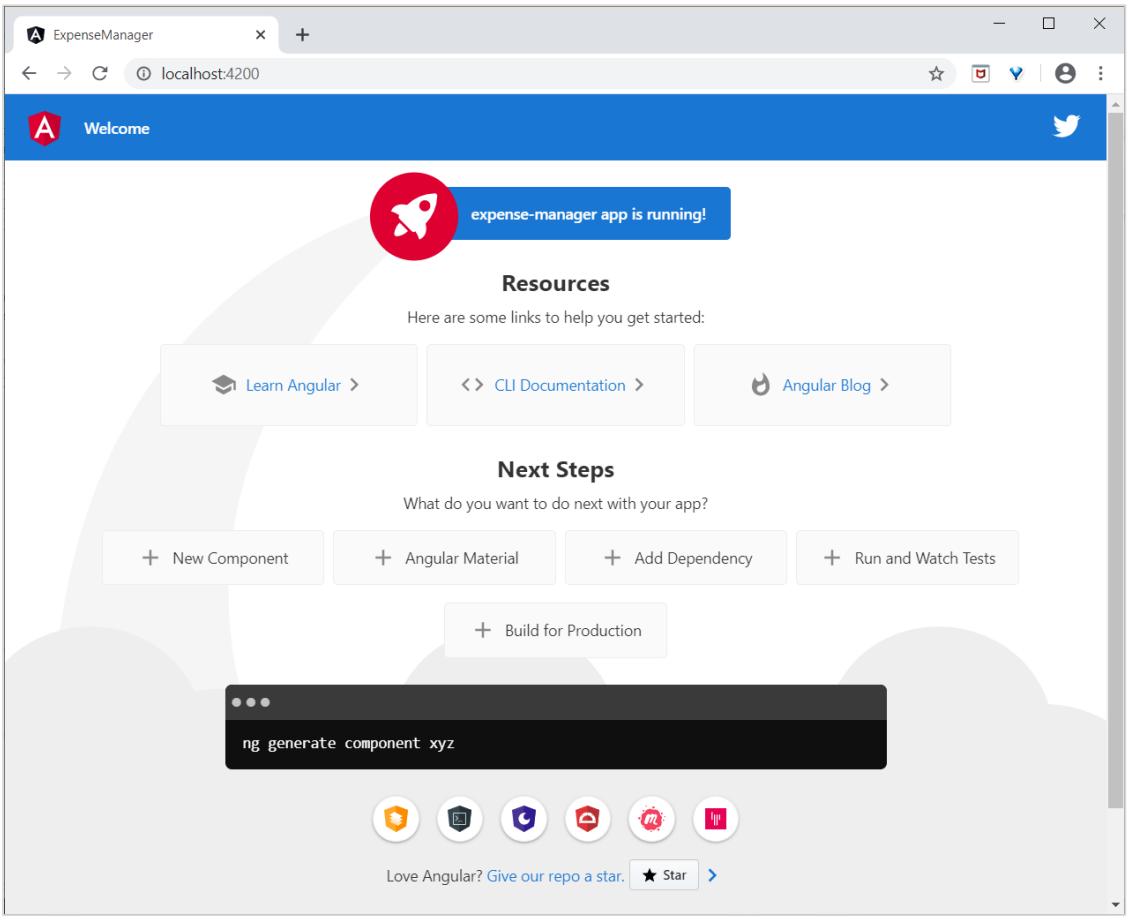
Let us move into the our newly created application folder.

```
cd expense-manager
```

Let us start the application using below command:

```
ng serve
```

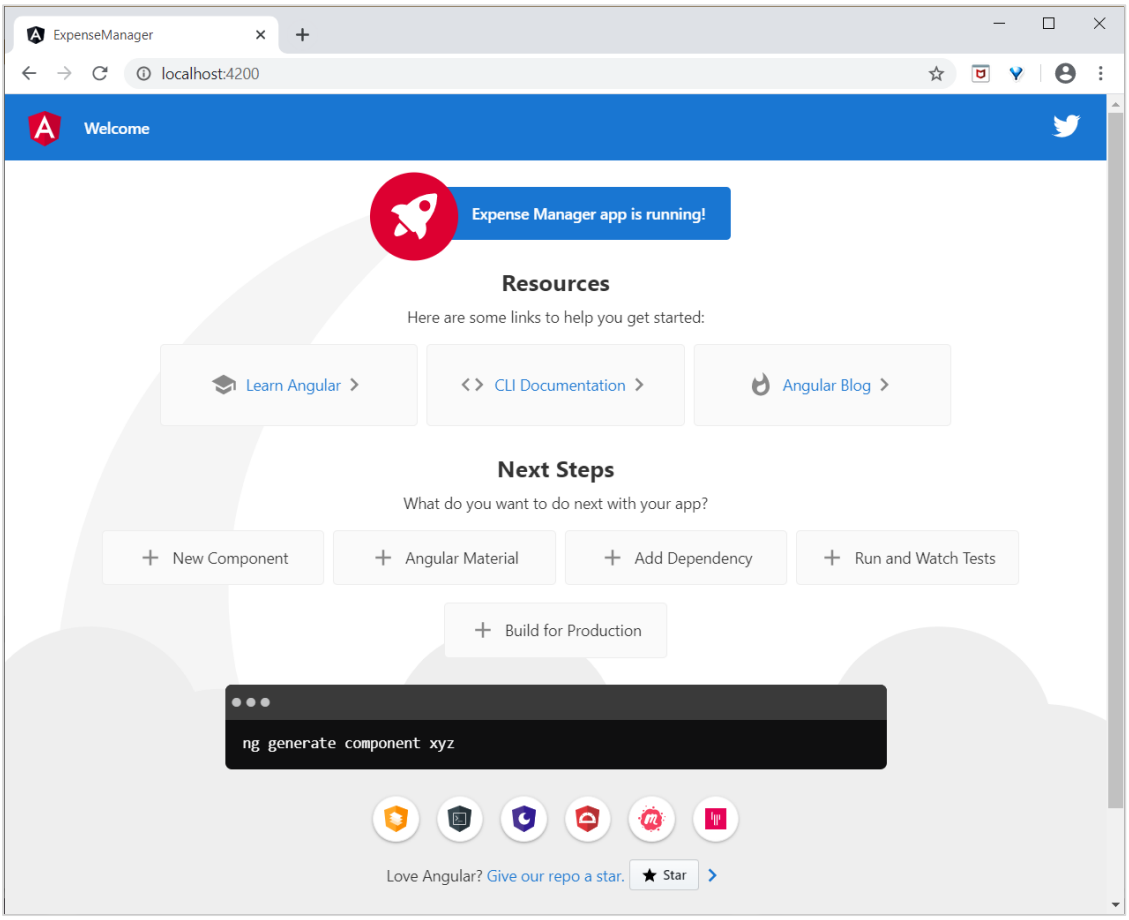
Let us fire up a browser and opens **http://localhost:4200**. The browser will show the application as shown below:



Let us change the title of the application to better reflect our application. Open **src/app/app.component.ts** and change the code as specified below:

```
export class AppComponent {
  title = 'Expense Manager';
}
```

Our final application will be rendered in the browser as shown below:



Add a component

Create a new component using **ng generate component** command as specified below:

```
ng generate component expense-entry
```

Output

The output is as follows:

```
CREATE src/app/expense-entry/expense-entry.component.html (28 bytes)
CREATE src/app/expense-entry/expense-entry.component.spec.ts (671 bytes)
CREATE src/app/expense-entry/expense-entry.component.ts (296 bytes)
CREATE src/app/expense-entry/expense-entry.component.css (0 bytes)
UPDATE src/app/app.module.ts (431 bytes)
```

Here,

- ExpenseEntryComponent is created under src/app/expense-entry folder.
- Component class, Template and stylesheet are created.
- AppModule is updated with new component.

Add title property to ExpenseEntryComponent (**src/app/expense-entry/expense-entry.component.ts**) component.

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

@Component({
        selector: 'app-expense-entry',
        templateUrl: './expense-entry.component.html',
        styleUrls: ['./expense-entry.component.css']
})
export class ExpenseEntryComponent implements OnInit {
        title: string;
        constructor() { }

        ngOnInit() {
                this.title = "Expense Entry"
        }
}
```

Update template, **src/app/expense-entry/expense-entry.component.html** with below content.

```
<p>{{ title }}</p>
```

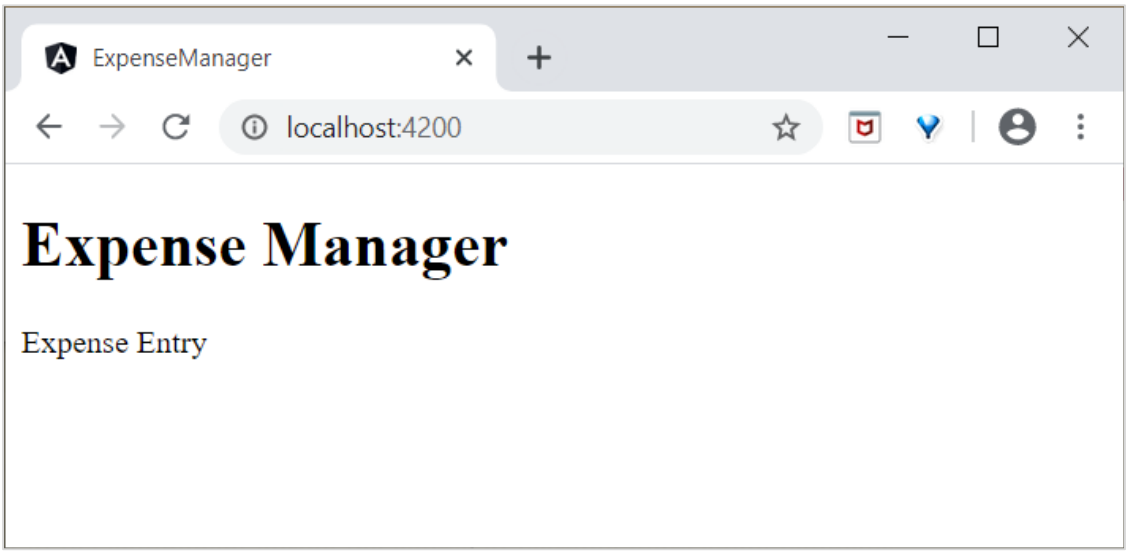
Open **src/app/app.component.html** and include newly created component.

```
<h1>{{ title }}</h1>
<app-expense-entry></app-expense-entry>
```

Here,

app-expense-entry is the selector value and it can be used as regular HTML Tag.

The output of the application is as shown below:



Include bootstrap

Let us include bootstrap into our **ExpenseManager** application using **styles** option and change the default template to use bootstrap components.

Open command prompt and go to **ExpenseManager** application.

```
cd /go/to/expense-manager
```

Install **bootstrap** and **JQuery** library using below commands:

```
npm install --save bootstrap@4.5.0 jquery@3.5.1
```

Here,

We have installed **JQuery** because, bootstrap uses jquery extensively for advanced components.

Option **angular.json** and set bootstrap and jquery library path.

```
{
  "projects": {
    "expense-manager": {
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/expense-manager",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "aot": false,
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
            "styles": [
              "./node_modules/bootstrap/dist/css/bootstrap.css",
              "src/styles.css"
            ],
            "scripts": [
              "./node_modules/jquery/dist/jquery.js",
              "./node_modules/bootstrap/dist/js/bootstrap.js"
            ]
          },
        },
      }
    }},
  "defaultProject": "expense-manager"
}
```

Here,

- **scripts** option is used to include JavaScript library. JavaScript registered through scripts will be available to all Angular components in the application.

Open **app.component.html** and change the content as specified below:

```
<!-- Navigation -->
<nav class="navbar navbar-expand-lg navbar-dark bg-dark static-top">
    <div class="container">
        <a class="navbar-brand" href="#">{{ title }}</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarResponsive" aria-controls="navbarResponsive" aria-
expanded="false" aria-label="Toggle navigation">
                <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarResponsive">
                <ul class="navbar-nav ml-auto">
                        <li class="nav-item active">
                        <a class="nav-link" href="#">Home
                                <span class="sr-only">(current)</span>
                        </a>
                        </li>
                        <li class="nav-item">
                        <a class="nav-link" href="#">Report</a>
                        </li>
                        <li class="nav-item">
                        <a class="nav-link" href="#">Add Expense</a>
                        </li>
                        <li class="nav-item">
                        <a class="nav-link" href="#">About</a>
                        </li>
                </ul>
        </div>
    </div>
</nav>

<app-expense-entry></app-expense-entry>
```

Here,

Used bootstrap navigation and containers.

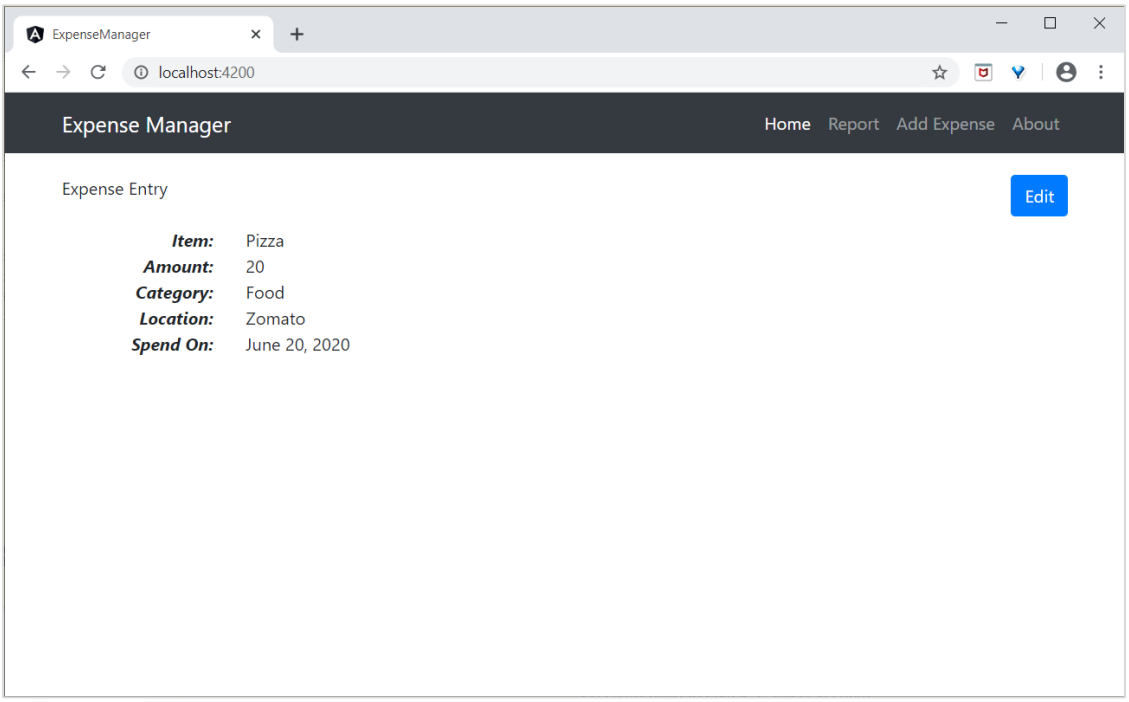
Open **src/app/expense-entry/expense-entry.component.html** and place below content.

