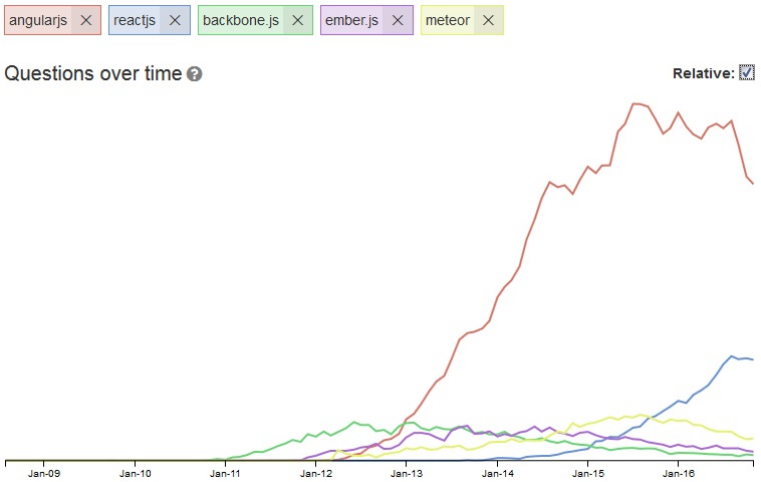
Implement a single-page application with Angular 2

Babu Suresh

December 14, 2016

Use Angular 2 and TypeScript to implement a single-page application. Consume microservices, improve application performance, autoscale your application, reduce server stress, and increase application usability.

Single-page application (SPA) technology is generating high interest in the software industry because of its potential for better-performing browser and smartphone applications. And over the last half decade, developers' interest in Angular— an open source SPA framework — has far surpassed their interest in other Web frameworks (React, Backbone, Meteor, Ember), judging by the number of questions about each on StackOverflow:



(Image source: tag-trends)

Web and mobile developers eagerly welcomed Angular 2 (released in September 2016). Angular 2 isn't an upgrade from Angular 1 but rather a completely new, different, and more advanced framework. Proficiency in Angular 2 is already a highly sought-after skill for building high-performing, scalable, robust, and modern cross-platform applications.

TypeScript

Angular 2 supports JavaScript, Dart, and TypeScript. You'll use TypeScript as the development language for this tutorial's project. The framework is built on TypeScript, and

most of the documentation, books, and tutorials on Angular focus on TypeScript as the development language.

This tutorial introduces you to the key building blocks of Angular 2 and walks you through coding and running a SPA using Angular 2 in both your development computer and a sandbox server. To make the most of the tutorial, you need some experience in web programming including general knowledge of JavaScript, TypeScript, and HTML. Prior Angular experience isn't required. After you work through the sample project, you'll be well prepared to create your own SPAs with Angular 2.

You can download the full sample code for the sandbox app from the "Downloadable resources" section at the end of the tutorial.

Why SPA and Angular 2?

An SPA renders only one HTML page from the server, when the user starts the app. Along with that one HTML page, the server sends an application engine to the client. The engine controls the entire application including processing, input, output, painting, and loading of the HTML pages. Typically, 90–95 percent of the application code runs in the browser; the rest works in the server when the user needs new data or must perform secured operations such as authentication. Because dependency on the server is mostly removed, an SPA autoscales in the Angular 2 environment: No matter how many users access the server simultaneously, 90–95 percent of the time the app's performance is never impacted.

Also, because most of the load is running on the client, the server is idle most of the time. Low demand on server resources reduces stress on the server significantly, potentially reducing server costs.

Another advantage of Angular 2 is that it helps SPAs to consume microservices effectively.

> *“ Typically, 90 to 95 percent of SPA code runs in the browser; the rest works in the server when the user needs new data or must perform secured operations such as authentication. ”*

Angular 2 concepts

Key concepts in Angular 2 include:

- Modules
- Components
- Services
- Routes

From now on, I'll refer to Angular 2 simply as Angular.

Modules

Angular applications are modular. Every Angular application has at least one *module*— the root module, conventionally named `AppModule`. The root module can be the only module in a small

application, but most apps have many more modules. As the developer, it's up to you to decide how to use the modules concept. Typically, you map major functionality or a feature to a module. Let's say you have five major areas in your system. Each one will have its own module in addition to the root module, totaling six modules.

Components

A *component* controls a patch of the page, called a *view*. You define a component's application logic — what it does to support the view inside a class. The class interacts with the view through an API of properties and methods. A component contains a class, a template, and metadata. A template is a form of HTML that tells Angular how to render the component. A component can belong to one and only one module.

Services

A *service* provides any value, function, or feature that your application needs. A service is typically a class with a narrow, well-defined purpose; it should do something specific and do it well. Components are big consumers of services. Services are big consumers of microservices.