```
<!-- Page Content -->
<div class="container">
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">
                        <div class="container" style="padding-left: 0px;
padding-right: 0px;">
                                <div class="row">
                                <div class="col-sm" style="text-align: left;">
                                        {{ title }}
                                </div>
                                <div class="col-sm" style="text-align: right;">
                                                <button type="button"
class="btn btn-primary">Edit</button>
                                </div>
```

```
                        </div>
                    </div>
                    <div class="container box" style="margin-top: 10px;">
                        <div class="row">
                        <div class="col-2" style="text-align: right;">
                            <strong><em>Item:</em></strong>
                        </div>
                        <div class="col" style="text-align: left;">
                            Pizza
                        </div>
                        </div>
                        <div class="row">
                        <div class="col-2" style="text-align: right;">
                            <strong><em>Amount:</em></strong>
                        </div>
                        <div class="col" style="text-align: left;">
                            20
                        </div>
                        </div>
                        <div class="row">
                        <div class="col-2" style="text-align: right;">
                            <strong><em>Category:</em></strong>
                        </div>
                        <div class="col" style="text-align: left;">
                            Food
                        </div>
                        </div>
                        <div class="row">
                        <div class="col-2" style="text-align: right;">
                            <strong><em>Location:</em></strong>
                        </div>
                        <div class="col" style="text-align: left;">
                           Zomato
                        </div>
                        </div>
                        <div class="row">
                        <div class="col-2" style="text-align: right;">
                            <strong><em>Spend On:</em></strong>
                        </div>
                        <div class="col" style="text-align: left;">
                            June 20, 2020
                        </div>
                        </div>
                    </div>
        </div>
    </div>
</div>
```

Restart the application.

The output of the application is as follows:



Add an interface

Create ExpenseEntry interface (**src/app/expense-entry.ts**) and add id, amount, category, Location, spendOn and createdOn.

```
export interface ExpenseEntry {
    id: number;
    item: string;
    amount: number;
    category: string;
    location: string;
    spendOn: Date;
    createdOn: Date;
}
```

Import **ExpenseEntry** into **ExpenseEntryComponent.**

```
import { ExpenseEntry } from '../expense-entry';
```

Create a **ExpenseEntry** object, expenseEntry as shown below:

```
export class ExpenseEntryComponent implements OnInit {
        title: string;
        expenseEntry: ExpenseEntry;
        constructor() { }

        ngOnInit() {
                this.title = "Expense Entry";

                this.expenseEntry = {
```

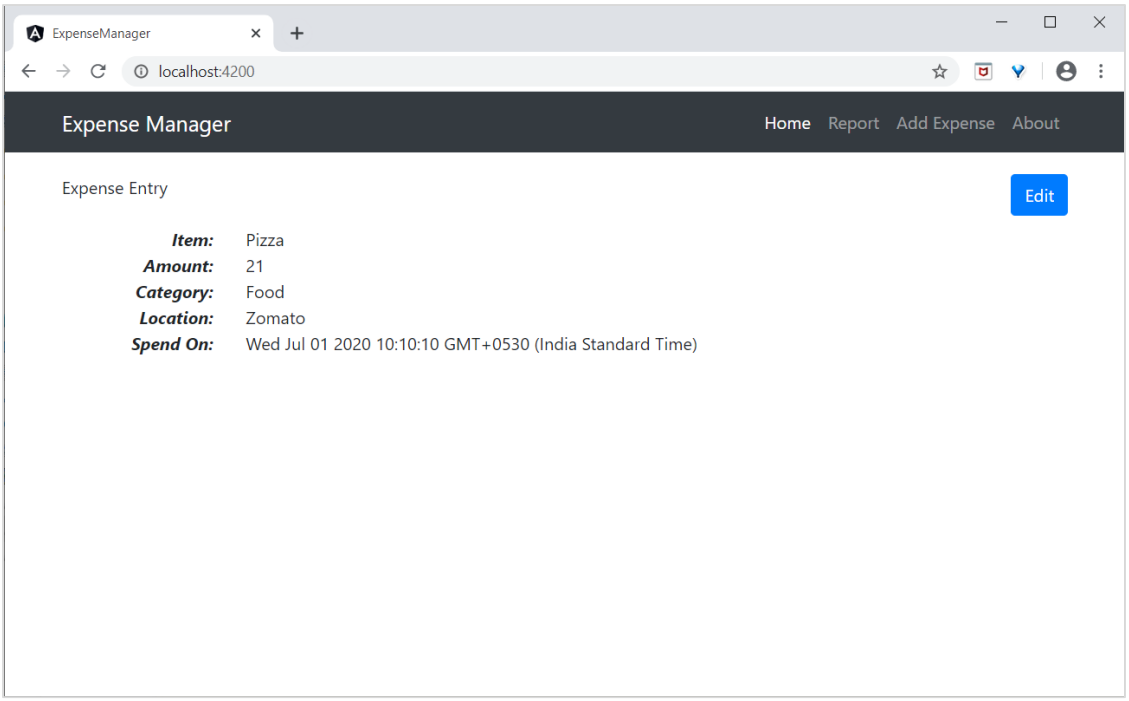
```
                        id: 1,
                        item: "Pizza",
                        amount: 21,
                        category: "Food",
                        location: "Zomato",
                        spendOn: new Date(2020, 6, 1, 10, 10, 10),
                        createdOn: new Date(2020, 6, 1, 10, 10, 10),
                };
        }
}
```

Update the component template using **expenseEntry object, src/app/expense-entry/expense-entry.component.html** as specified below:

```
<!-- Page Content -->
<div class="container">
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">
            <div class="container" style="padding-left: 0px; padding-right:
0px;">
                <div class="row">
                    <div class="col-sm" style="text-align: left;">
                        {{ title }}
                    </div>
                    <div class="col-sm" style="text-align: right;">
                        <button type="button" class="btn btn-
primary">Edit</button>
                    </div>
                </div>
            </div>
            <div class="container box" style="margin-top: 10px;">
                <div class="row">
                    <div class="col-2" style="text-align: right;">
                        <strong><em>Item:</em></strong>
                    </div>
                    <div class="col" style="text-align: left;">
                        {{ expenseEntry.item }}
                    </div>
                </div>
                <div class="row">
                    <div class="col-2" style="text-align: right;">
                        <strong><em>Amount:</em></strong>
                    </div>
                    <div class="col" style="text-align: left;">
                        {{ expenseEntry.amount }}
                    </div>
                </div>
                <div class="row">
                    <div class="col-2" style="text-align: right;">
                        <strong><em>Category:</em></strong>
                    </div>
                    <div class="col" style="text-align: left;">
```

```
                    {{ expenseEntry.category }}
                </div>
            </div>
            <div class="row">
                <div class="col-2" style="text-align: right;">
                    <strong><em>Location:</em></strong>
                </div>
                <div class="col" style="text-align: left;">
                    {{ expenseEntry.location }}
                </div>
            </div>
            <div class="row">
                <div class="col-2" style="text-align: right;">
                    <strong><em>Spend On:</em></strong>
                </div>
                <div class="col" style="text-align: left;">
                    {{ expenseEntry.spendOn }}
                </div>
            </div>
        </div>
    </div>
</div>
```

The output of the application is as follows:



Using directives

Let us add a new component in our **ExpenseManager** application to list the expense entries.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Start the application.

```
ng serve
```

Create a new component, **ExpenseEntryListComponent** using below command:

```
ng generate component ExpenseEntryList
```

Output

The output is given below:

```
CREATE src/app/expense-entry-list/expense-entry-list.component.html (33 bytes)
CREATE src/app/expense-entry-list/expense-entry-list.component.spec.ts (700
bytes)
CREATE src/app/expense-entry-list/expense-entry-list.component.ts (315 bytes)
CREATE src/app/expense-entry-list/expense-entry-list.component.css (0 bytes)
UPDATE src/app/app.module.ts (548 bytes)
```

Here, the command creates the ExpenseEntryList Component and update the necessary code in **AppModule.**

Import **ExpenseEntry** into **ExpenseEntryListComponent** component (**src/app/expense-entry-list/expense-entry-list.component**).

```
import { ExpenseEntry } from '../expense-entry';
```

Add a method, **getExpenseEntries()** to return list of expense entry (mock items) in ExpenseEntryListComponent (**src/app/expense-entry-list/expense-entry-list.component**).

```
getExpenseEntries() : ExpenseEntry[] {
        let mockExpenseEntries : ExpenseEntry[] = [
                { id: 1,
                        item: "Pizza",
                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "Mcdonald",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },

                { id: 1,
                        item: "Pizza",
                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "KFC",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
```

```
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },

                { id: 1,
                        item: "Pizza",
                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "Mcdonald",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },

                { id: 1,
                        item: "Pizza",
                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "KFC",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },

                { id: 1,
                        item: "Pizza",
                        amount: Math.floor((Math.random() * 10) + 1),
                        category: "Food",
                        location: "KFC",
                        spendOn: new Date(2020, 4, Math.floor((Math.random() *
30) + 1), 10, 10, 10),
                        createdOn: new Date(2020, 4, Math.floor((Math.random()
* 30) + 1), 10, 10, 10) },
        ];

        return mockExpenseEntries;
}
```

Declare a local variable, **expenseEntries** and load the mock list of expense entries as mentioned below:

```
title: string;
expenseEntries: ExpenseEntry[];
constructor() { }

ngOnInit() {
        this.title = "Expense Entry List";
        this.expenseEntries = this.getExpenseEntries();
}
```

Open the template file (**src/app/expense-entry-list/expense-entry-list.component.html**) and show the mock entries in a table.

```
<!-- Page Content -->
<div class="container">
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">
            <div class="container" style="padding-left: 0px; padding-right:
0px;">
                <div class="row">
                    <div class="col-sm" style="text-align: left;">
                        {{ title }}
                    </div>
                    <div class="col-sm" style="text-align: right;">
                        <button type="button" class="btn btn-
primary">Edit</button>
                    </div>
                </div>
            </div>
            <div class="container box" style="margin-top: 10px;">
                            <table class="table table-striped">
                                    <thead>
                                    <tr>
                                            <th>Item</th>
                                            <th>Amount</th>
                                            <th>Category</th>
                                            <th>Location</th>
                                            <th>Spent On</th>
                                    </tr>
                                </thead>
                                <tbody>
                                    <tr *ngFor="let entry of
expenseEntries">
                                            <th scope="row">{{ entry.item
}}</th>
                                            <th>{{ entry.amount }}</th>
                                            <td>{{ entry.category }}</td>
                                            <td>{{ entry.location }}</td>
                                            <td>{{ entry.spendOn }}</td>
                                    </tr>
                                </tbody>
                        </table>
            </div>
        </div>
    </div>
</div>
```

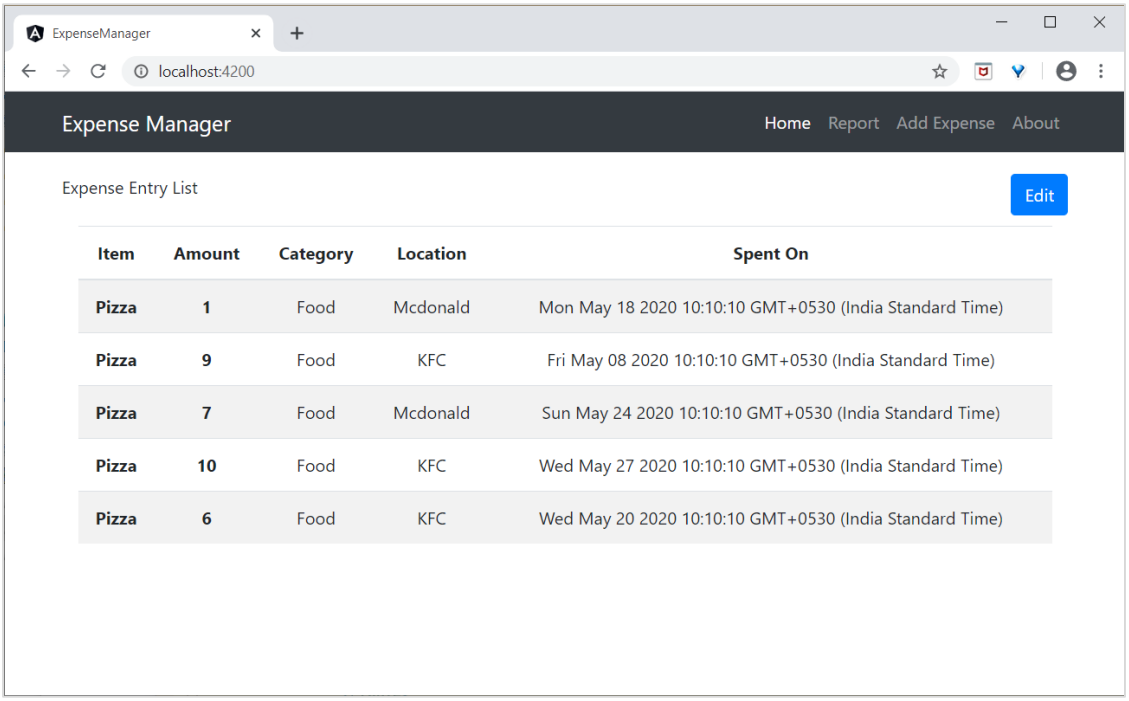
Here,

- Used bootstrap table. **table** and **table-striped** will style the table according to Boostrap style standard.
- Used **ngFor** to loop over the **expenseEntries** and generate table rows.

Open **AppComponent** template, **src/app/app.component.html** and include **ExpenseEntryListComponent** and remove **ExpenseEntryComponent** as shown below:

```
...
<app-expense-entry-list></app-expense-entry-list>
```

Finally, the output of the application is as shown below:



Use pipes

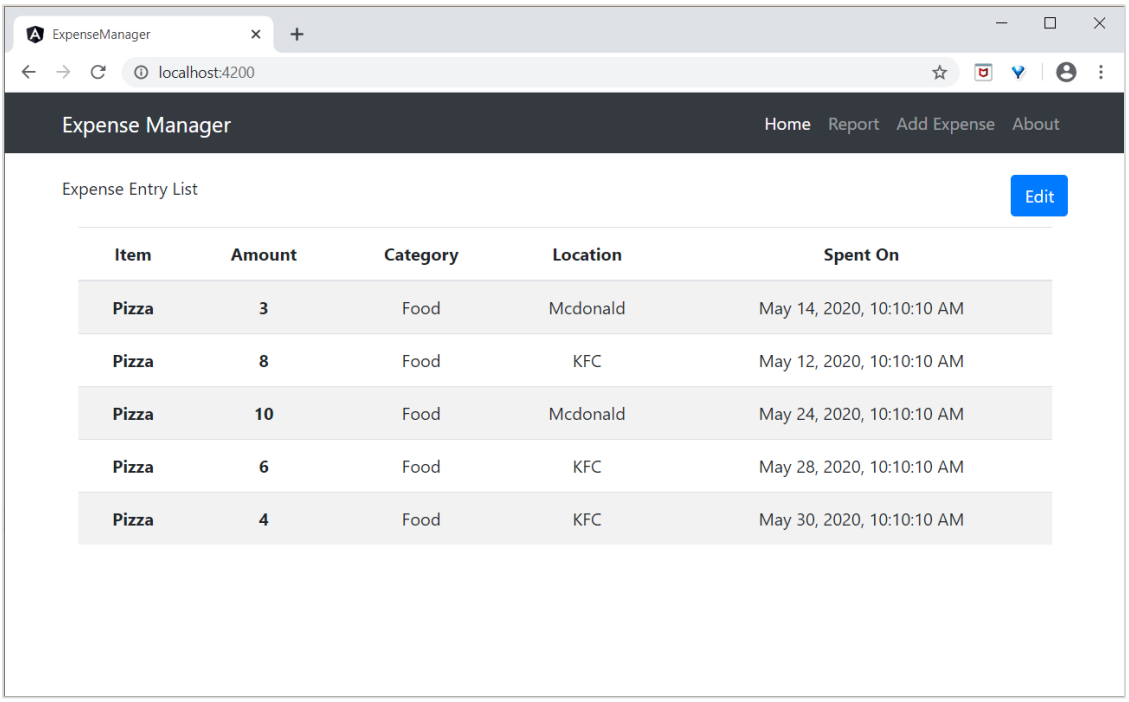
Let us use the pipe in the our **ExpenseManager** application.

Open **ExpenseEntryListComponent's** template, **src/app/expense-entry-list/expense-entry-list.component.html** and include pipe in **entry.spendOn** as mentioned below:

```
<td>{{ entry.spendOn | date: 'medium' }}</td>
```

Here, we have used the date pipe to show the spend on date in the short format.

Finally, the output of the application is as shown below:



Add debug service

Run the below command to generate an Angular service, **DebugService.**

```
ng g service debug
```

This will create two Typescript files (debug service & its test) as specified below:

```
CREATE src/app/debug.service.spec.ts (328 bytes)
CREATE src/app/debug.service.ts (134 bytes)
```

Let us analyse the content of the **DebugService** service.

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

@Injectable({
  providedIn: 'root'
})
export class DebugService {

  constructor() { }
}
```

Here,

- **@Injectable** decorator is attached to **DebugService** class, which enables the DebugService to be used in Angular component of the application.
- **providerIn** option and its value, **root** enables the DebugService to be used in all component of the application.

Let us add a method, **Info,** which will print the message into the browser console.

```
  info(message : String) : void {
          console.log(message);
  }
```

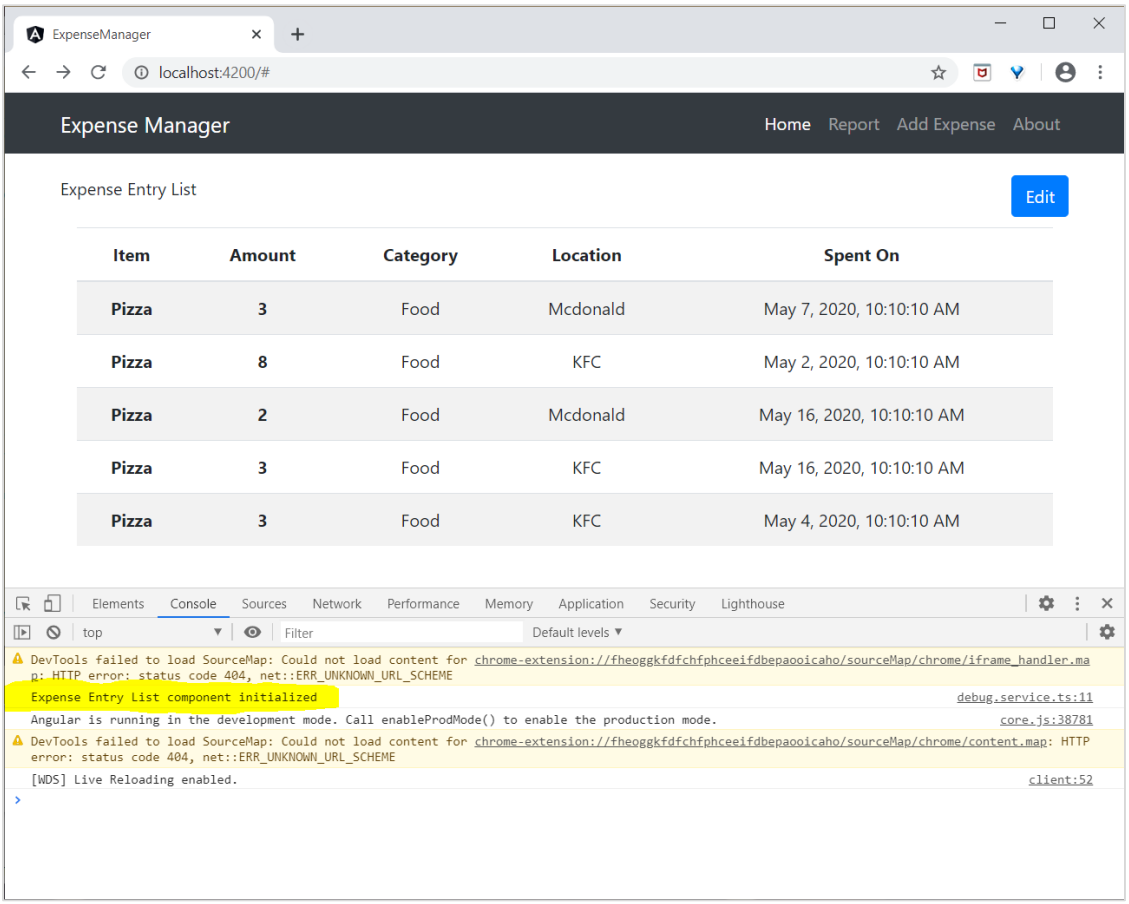
Let us initialise the service in the **ExpenseEntryListComponent** and use it to print message.

```
import { Component, OnInit } from '@angular/core';
import { ExpenseEntry } from '../expense-entry';
import { DebugService } from '../debug.service';

@Component({
        selector: 'app-expense-entry-list',
        templateUrl: './expense-entry-list.component.html',
        styleUrls: ['./expense-entry-list.component.css']
})
export class ExpenseEntryListComponent implements OnInit {
        title: string;
        expenseEntries: ExpenseEntry[];
        constructor(private debugService: DebugService) { }

        ngOnInit() {
        this.debugService.info("Expense Entry List component initialized");
                this.title = "Expense Entry List";
                this.expenseEntries = this.getExpenseEntries();
        }

        // other coding
}
```

Here,

- **DebugService** is initialised using constructor parameters. Setting an argument (**debugService**) of type **DebugService** will trigger the dependency injection to create a new DebugService object and set it into the **ExpenseEntryListComponent** component.
- Calling the **info** method of **DebugService** in the **ngOnInit** method prints the message in the browser console.

The result can be viewed using developer tools and it looks similar as shown below:



Let us extend the application to understand the scope of the service.

Let us a create a **DebugComponent** by using below mentioned command:

```
ng generate component debug

CREATE src/app/debug/debug.component.html (20 bytes)
CREATE src/app/debug/debug.component.spec.ts (621 bytes)
CREATE src/app/debug/debug.component.ts (265 bytes)
CREATE src/app/debug/debug.component.css (0 bytes)
UPDATE src/app/app.module.ts (392 bytes)
```

Let us delete the **DebugService** in the root module.

```
// src/app/debug.service.ts
import { Injectable } from '@angular/core';

@Injectable()
export class DebugService {
        constructor() {
        }

        info(message : String) : void {
        console.log(message);
        }
}
```

Register the **DebugService** under **ExpenseEntryListComponent** component.

```
// src/app/expense-entry-list/expense-entry-list.component.ts
@Component({
        selector: 'app-expense-entry-list',
        templateUrl: './expense-entry-list.component.html',
        styleUrls: ['./expense-entry-list.component.css']
        providers: [DebugService]
})
```

Here, we have used **providers** meta data (**ElementInjector**) to register the service.

Open **DebugComponent** *(*src/app/debug/debug.component.ts*)* and import **DebugService** and set an instance in the constructor of the component.

```
import { Component, OnInit } from '@angular/core';
import { DebugService } from '../debug.service';

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

  constructor(private debugService: DebugService) { }

  ngOnInit() {
          this.debugService.info("Debug component gets service from Parent");
  }
}
```

Here, we have not registered **DebugService**. So, DebugService will not be available if used as parent component. When used inside a parent component, the service may available from parent, if the parent has access to the service.

Open **ExpenseEntryListComponent** template (src/app/expense-entry-list/expense-entry-list.component.html) and include a content section as shown below:

```
// existing content
<app-debug></app-debug>
<ng-content></ng-content>
```

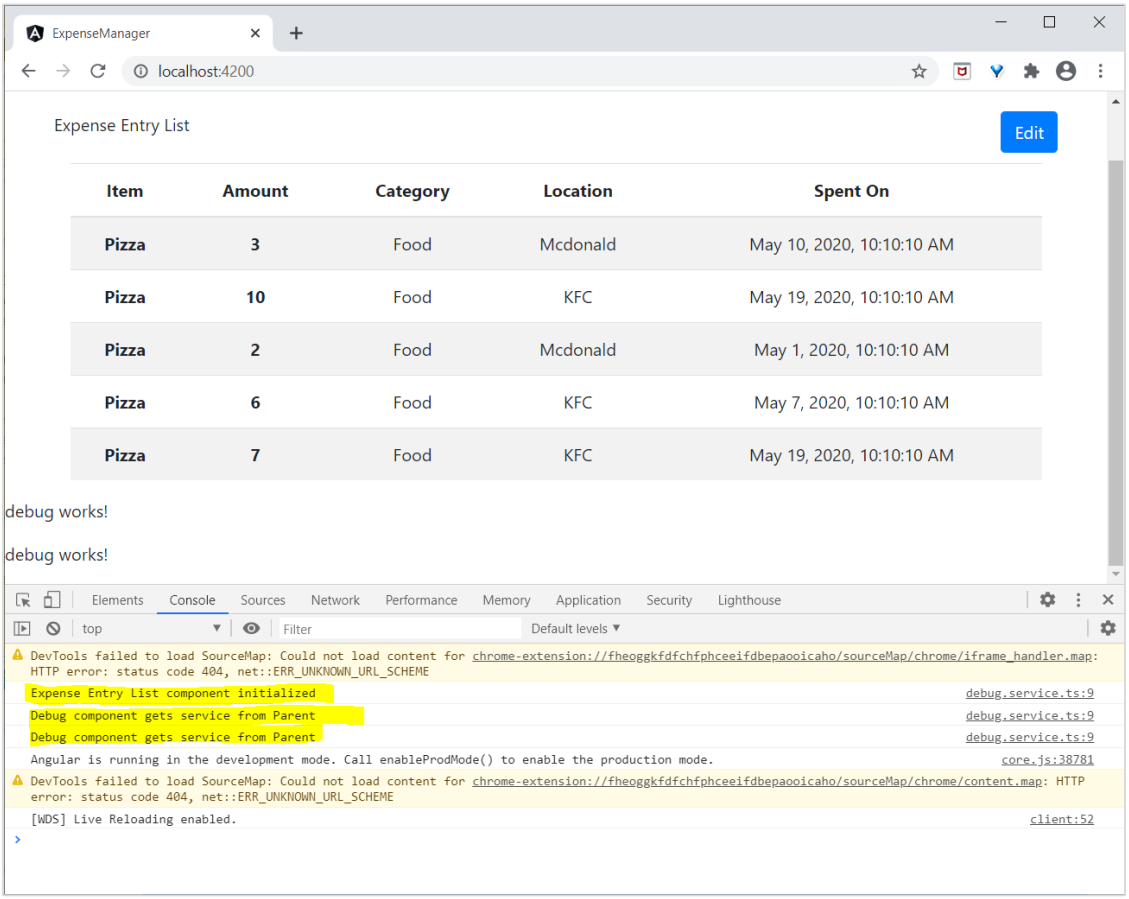
Here, we have included a content section and **DebugComponent** section.

Let us include the debug component as a content inside the **ExpenseEntryListComponent** component in the **AppComponent** template. Open **AppComponent** template and change **app-expense-entry-list** as below:

```
// navigation code

<app-expense-entry-list>
        <app-debug></app-debug>
</app-expense-entry-list>
```

Here, we have included the **DebugComponent** as content.