Routes

Routes enable navigation from one view to the next as users perform application tasks. A route is equivalent to a mechanism used to control menus and submenus.

Now that you understand the benefits of SPAs and have a grasp on Angular concepts, it's time to get set up to work on the sample project.

What you'll need

To complete the sample project, you need Node.js and Angular CLI (a command-line interface for Angular) installed on your development PC:

- To install Node.js:
  1. Download the version for your system and choose the default options to complete the installation.
  2. Run `node -v` from your OS command line to verify the version number — in my case, `v6.9.1`.

  The Node Package Manager (NPM) is automatically installed along with Node. Type `npm -v` to see its version number. NPM is used in the background when you install packages from the public NPM repository. A common NPM command is `npm install`, which is used to download the package versions listed in your Angular project's package.json file.
- To install Angular CLI:
  1. Run `npm install -g angular-cli@1.0.0-beta.21` to install the version (still in beta at the time of writing) that I used for the sample application. (If you want to try a different build, visit the CLI site.) Installation takes about 10 minutes to complete.
  2. After successful installation, type `ng -v` at the OS command line to see your CLI version number — in my case:

```
angular-cli: 1.0.0-beta.21
node: 6.9.1
os: win32 x64
```

If you need help on `ng` syntax, type `ng help` at your OS command line.

The package.json file

The package.json file — a key metadata file in an Angular application — contains the details of the application and its dependent packages. This file is the most important one in an Angular application, especially when you move your code from one computer to another, and during upgrades. The package.json file controls the version of the packages that need to be installed.

You don't need to worry about the package.json for this tutorial's exercise. But when you develop a production-level application, you must pay more attention to the version numbers that are listed in this file.

Here are some valid statements from a package.json file:

```
{ "dependencies" :
  { "foo" : "1.0.0 - 2.9999.9999"
  , "bar" : ">=1.0.2 <2.1.2"
  , "baz" : ">1.0.2 <=2.3.4"
  , "boo" : "2.0.1"
  , "qux" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
  , "til" : "^1.2"
  , "elf" : "~1.2.3"
  , "two" : "2.x"
  , "thr" : "3.3.x"
  , "lat" : "latest"
  }
}
```

The `npm install` command interprets the portion of the statement to right of the colon as follows (where *version* stands for the version number used):

- "*version*": Must match *version* exactly
- ">*version*": Must be greater than *version*
- "~*version*": Approximately equivalent to *version*
- "^*version*": Compatible with *version*
- "1.2.x": 1.2.0, 1.2.1, and so on, but not 1.3.0
- "*" : Matches any *version*
- "" (empty string)": Same as `*`
- "*version1* - *version2*": Same as ">=*version1* <=*version2*"

To learn more about package.json, type `npm help package.json` at the OS command line to see the help contents in your browser.

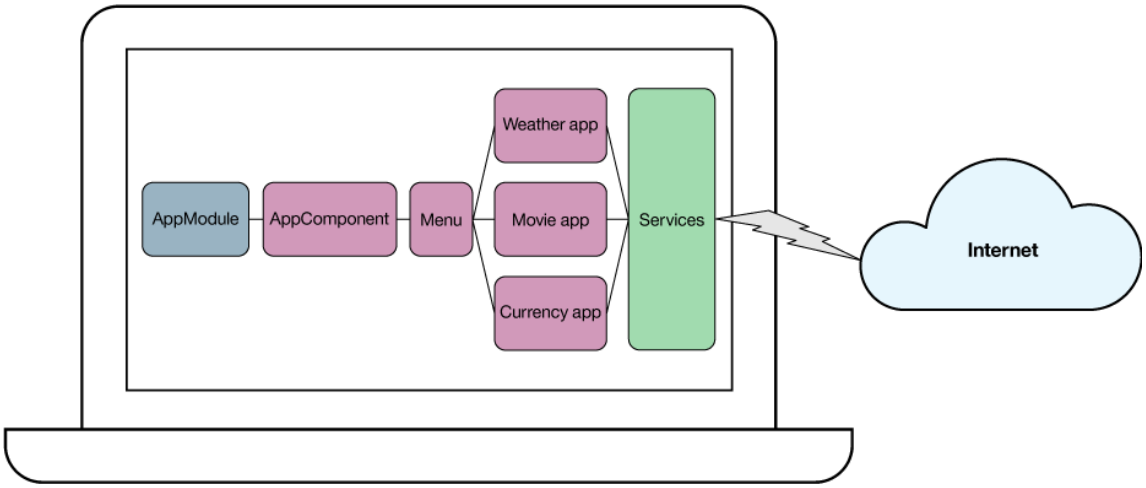
Sample project overview

The sample project consists of an out-of-the-box Angular application and a custom application that you'll develop on top of the out-of-the-box application. When you're finished, you'll have an Angular application consisting of three mini applications with features that use three web service APIs:

- Weather from Yahoo!
- Currency exchange
- Movie details

All of the application logic will run in your browser. The server is needed only when the browser needs new data. In fact, you can shut down the server process and still work in your application because it's a SPA.