Let us check the application and it will show **DebugService** template at the end of the page as shown below:

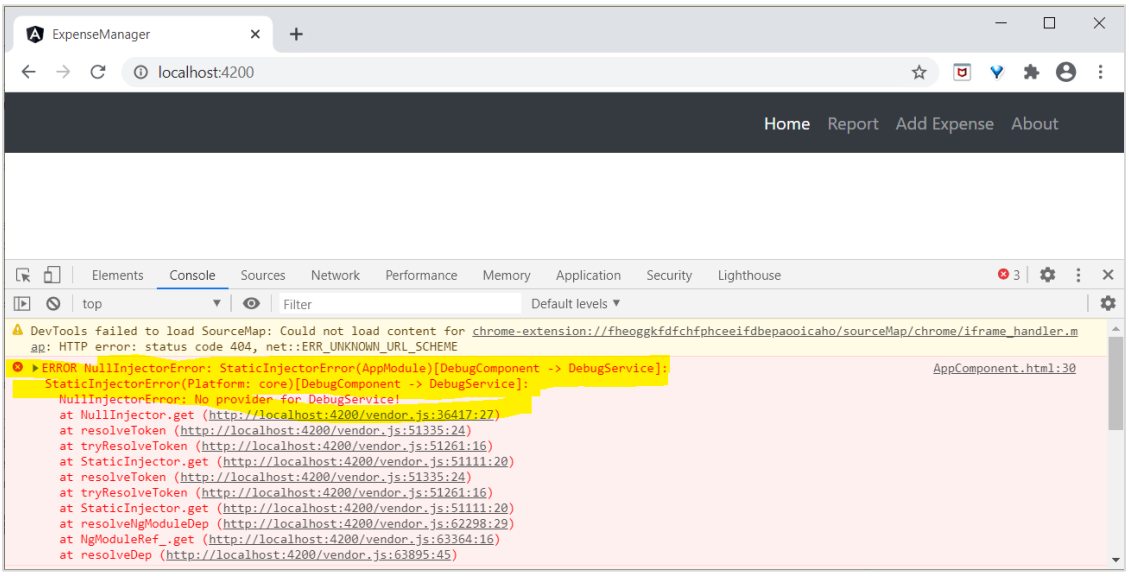


Also, we could able to see two debug information from debug component in the console. This indicate that the debug component gets the service from its parent component.

Let us change how the service is injected in the **ExpenseEntryListComponent** and how it affects the scope of the service. Change **providers** injector to **viewProviders** injection. viewProviders does not inject the service into the content child and so, it should fail.

```
viewProviders: [DebugService]
```

Check the application and you will see that the one of the debug component (used as content child) throws error as shown below.



Let us remove the debug component in the templates and restore the application.

Open **ExpenseEntryListComponent** template (**src/app/expense-entry-list/expense-entry-list.component.html**) and remove below content:

```
<app-debug></app-debug>
<ng-content></ng-content>
```

Open **AppComponent** template and change **app-expense-entry-list** as below:

```
// navigation code
<app-expense-entry-list> </app-expense-entry-list>
```

Change the **viewProviders** setting to **providers** in **ExpenseEntryListComponent.**

```
providers: [DebugService]
```

Rerun the appliation and check the result.

Create expense service

Let us create a new service **ExpenseEntryService** in our **ExpenseManager** application to interact with **Expense REST API**. Check the coding of Expense REST API in **Http Rest Programming** chapter.

ExpenseEntryService will get the latest expense entries, insert new expense entries, modify existing expense entries and delete the unwanted expense entries.

Run the below command to generate an Angular service, **ExpenseService.**

```
ng generate service ExpenseEntry
```

This will create two Typescript files (expense entry service & its test) as specified below:

```
CREATE src/app/expense-entry.service.spec.ts (364 bytes)
CREATE src/app/expense-entry.service.ts (141 bytes)
```

Import **HttpClientModule** into **AppModule** *(***src/app/app.module.ts***)* as specified below:

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

@NgModule({
  imports: [
    ...
    HttpClientModule
    ...
  ]
  ...
  ...
})
export class AppModule { }
```

Open **ExpenseEntryService** *(***src/app/expense-entry.service.ts***)* and import **ExpenseEntry, throwError** and **catchError** from **rxjs** library and **HttpClient, HttpHeaders** and **HttpErrorResponse** from **@angular/common/http** package.

```
import { Injectable } from '@angular/core';
import { ExpenseEntry } from './expense-entry';
import { throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';
import { HttpClient, HttpHeaders, HttpErrorResponse } from
'@angular/common/http';
```

Inject the **HttpClient** service into our service.

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

Create a variable, **expenseRestUrl** to specify the **Expense Rest API** endpoints.

```
private expenseRestUrl = 'http://localhost:8000/api/expense';
```

Create a variable, **httpOptions** to set the Http Header option. This will be used during the Http Rest API call by Angular **HttpClient** service.

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

The complete code is as follows:

```
import { Injectable } from '@angular/core';
import { ExpenseEntry } from './expense-entry';
import { Observable, throwError } from 'rxjs';
```

```
import { catchError, retry } from 'rxjs/operators';
import { HttpClient, HttpHeaders, HttpErrorResponse } from
'@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class ExpenseEntryService {
        private expenseRestUrl = 'api/expense';
        private httpOptions = {
                headers: new HttpHeaders( { 'Content-Type': 'application/json'
})
        };

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

Http programming using HttpClient service

Start the **Expense REST API** application as shown below:

```
cd /go/to/expense-rest-api
node .\server.js
```

Add **getExpenseEntries()** and **httpErrorHandler()** method in **ExpenseEntryService** (**src/app/expense-entry.service.ts**) service.

```
getExpenseEntries() : Observable<ExpenseEntry[]> {
    return this.httpClient.get<ExpenseEntry[]>(this.expenseRestUrl,
this.httpOptions)
                            .pipe(
                                    retry(3),
                                    catchError(this.httpErrorHandler)
                            );
}


getExpenseEntry(id: number) : Observable<ExpenseEntry> {
    return this.httpClient.get<ExpenseEntry>(this.expenseRestUrl + "/" + id,
this.httpOptions)
    .pipe(
        retry(3),
        catchError(this.httpErrorHandler)
    );
}

private httpErrorHandler (error: HttpErrorResponse) {
        if (error.error instanceof ErrorEvent) {
        console.error("A client side error occurs. The error message is " +
```

```
error.message);
        } else {
        console.error(
                "An error happened in server. The HTTP status code is "  +
error.status + " and the error returned is " + error.message);
        }

        return throwError("Error occurred. Pleas try again");
}
```

Here,

- **getExpenseEntries()** calls the **get()** method using expense end point and also configures the error handler. Also, it configures **httpClient** to try for maximum of 3 times in case of failure. Finally, it returns the response from server as typed (ExpenseEntry[]) Observable object.
- **getExpenseEntry** is similar to **getExpenseEntries()** except it passes the id of the **ExpenseEntry** object and gets ExpenseEntry Observable object.

The complete coding of **ExpenseEntryService** is as follows:

```
import { Injectable } from '@angular/core';
import { ExpenseEntry } from './expense-entry';
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
import { HttpClient, HttpHeaders, HttpErrorResponse } from
'@angular/common/http';

@Injectable({
        providedIn: 'root'
})
export class ExpenseEntryService {
        private expenseRestUrl = 'http://localhost:8000/api/expense';
        private httpOptions = {
                headers: new HttpHeaders( { 'Content-Type': 'application/json'
})
        };

    constructor(private httpClient : HttpClient) { }

    getExpenseEntries() : Observable<ExpenseEntry[]> {
        return this.httpClient.get<ExpenseEntry[]>(this.expenseRestUrl,
this.httpOptions)
        .pipe(
                retry(3),
                catchError(this.httpErrorHandler)
        );
    }

    getExpenseEntry(id: number) : Observable<ExpenseEntry> {
```

```
        return this.httpClient.get<ExpenseEntry>(this.expenseRestUrl + "/" +
id, this.httpOptions)
        .pipe(
                retry(3),
                catchError(this.httpErrorHandler)
        );
    }

    private httpErrorHandler (error: HttpErrorResponse) {
            if (error.error instanceof ErrorEvent) {
            console.error("A client side error occurs. The error message is " +
error.message);
            } else {
            console.error(
                    "An error happened in server. The HTTP status code is "  +
error.status + " and the error returned is " + error.message);
            }

            return throwError("Error occurred. Pleas try again");
    }
}
```

Open **ExpenseEntryListComponent** *(***src-entry-list-entry-list.component.ts***)* and inject **ExpenseEntryService** through constructor as specified below:

```
constructor(private debugService: DebugService, private restService :
ExpenseEntryService ) { }
```

Change the **getExpenseEntries()** function. Call **getExpenseEntries()** method from **ExpenseEntryService** instead of returning the mock items.

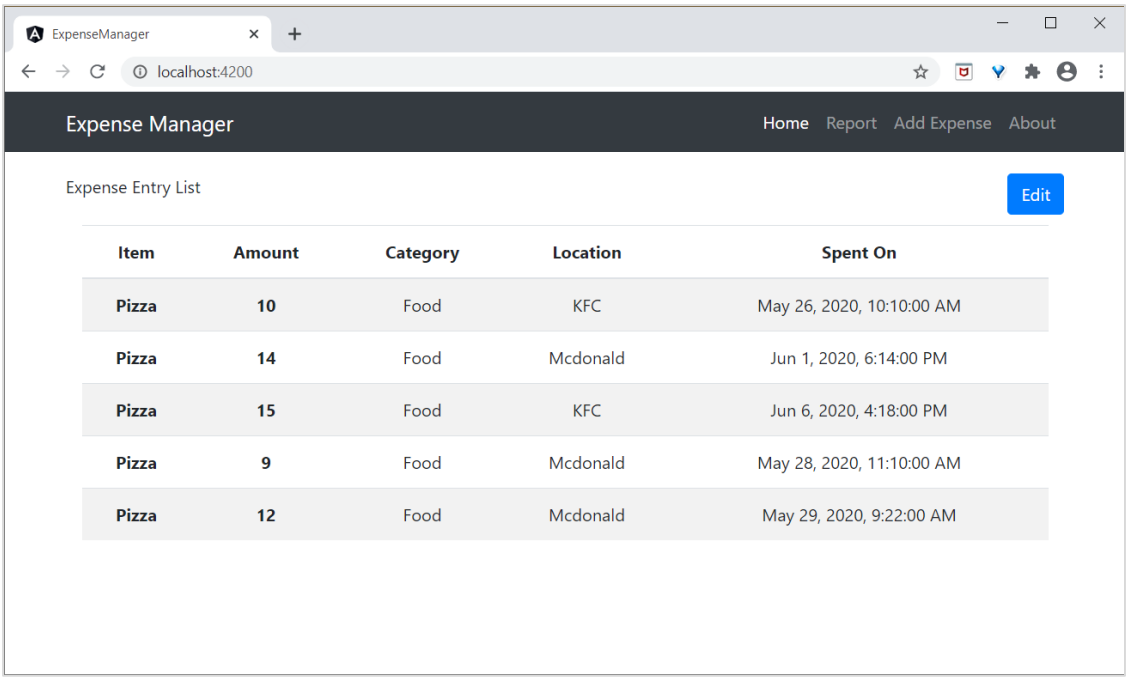
```
getExpenseItems() {
        this.restService.getExpenseEntries()
                .subscribe( data => this.expenseEntries = data );
}
```

The complete **ExpenseEntryListComponent** coding is as follows:

```
import { Component, OnInit } from '@angular/core';
import { ExpenseEntry } from '../expense-entry';
import { DebugService } from '../debug.service';
import { ExpenseEntryService } from '../expense-entry.service';

@Component({
        selector: 'app-expense-entry-list',
        templateUrl: './expense-entry-list.component.html',
        styleUrls: ['./expense-entry-list.component.css'],
        providers: [DebugService]
})
export class ExpenseEntryListComponent implements OnInit {
        title: string;
```

```
        expenseEntries: ExpenseEntry[];
        constructor(private debugService: DebugService, private restService :
ExpenseEntryService ) { }

        ngOnInit() {
        this.debugService.info("Expense Entry List component initialized");
                this.title = "Expense Entry List";

                this.getExpenseItems();
        }

        getExpenseItems() {
                this.restService.getExpenseEntries()
                        .subscribe( data => this.expenseEntries = data );
        }
}
```

Finally, check the application and you will see the below response:



Add Expense functionality

Let us add a new method, **addExpenseEntry()** in our **ExpenseEntryService** to add new expense entry as mentioned below:

```
addExpenseEntry(expenseEntry: ExpenseEntry): Observable<ExpenseEntry> {
    return this.httpClient.post<ExpenseEntry>(this.expenseRestUrl,
expenseEntry, this.httpOptions)
    .pipe(
            retry(3),
```

```
            catchError(this.httpErrorHandler)
    );
}
```

Update Expense functionality

Let us add a new method, **updateExpenseEntry()** in our **ExpenseEntryService** to update existing expense entry as mentioned below:

```
updateExpenseEntry(expenseEntry: ExpenseEntry): Observable<ExpenseEntry> {
    return this.httpClient.put<ExpenseEntry>(this.expenseRestUrl + "/" +
expenseEntry.id, expenseEntry, this.httpOptions)
    .pipe(
            retry(3),
            catchError(this.httpErrorHandler)
    );
}
```

Delete expense entry functionality

Let us add a new method, **deleteExpenseEntry()** in our **ExpenseEntryService** to delete existing expense entry as mentioned below:

```
deleteExpenseEntry(expenseEntry: ExpenseEntry | number) :
Observable<ExpenseEntry> {
    const id = typeof expenseEntry == 'number' ? expenseEntry : expenseEntry.id
    const url = `${this.expenseRestUrl}/${id}`;

    return this.httpClient.delete<ExpenseEntry>(url, this.httpOptions)
    .pipe(
            retry(3),
            catchError(this.httpErrorHandler)
    );
}
```

Add Routing

Generate routing module using below command, if not done before.

```
ng generate module app-routing --module app --flat
```

Output

The output is as follows:

```
CREATE src/app/app-routing.module.ts (196 bytes)
UPDATE src/app/app.module.ts (785 bytes)
```

Here,

CLI generate **AppRoutingModule** and then, configures it in **AppModule.**

Update **AppRoutingModule** *(***src/app/app.module.ts***)* as mentioned below:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ExpenseEntryComponent } from './expense-entry/expense-
entry.component';
import { ExpenseEntryListComponent } from './expense-entry-list/expense-entry-
list.component';

const routes: Routes = [
        { path: 'expenses', component: ExpenseEntryListComponent },
        { path: 'expenses/detail/:id', component: ExpenseEntryComponent },
        { path: '', redirectTo: 'expenses', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Here, we have added route for our expense list and expense details component.

Update **AppComponent** template (**src/app/app.component.html**) to include **router-outlet** and **routerLink.**

```
<!-- Navigation -->
<nav class="navbar navbar-expand-lg navbar-dark bg-dark static-top">
    <div class="container">
        <a class="navbar-brand" href="#">{{ title }}</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarResponsive" aria-controls="navbarResponsive" aria-
expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarResponsive">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item active">
                    <a class="nav-link" href="#">Home
                        <span class="sr-only" routerLink="/">(current)</span>
                    </a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" routerLink="/expenses">Report</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Add Expense</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">About</a>
                </li>
            </ul>
        </div>
```

```
    </div>
</nav>

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

Open **ExpenseEntryListComponent** template (**src/app/expense-entry-list/expense-entry-list.component.html**) and include view option for every expense entry.

```
<table class="table table-striped">
    <thead>
        <tr>
                <th>Item</th>
                <th>Amount</th>
                <th>Category</th>
                <th>Location</th>
                <th>Spent On</th>
                <th>View</th>
        </tr>
    </thead>
    <tbody>
            <tr *ngFor="let entry of expenseEntries">
                    <th scope="row">{{ entry.item }}</th>
                    <th>{{ entry.amount }}</th>
                    <td>{{ entry.category }}</td>
                    <td>{{ entry.location }}</td>
                    <td>{{ entry.spendOn | date: 'medium' }}</td>
                    <td><a routerLink="../expenses/detail/{{ entry.id
}}">View</a></td>
            </tr>
    </tbody>
</table>
```

Here, we have updated the expense list table and added a new column to show the view option.

Open **ExpenseEntryComponent** *(***src/app/expense-entry/expense-entry.component.ts***)* and add functionality to fetch the current selected expense entry. It can be done by first getting the **id** through the **paramMap** and then, using the **getExpenseEntry()** method from **ExpenseEntryService.**

```
        this.expenseEntry$ = this.route.paramMap.pipe(
                switchMap(params => {
                        this.selectedId = Number(params.get('id'));
                        return
this.restService.getExpenseEntry(this.selectedId);
                }));

        this.expenseEntry$.subscribe( (data) => this.expenseEntry = data );
```

Update **ExpenseEntryComponent** and add option to go to expense list.

```
    goToList() {
        this.router.navigate(['/expenses']);
    }
```

The complete code of **ExpenseEntryComponent** is as follows:

```
import { Component, OnInit } from '@angular/core';
import { ExpenseEntry } from '../expense-entry';
import { ExpenseEntryService } from '../expense-entry.service';
import { Router, ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
import { switchMap } from 'rxjs/operators';

@Component({
        selector: 'app-expense-entry',
        templateUrl: './expense-entry.component.html',
        styleUrls: ['./expense-entry.component.css']
})
export class ExpenseEntryComponent implements OnInit {
        title: string;
        expenseEntry$ : Observable<ExpenseEntry>;
        expenseEntry: ExpenseEntry = {} as ExpenseEntry;
        selectedId: number;

        constructor(private restService : ExpenseEntryService,
                                private router : Router, private route :
ActivatedRoute ) { }

        ngOnInit() {
                this.title = "Expense Entry";

        this.expenseEntry$ = this.route.paramMap.pipe(
                switchMap(params => {
                        this.selectedId = Number(params.get('id'));
                        return
this.restService.getExpenseEntry(this.selectedId);
                }));

        this.expenseEntry$.subscribe( (data) => this.expenseEntry = data );
    }

    goToList() {
        this.router.navigate(['/expenses']);
    }
}
```

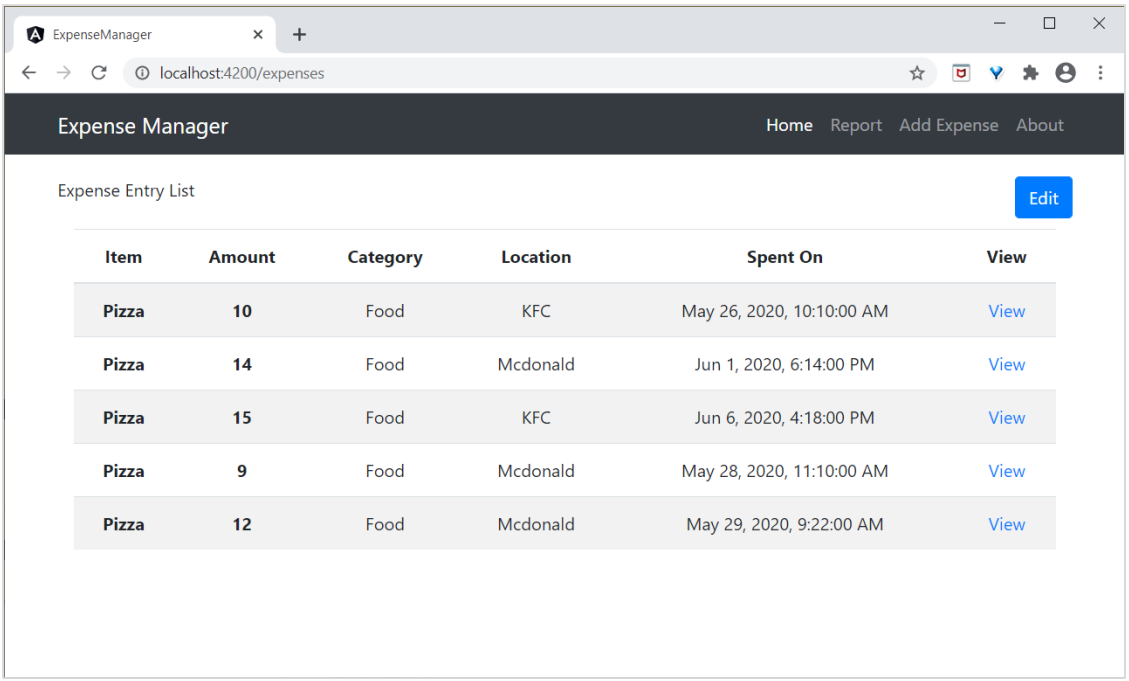
Open **ExpenseEntryComponent** *(***src/app/expense-entry/expense-entry.component.html***)* template and add a new button to navigate back to expense list page.

```
<div class="col-sm" style="text-align: right;">
    <button type="button" class="btn btn-primary" (click)="goToList()">Go to
```

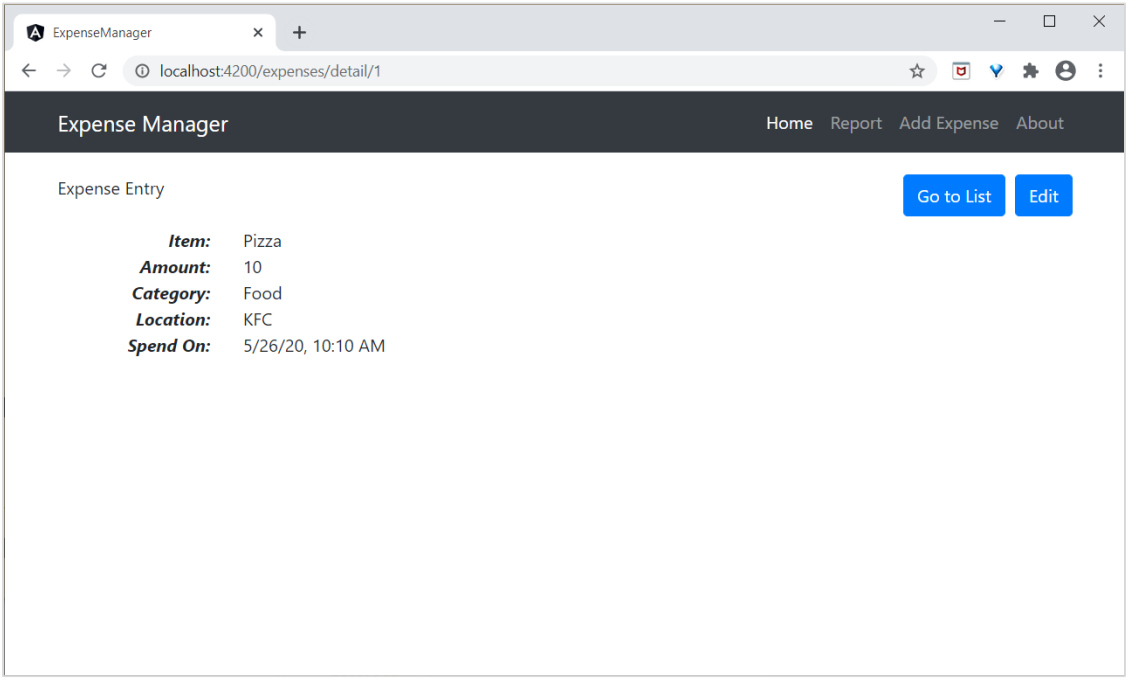
```
List</button>
     <button type="button" class="btn btn-primary">Edit</button>
</div>
```

Here, we have added **Go to List** button before **Edit** button.

The final output of the application is as follows:



Clicking the view option of the first entry will navigate to details page and show the selected expense entry as shown below:



Enable login and logout feature

Create a new service, **AuthService** to authenticate the user.

```
ng generate service auth

CREATE src/app/auth.service.spec.ts (323 bytes)
CREATE src/app/auth.service.ts (133 bytes)
```

Open **AuthService** and include below code:

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

import { Observable, of } from 'rxjs';
import { tap, delay } from 'rxjs/operators';

@Injectable({
        providedIn: 'root'
})
export class AuthService {

        isUserLoggedIn: boolean = false;

        login(userName: string, password: string): Observable<boolean> {
                console.log(userName);
                console.log(password);
                this.isUserLoggedIn = userName == 'admin' && password ==
'admin';
                localStorage.setItem('isUserLoggedIn', this.isUserLoggedIn ?
"true" : "false");

        return of(this.isUserLoggedIn).pipe(
                delay(1000),
                tap(val => {
                        console.log("Is User Authentication is successful: " +
val);
                })
        );
        }

        logout(): void {
        this.isUserLoggedIn = false;
                localStorage.removeItem('isUserLoggedIn');
        }

        constructor() { }
}
```

Here,

- We have written two methods, **login** and **logout.**
- The purpose of the **login** method is to validate the user and if the user successfully validated, it stores the information in **localStorage** and then, returns true.

- Authentication validation is that the user name and password should be **admin**.
- We have not used any backend. Instead we have simulated a delay of 1s using Observables.
- The purpose of the **logout** method is to invalidate the user and removes the information stored in **localStorage**.

Create a **login** component using below command:

```
ng generate component login

CREATE src/app/login/login.component.html (20 bytes)
CREATE src/app/login/login.component.spec.ts (621 bytes)
CREATE src/app/login/login.component.ts (265 bytes)
CREATE src/app/login/login.component.css (0 bytes)
UPDATE src/app/app.module.ts (1207 bytes)
```

Open **LoginComponent** and include below code:

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

import { FormGroup, FormControl } from '@angular/forms';
import { AuthService } from '../auth.service';
import { Router } from '@angular/router';

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

        userName: string;
        password: string;
        formData: FormGroup;

        constructor(private authService : AuthService, private router : Router)
{ }

        ngOnInit() {
                this.formData = new FormGroup({
                        userName: new FormControl("admin"),
                        password: new FormControl("admin"),
                });
        }

        onClickSubmit(data: any) {
                this.userName = data.userName;
                this.password = data.password;

                console.log("Login page: " + this.userName);
```

```
                console.log("Login page: " + this.password);

                this.authService.login(this.userName, this.password)
                        .subscribe( data => {
                                console.log("Is Login Success: " + data);
                                if(data) this.router.navigate(['/expenses']);
                });
        }
}
```

Here,

- Used reactive forms.
- Imported **AuthService** and **Router** and configured it in constructor.
- Created an instance of **FormGroup** and included two instance of **FormControl,** one for **user name** and another for **password.**
- Created a **onClickSubmit** to validate the user using **authService** and if successful, navigate to expense list.

Open **LoginComponent** template and include below template code:

```
<!-- Page Content -->
<div class="container">
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">
            <div class="container box" style="margin-top: 10px; padding-left:
0px; padding-right: 0px;">
                <div class="row">
                    <div class="col-12" style="text-align: center;">
                                                <form [formGroup]="formData"
(ngSubmit)="onClickSubmit(formData.value)"
                                                        class="form-signin">
                                                <h2 class="form-signin-
heading">Please sign in</h2>
                                                <label for="inputEmail"
class="sr-only">Email address</label>
                                                <input type="text"
id="username" class="form-control"

formControlName="userName" placeholder="Username" required autofocus>
                                                <label for="inputPassword"
class="sr-only">Password</label>
                                                <input type="password"
id="inputPassword" class="form-control"

formControlName="password" placeholder="Password" required>
                                                <button class="btn btn-lg btn-
primary btn-block" type="submit">Sign in</button>
                                                </form>
```

```
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

Here,

- Created a reactive form and designed a login form.
- Attached the **onClickSubmit** method to the form submit action.

Open **LoginComponent** style and include below CSS Code:

```
.form-signin {
        max-width: 330px;
        padding: 15px;
        margin: 0 auto;
}

input {
        margin-bottom: 20px;
}
```

Here, some styles are added to design the login form.

Create a logout component using below command:

```
ng generate component logout

CREATE src/app/logout/logout.component.html (21 bytes)
CREATE src/app/logout/logout.component.spec.ts (628 bytes)
CREATE src/app/logout/logout.component.ts (269 bytes)
CREATE src/app/logout/logout.component.css (0 bytes)
UPDATE src/app/app.module.ts (1368 bytes)
```

Open **LogoutComponent** and include below code:

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

import { AuthService } from '../auth.service';
import { Router } from '@angular/router';

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

        constructor(private authService : AuthService, private router: Router)
```

```
 { }

        ngOnInit() {
                this.authService.logout();
                this.router.navigate(['/']);
        }

}
```

Here,

- Used **logout** method of **AuthService**.
- Once the user is logged out, the page will redirect to home page (/).

Create a guard using below command:

```
ng generate guard expense

CREATE src/app/expense.guard.spec.ts (364 bytes)
CREATE src/app/expense.guard.ts (459 bytes)
```

Open **ExpenseGuard** and include below code:

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router,
UrlTree } from '@angular/router';
import { Observable } from 'rxjs';

import { AuthService } from './auth.service';

@Injectable({
        providedIn: 'root'
})
export class ExpenseGuard implements CanActivate {

        constructor(private authService: AuthService, private router: Router)
{}

        canActivate(
        next: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): boolean | UrlTree {
                let url: string = state.url;

                    return this.checkLogin(url);
                }

                checkLogin(url: string): true | UrlTree {
                        console.log("Url: " + url)
                        let val: string =
localStorage.getItem('isUserLoggedIn');

                        if(val != null && val == "true"){
                                if(url == "/login")
```

```
                                        this.router.parseUrl('/expenses');
                                else
                                        return true;
                        } else {
                                return this.router.parseUrl('/login');
                        }
                }
}
```

Here,

- **checkLogin** will check whether the localStorage has the user information and if it is available, then it returns true.
- If the user is logged in and goes to login page, it will redirect the user to **expenses** page.
- If the user is not logged in, then the user will be redirected to **login** page.

Open **AppRoutingModule** *(***src/app/app-routing.module.ts***)* and update below code:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ExpenseEntryComponent } from './expense-entry/expense-
entry.component';
import { ExpenseEntryListComponent } from './expense-entry-list/expense-entry-
list.component';
import { LoginComponent } from './login/login.component';
import { LogoutComponent } from './logout/logout.component';

import { ExpenseGuard } from './expense.guard';

const routes: Routes = [
        { path: 'login', component: LoginComponent },
        { path: 'logout', component: LogoutComponent },
        { path: 'expenses', component: ExpenseEntryListComponent, canActivate:
[ExpenseGuard]},
        { path: 'expenses/detail/:id', component: ExpenseEntryComponent,
canActivate: [ExpenseGuard]},
        { path: '', redirectTo: 'expenses', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Here,

- Imported LoginComponent and LogoutComponent.
- Imported ExpenseGuard.

- Created two new routes, login and logout to access LoginComponent and LogoutComponent respectively.
- Add new option canActivate for ExpenseEntryComponent and ExpenseEntryListComponent.

Open **AppComponent** template and add two **login** and **logout** link.

```
<div class="collapse navbar-collapse" id="navbarResponsive">
    <ul class="navbar-nav ml-auto">
        <li class="nav-item active">
            <a class="nav-link" href="#">Home
                <span class="sr-only" routerLink="/">(current)</span>
            </a>
        </li>
        <li class="nav-item">
            <a class="nav-link" routerLink="/expenses">Report</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="#">Add Expense</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="#">About</a>
        </li>
        <li class="nav-item">
                        <div *ngIf="isUserLoggedIn; else isLogOut">
                                <a class="nav-link"
routerLink="/logout">Logout</a>
                        </div>

                        <ng-template #isLogOut>
                                        <a class="nav-link"
routerLink="/login">Login</a>
                        </ng-template>
        </li>
    </ul>
</div>
```

Open **AppComponent** and update below code:

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

import { AuthService } from './auth.service';

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

        title = 'Expense Manager';
```

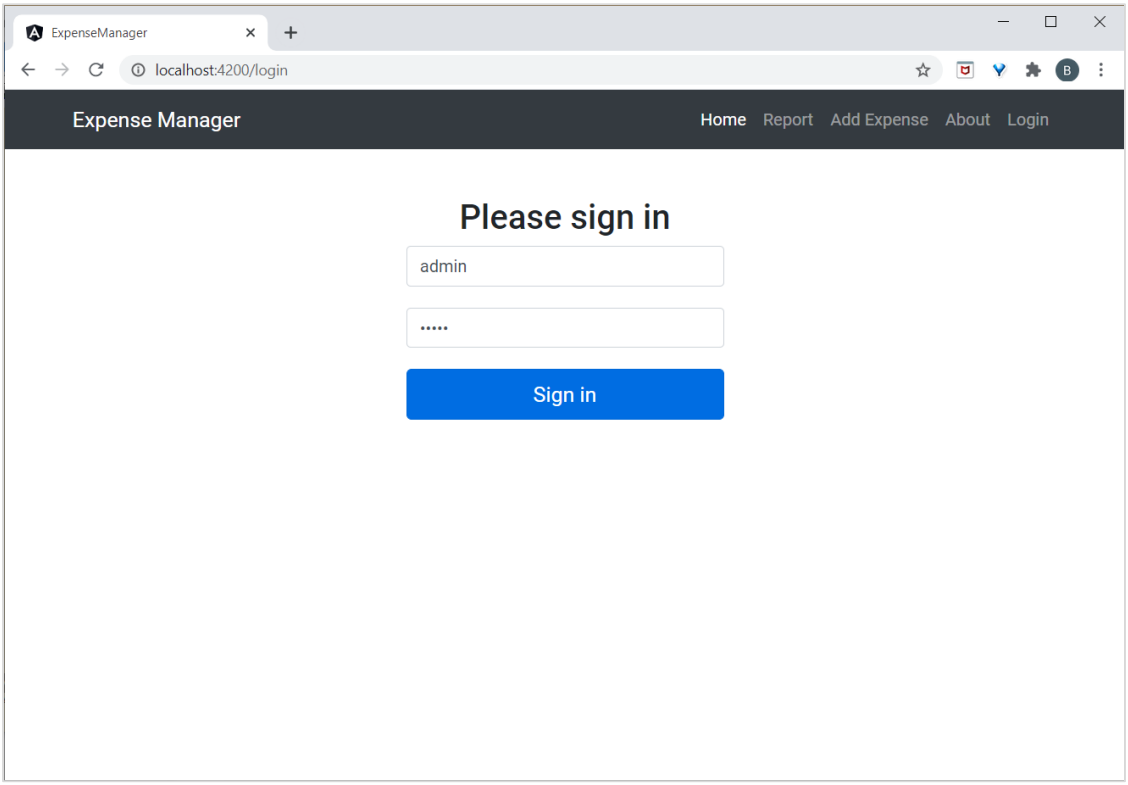
```
        isUserLoggedIn = false;

        constructor(private authService: AuthService) {}

        ngOnInit() {
                let storeData = localStorage.getItem("isUserLoggedIn");
                console.log("StoreData: " + storeData);

                if( storeData != null && storeData == "true")
                        this.isUserLoggedIn = true;
                else
                        this.isUserLoggedIn = false;
        }
}
```

Here, we have added the logic to identify the user status so that we can show **login / logout** functionality.

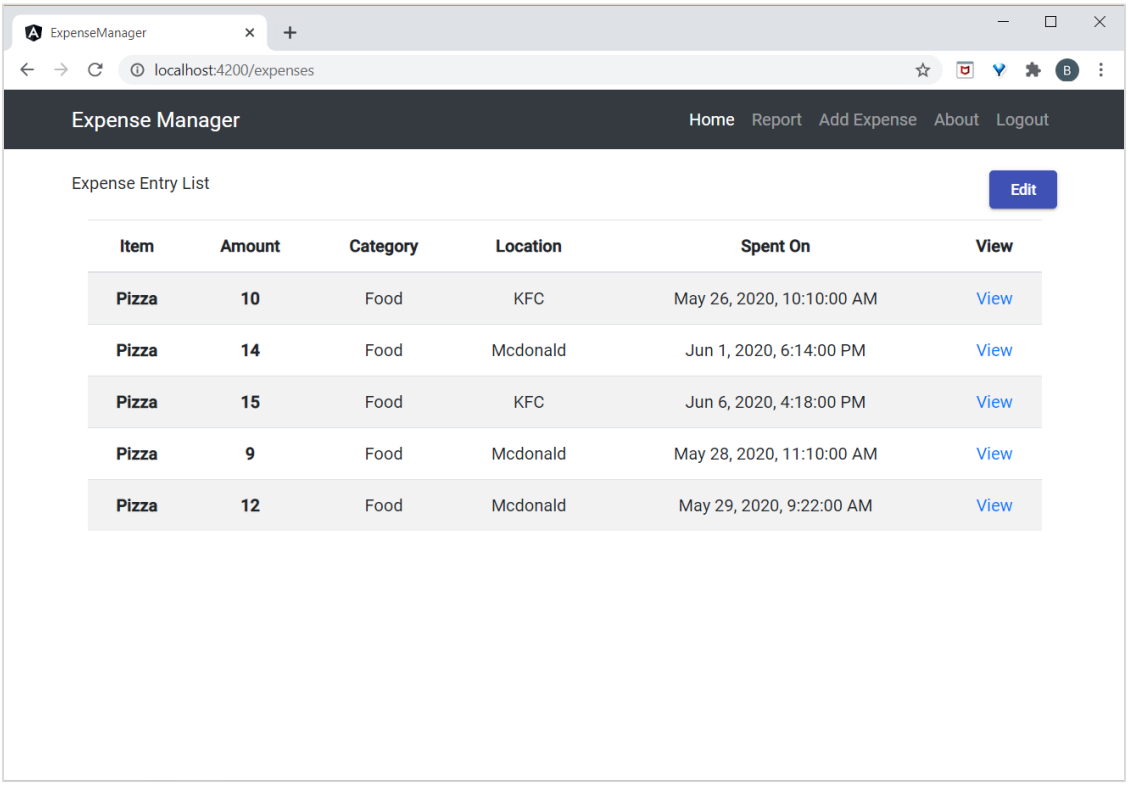
Open **AppModule (src/app/app.module.ts)** and configure **ReactiveFormsModule.**

```
import { ReactiveFormsModule } from '@angular/forms';

imports: [
    ReactiveFormsModule
]
```

Now, run the application and the application opens the login page.



Enter **admin** and **admin** as username and password and then click submit. The application process the login and redirects the user to expense list page as shown below:



Finally, you can click logout and exit the application.

Add / Edit / Delete Expenses

Add new component, **EditEntryComponent** to add new expense entry and edit the existing expense entries using below command:

```
ng generate component EditEntry
```

```
CREATE src/app/edit-entry/edit-entry.component.html (25 bytes)
CREATE src/app/edit-entry/edit-entry.component.spec.ts (650 bytes)
CREATE src/app/edit-entry/edit-entry.component.ts (284 bytes)
CREATE src/app/edit-entry/edit-entry.component.css (0 bytes)
UPDATE src/app/app.module.ts (1146 bytes)
```

Update **EditEntryComponent** with below code:

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

import { FormGroup, FormControl, Validators } from '@angular/forms';

import { ExpenseEntry } from '../expense-entry';
import { ExpenseEntryService } from '../expense-entry.service';

import { Router, ActivatedRoute } from '@angular/router';
```

```
@Component({
        selector: 'app-edit-entry',
        templateUrl: './edit-entry.component.html',
        styleUrls: ['./edit-entry.component.css']
})
export class EditEntryComponent implements OnInit {
        id: number;
        item: string;
        amount: number;
        category: string;
        location: string;
        spendOn: Date;

        formData: FormGroup;
        selectedId: number;
        expenseEntry: ExpenseEntry;

        constructor(private expenseEntryService : ExpenseEntryService, private
router: Router, private route: ActivatedRoute) { }

        ngOnInit() {
                this.formData = new FormGroup({
                        id: new FormControl(),
                        item: new FormControl('', [Validators.required]),
                        amount: new FormControl('', [Validators.required]),
                        category: new FormControl(),
                        location: new FormControl(),
                        spendOn: new FormControl()
                });

                this.selectedId =
Number(this.route.snapshot.paramMap.get('id'));

                if(this.selectedId != null && this.selectedId != 0) {

this.expenseEntryService.getExpenseEntry(this.selectedId)
                                .subscribe( (data) =>
                                        {
                                                this.expenseEntry = data;

this.formData.controls['id'].setValue(this.expenseEntry.id);

this.formData.controls['item'].setValue(this.expenseEntry.item);

this.formData.controls['amount'].setValue(this.expenseEntry.amount);

this.formData.controls['category'].setValue(this.expenseEntry.category);

this.formData.controls['location'].setValue(this.expenseEntry.location);
```

```
this.formData.controls['spendOn'].setValue(this.expenseEntry.spendOn);
                                        })
                }



        }

        get itemValue() {
        return this.formData.get('item');
        }

        get amountValue() {
        return this.formData.get('amount');
        }

    onClickSubmit(data: any) {
        console.log('onClickSubmit fired');
        this.id = data.id;
        this.item = data.item;
        this.amount = data.amount;
        this.category = data.category;
        this.location = data.location;
        this.spendOn = data.spendOn;

        let expenseEntry : ExpenseEntry = {
                id: this.id,
            item: this.item,
            amount: this.amount,
            category: this.category,
            location: this.location,
            spendOn: this.spendOn,
            createdOn: new Date(2020, 5, 20)
        }
        console.log(expenseEntry);

                if(expenseEntry.id == null || expenseEntry.id == 0) {
                        console.log('add fn fired');
                this.expenseEntryService.addExpenseEntry(expenseEntry)
                        .subscribe( data => { console.log(data);
this.router.navigate(['/expenses']); });
        } else {
                        console.log('edit fn fired');
                this.expenseEntryService.updateExpenseEntry(expenseEntry)
                        .subscribe( data => { console.log(data);
this.router.navigate(['/expenses']); });
        }
    }
}
```

Here,

- Created a form, **formData** in the **ngOnInit** method using **FormControl** and **FormGroup** classes with proper validation rules.
- Loaded the expense entry to be edited in the **ngOnInit** method.
- Created two methods, **itemValue** and **amountValue** to get the item and amount values respectively entered by user for the validation purpose.
- Created method, **onClickSubmit** to save (add / update) the expense entry.
- Used Expense service to add and update expense entries.

Update the **EditEntryComponent** template with expense form as shown below:

```
<!-- Page Content -->
<div class="container">
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">
            <div class="container" style="padding-left: 0px; padding-right:
0px;">
            </div>
            <div class="container box" style="margin-top: 10px;">
<form [formGroup]="formData" (ngSubmit)="onClickSubmit(formData.value)"
class="form" novalidate>
  <div class="form-group">
    <label for="item">Item</label>
    <input type="hidden" class="form-control" id="id" formControlName="id">
    <input type="text" class="form-control" id="item" formControlName="item">
    <div
        *ngIf="!itemValue?.valid && (itemValue?.dirty ||itemValue?.touched)">
        <div [hidden]="!itemValue.errors.required">
                Item is required
        </div>
        </div>
  </div>
  <div class="form-group">
    <label for="amount">Amount</label>
    <input type="text" class="form-control" id="amount"
formControlName="amount">
    <div
        *ngIf="!amountValue?.valid && (amountValue?.dirty
||amountValue?.touched)">
        <div [hidden]="!amountValue.errors.required">
                Amount is required
        </div>
        </div>
  </div>
  <div class="form-group">
    <label for="category">Category</label>
    <select class="form-control" id="category" formControlName="category">
      <option>Food</option>
      <option>Vegetables</option>
      <option>Fruit</option>
      <option>Electronic Item</option>
```

```
      <option>Bill</option>
    </select>
  </div>
  <div class="form-group">
    <label for="location">location</label>
    <input type="text" class="form-control" id="location"
formControlName="location">
  </div>
  <div class="form-group">
    <label for="spendOn">spendOn</label>
    <input type="text" class="form-control" id="spendOn"
formControlName="spendOn">
  </div>
<button class="btn btn-lg btn-primary btn-block" type="submit"
[disabled]="!formData.valid">Submit</button>
</form>
            </div>
        </div>
    </div>
</div>
```

Here,

- Created a form and bind it to the form, **formData** created in the class.
- Validated **item** and **amount** as required values.
- Called **onClickSubmit** function once validation in successful.

Open **EditEntryComponent** stylesheet and update below code:

```
.form {
        max-width: 330px;
        padding: 15px;
        margin: 0 auto;
}

.form label {
        text-align: left;
        width: 100%;
}

input {
        margin-bottom: 20px;
}
```

Here, we have styled the expense entry form.

Add **AboutComponent** using below command:

```
ng generate component About

CREATE src/app/about/about.component.html (20 bytes)
```

```
CREATE src/app/about/about.component.spec.ts (621 bytes)
CREATE src/app/about/about.component.ts (265 bytes)
CREATE src/app/about/about.component.css (0 bytes)
UPDATE src/app/app.module.ts (1120 bytes)
```

Open **AboutComponent** and add title as specified below:

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

@Component({
        selector: 'app-about',
        templateUrl: './about.component.html',
        styleUrls: ['./about.component.css']
})
export class AboutComponent implements OnInit {
        title = "About";
        constructor() { }

        ngOnInit() {
        }

}
```

Open **AboutComponent** template and updated content as specified below:

```
<!-- Page Content -->
<div class="container">
    <div class="row">
        <div class="col-lg-12 text-center" style="padding-top: 20px;">
            <div class="container" style="padding-left: 0px; padding-right:
0px;">
                <div class="row">
                    <div class="col-sm" style="text-align: left;">
                        <h1>{{ title }}</h1>
                    </div>
                </div>
            </div>
            <div class="container box" style="margin-top: 10px;">
                <div class="row">
                    <div class="col" style="text-align: left;">
                        <p>Expense management Application</p>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

Add routing for add and edit expense entries as specified below:

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

```
import { Routes, RouterModule } from '@angular/router';
import { ExpenseEntryComponent } from './expense-entry/expense-
entry.component';
import { ExpenseEntryListComponent } from './expense-entry-list/expense-entry-
list.component';
import { LoginComponent } from './login/login.component';
import { LogoutComponent } from './logout/logout.component';
import { EditEntryComponent } from './edit-entry/edit-entry.component';
import { AboutComponent } from './about/about.component';

import { ExpenseGuard } from './expense.guard';

const routes: Routes = [
    { path: 'about', component: AboutComponent },
    { path: 'login', component: LoginComponent },
    { path: 'logout', component: LogoutComponent },
    { path: 'expenses', component: ExpenseEntryListComponent, canActivate:
[ExpenseGuard]},
    { path: 'expenses/detail/:id', component: ExpenseEntryComponent,
canActivate: [ExpenseGuard]},
    { path: 'expenses/add', component: EditEntryComponent, canActivate:
[ExpenseGuard]},
    { path: 'expenses/edit/:id', component: EditEntryComponent, canActivate:
[ExpenseGuard]},
    { path: '', redirectTo: 'expenses', pathMatch: 'full' }
];

@NgModule({
        imports: [RouterModule.forRoot(routes)],
        exports: [RouterModule]
})
export class AppRoutingModule { }
```

Here, we have added **about, add expense** and **edit expense** routes.

Add **Edit** and **Delete** links in **ExpenseEntryListComponent** template.

```
<table class="table table-striped">
    <thead>
            <tr>
            <th>Item</th>
            <th>Amount</th>
            <th>Category</th>
            <th>Location</th>
            <th>Spent On</th>
            <th>View</th>
                    <th>Edit</th>
                    <th>Delete</th>
            </tr>
    </thead>
    <tbody>
        <tr *ngFor="let entry of expenseEntries">
```

```
        <th scope="row">{{ entry.item }}</th>
        <th>{{ entry.amount }}</th>
        <td>{{ entry.category }}</td>
        <td>{{ entry.location }}</td>
        <td>{{ entry.spendOn | date: 'medium' }}</td>
        <td><a routerLink="../expenses/detail/{{ entry.id }}">View</a></td>
        <td><a routerLink="../expenses/edit/{{ entry.id }}">Edit</a></td>
        <td><a href="#" (click)="deleteExpenseEntry($event,
entry.id)">Delete</a></td>
        </tr>
    </tbody>
</table>
```

Here, we have included two more columns. One column is used to show edit link and another to show delete link.

Update **deleteExpenseEntry** method in **ExpenseEntryListComponent** as shown below:

```
deleteExpenseEntry(evt, id) {
        evt.preventDefault();
        if(confirm("Are you sure to delete the entry?")) {
                this.restService.deleteExpenseEntry(id)
                        .subscribe( data => console.log(data) );

                this.getExpenseItems();
        }
}
```

Here, we have asked to confirm the deletion and it user confirmed, called the **deleteExpenseEntry** method from expense service to delete the selected expense item.

Change **Edit** link in the **ExpenseEntryListComponent** template at the top to **Add** link as shown below:

```
<div class="col-sm" style="text-align: right;">
    <button class="btn btn-primary" routerLink="/expenses/add">ADD</button>
    <!-- <button type="button" class="btn btn-primary">Edit</button> -->
</div>
```

Add **Edit** link in **ExpenseEntryComponent** template.

```
<div class="col-sm" style="text-align: right;">
    <button type="button" class="btn btn-primary" (click)="goToList()">Go to
List</button>
     <button type="button" class="btn btn-primary"
(click)="goToEdit()">Edit</button>
</div>
```

Open **ExpenseEntryComponent** and add **goToEdit()** method as shown below:

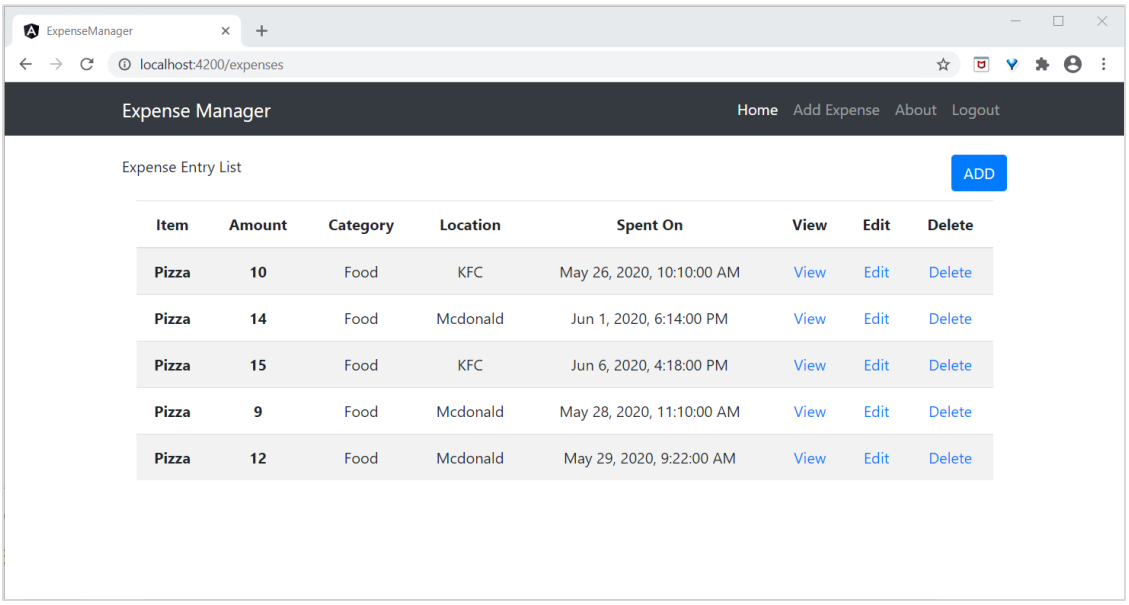
```
goToEdit() {
    this.router.navigate(['/expenses/edit', this.selectedId]);
}
```

Update navigation links in **AppComponent** template.

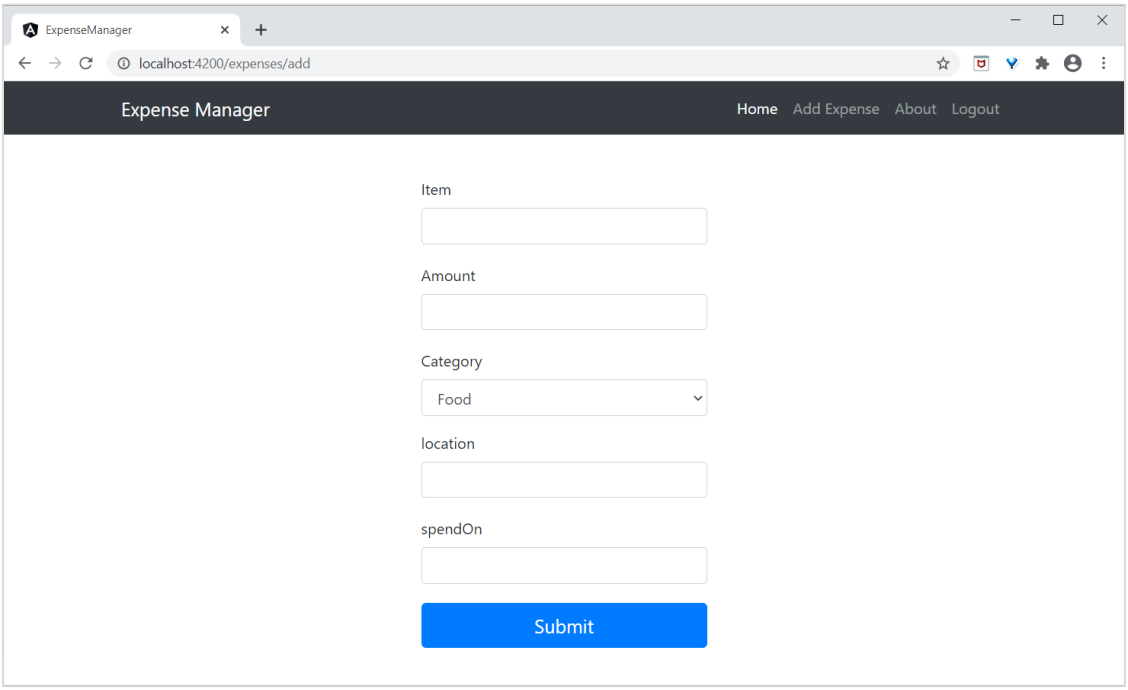
```
<!-- Navigation -->
<nav class="navbar navbar-expand-lg navbar-dark bg-dark static-top">
    <div class="container">
        <a class="navbar-brand" href="#">{{ title }}</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarResponsive" aria-controls="navbarResponsive" aria-
expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarResponsive">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item active">
                    <a class="nav-link" href="#">Home
                        <span class="sr-only" routerLink="/">(current)</span>
                    </a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" routerLink="/expenses/add">Add
Expense</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" routerLink="/about">About</a>
                </li>
                <li class="nav-item">
                                <div *ngIf="isUserLoggedIn; else isLogOut">
                                        <a class="nav-link"
routerLink="/logout">Logout</a>
                                </div>

                                <ng-template #isLogOut>
                                                <a class="nav-link"
routerLink="/login">Login</a>
                                </ng-template>
                </li>
            </ul>
        </div>
    </div>
</nav>

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

Here, we have updated the **add expense** link and **about** link.

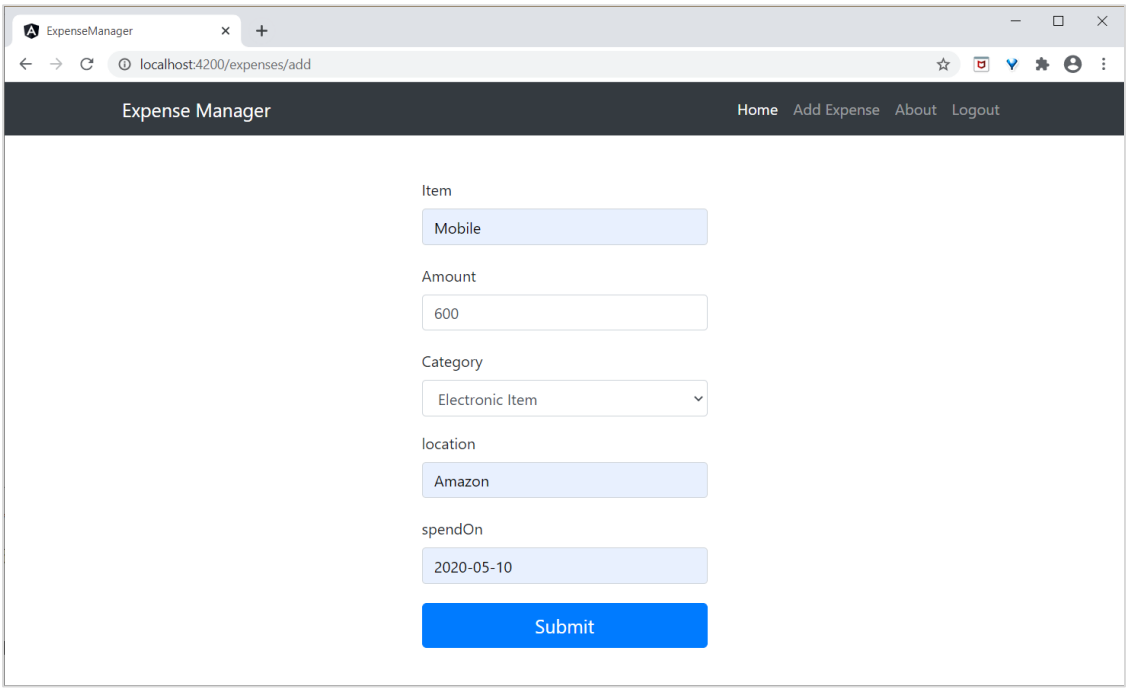
Run the application and the output will be similar as shown below:



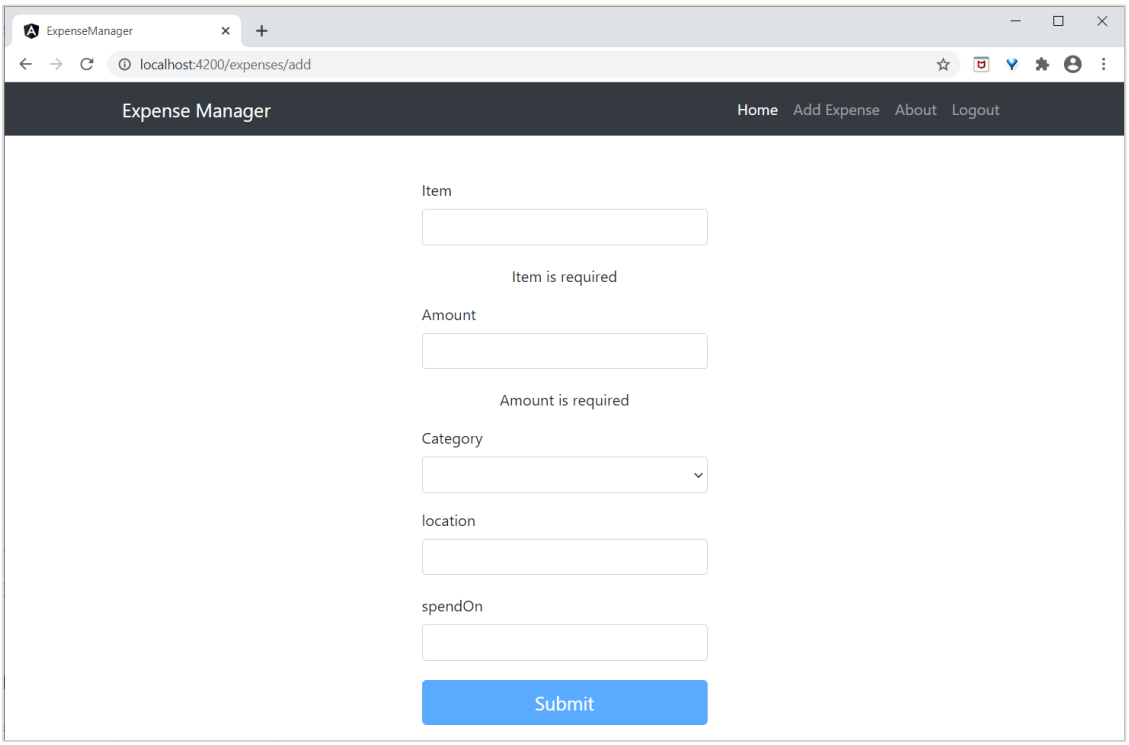
Try to add new expense using **Add** link in expense list page. The output will be similar as shown below:



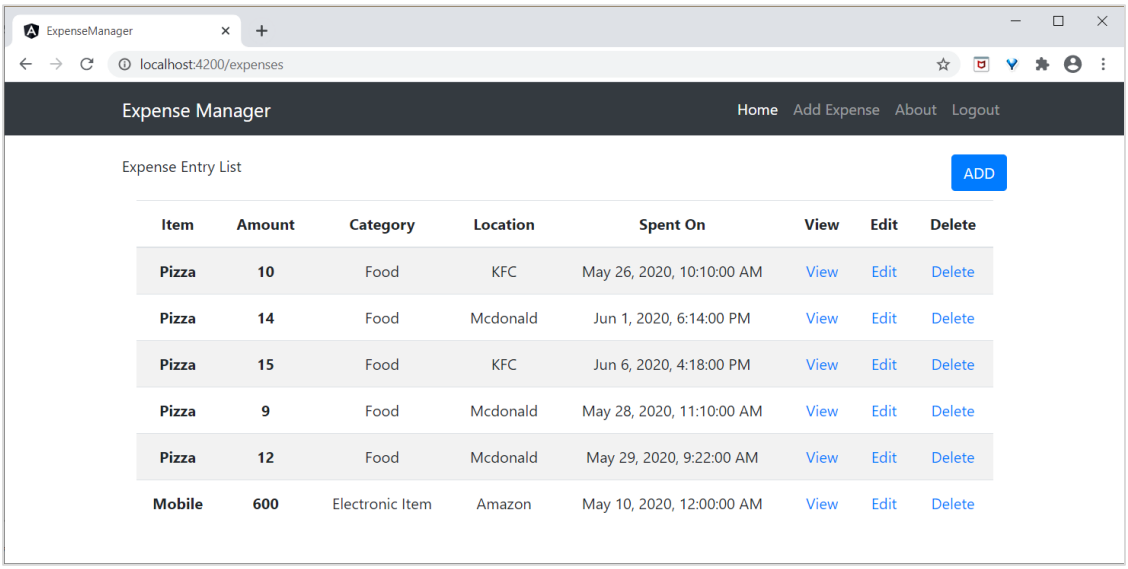
Fill the form as shown below:



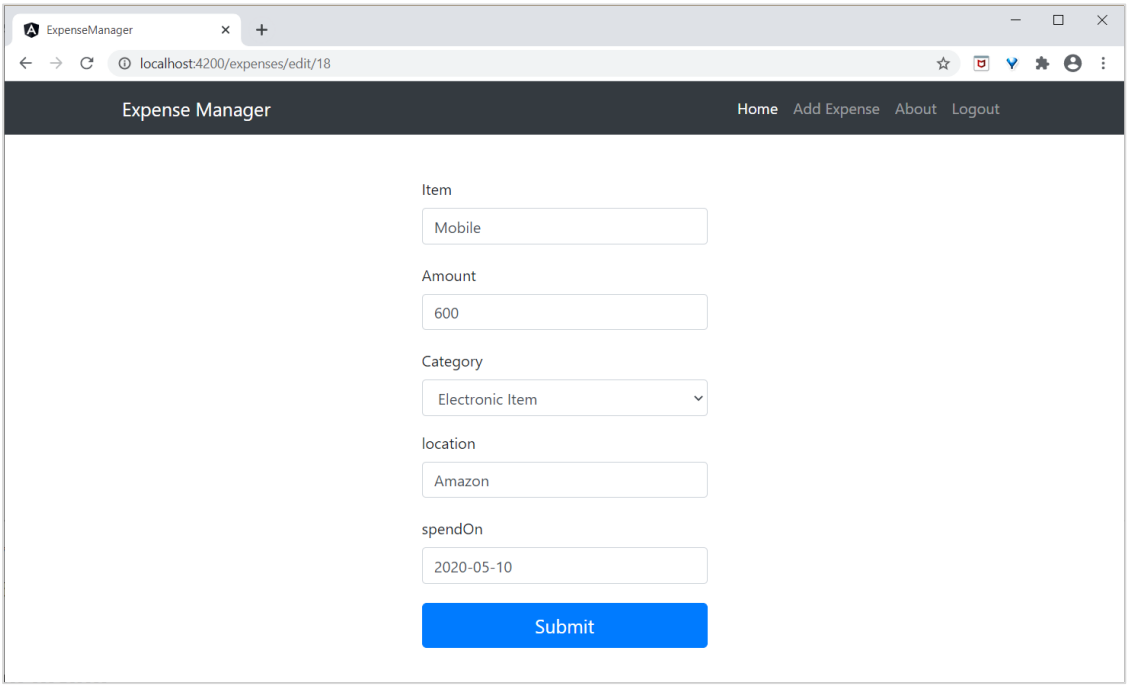
If the data is not filled properly, the validation code will alert as shown below:



Click **Submit**. It will trigger the submit event and the data will be saved to the backend and redirected to list page as shown below:

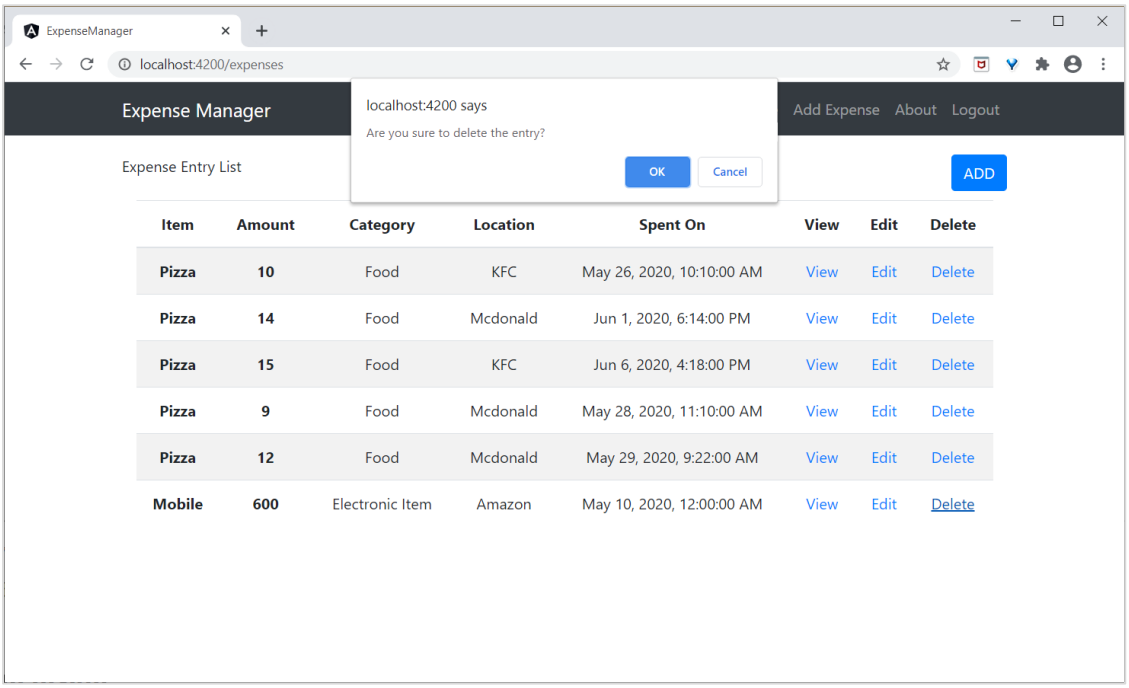


Try to edit existing expense using **Edit** link in expense list page. The output will be similar as shown below:



Click **Submit**. It will trigger the submit event and the data will be saved to the backend and redirected to list page.

To delete an item, click delete link. It will confirm the deletion as shown below:



Finally, we have implemented all features necessary to manage expenses in our application.

29. Angular 9 — What's New?

Angular community has continuosly updating its version. This chapter explains about Angular 9 version updates.

Install Angular 9

If you want to work with Angular 9, first you need to setup Angular 9 CLI using the below command:

```
npm install -g @angular/cli@^9.0.0
```

After executing this command, you can check the version using the below command:

```
ng version
```

Angular 9 Updates

Let's understand Angular 9 updates in brief.

Ivy compiler

Ivy compiler becomes the default compiler in Angular 9. This makes apps will be faster and very efficient. Whereas, Angular 8 Ivy is optional. We have to enable it inside tsconfig.json file.

Ivy compiler supports the following features:

- **Performs faster testing** - TestBed implementation helps to test more efficient.
- **Improved CSS class and styles** - Ivy styles are easily merged and designed as predictable.
- **Improved type checking** - This feature helps to find the errors earlier in development process.
- **Enhanced debugging** - Ivy comes with more tools to enable better debugging features. This will be helpful to show useful stack trace so that we can easily jump to the instruction.
- **Ahead-of-Time compiler** - This is one of the important improvements in compiler's performance. AOT builds are very faster.
- **Improved internationalization** - i18n substitutions helps to build more than ten times faster than previous versions.

Reliable ng update

ng updates are very reliable. It contains clear progress updates and runs all of the migrations. This can be done using the below command:

```
ng update --create-commits
```

Here,

–create-commits flag is used to commit your code after each migration.

Improved Dependency Injection

@Injectable service helps to add injector in your application. **providedIn** meta data provides new option, **platform** to ensure the object can be used and shared by all application. It is defined below:

```
@Injectable({
  providedIn: 'platform'
})
class MyService {...}
```

TypeScript 3.8

Angular 9 is designed to support 3.8 version. TypeScript 3.8 brings support for the below features:

- Type-Only Imports and Exports.
- ECMAScript Private Fields.
- Top-Level await.
- JSDoc Property Modifiers.
- export * as ns Syntax.

Angular 9.0.0-next.5

Angular 9.0.0-next.5 build has small size of main.js file, which makes better performance compare to previous version of Angular 8.

IDE enhancement

Angular 9 provides imporves IDE supports. TextMate grammar enables for syntax highlighting in inline and external templates.

Conclusion

Angular is flexible, ever improving, continuously updated and dependable framework. Angular greatly simplify the process of SPA development. By providing new features in each release like **Angular Universal, Progressive Web App, Web workers, Bazel build, Ivy Compiler,** etc., Angular will have a long life and complete support of the front end developer.