This diagram shows the application topology:



You'll use Angular CLI to create your project (which by default will include `AppModule` and `AppComponent`) and four custom components:

- Menu component
- Weather component
- Movie component
- Currency component

You'll create routing for menu navigation, and you'll inject the following services into the weather, movie, and currency components:

- Data coming from the HTTP that consumes microservices
- Resource sharing across components while the services are used

Creating the base application and module

Ready to begin? Start at your OS command line at a location where you want to put your project directory.

Creating an Angular project

Generate a new Angular project by running the following command (where `dw_ng2_app` is the project name):

```
ng new dw_ng2_app --skip-git
```

After all the required packages and the Angular base application are installed (which takes about 10 minutes), you're back at your OS command prompt. If you then list the /dw_ng2_app directory, you can see the project structure:

```
|— e2e
|— node_modules
|— src
angular-cli.json
karma.conf.js
package.json
protractor.conf.js
README.md
tslint.json
```

The ../dw_ng2_app/src directory's contents are:

```
|— app
|— assets
|— environments
favicon.ico
index.html
main.ts
polyfills.ts
styles.css
test.ts
tsconfig.json
typings.d.ts
```

And the ../dw_ng2_app/src/app directory (the *root module folder*) contains the following files:

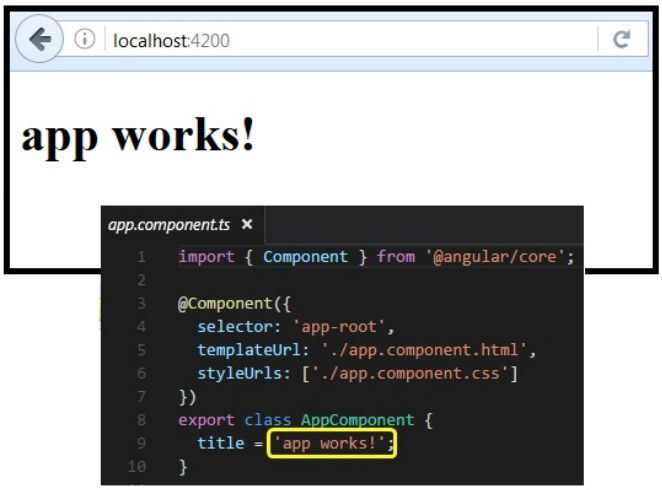
```
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts
index.ts
```

Running the out-of-the-box Angular application

Change to the project directory and run `ng serve` to start the out-of-the box Angular application.

By default, the process starts in port number 4200. If the value of your `port` system environment variable is other than 4200, the process will start in that port number. Optionally, you can override the default port number by running the `ng serve --port 4200` command.

Open your browser and enter the URL `http://localhost:4200/`. Your Angular application displays **app works!** to indicate that the app is up, running, and ready:



If you make changes to the code while the application is running, Angular is smart enough to monitor and restart the application automatically. Try editing the app.component.ts file by changing the value of the `title`. You can see that your browser page reflects the change:

How a module is linked to a component

In Listing 1, line 20 shows the declaration of the `AppModule` module.

Listing 1. app.module.ts

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

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

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

This module contains only one component —`AppComponent`— as shown on line 10. Line 18 says that the first component that gets started under the bootstrapping process is `AppComponent`.

Inside a component

Listing 2 shows what main application component looks like in the app.component.ts file.

Listing 2. app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'My Angular 2 app works!';
}
```

And here's what main application component looks like in the app.component.html file:

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

Metadata location

Metadata tells Angular how to process a class. In fact, `AppComponent` isn't a component until you tell Angular about it by attaching metadata to the class.

You attach metadata by using the `@Component` decorator, which identifies the class as a component. The `@Component` decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

The code in Listing 2 uses three options from among the available `@Component` configuration options:

- `selector`: CSS selector that tells Angular to create and insert an instance of this component where it finds the `selector` tag in the parent HTML file
- `templateUrl`: Component's HTML file
- `styleUrls`: Component's style sheets, such as .css files

Location of the template inside the component

A template is a form of HTML that tells Angular how to render the component. On line 5 of Listing 2, `templateUrl` points to a view named `app.component.html`.

Data binding inside the component

Data binding is a form of HTML that tells Angular how to render the component. In the app.component.ts file, the value of `title` is set inside the class and is used in the app.component.html file. Data binding can be one-way or two-way. In this case, the mapping is one-way if you mention the variable inside double curly braces `{{ }}`. The value is passed from the class to the HTML file.

Creating custom components and routes

At this point your Angular application is ready and working. This base application has a module, a component, a class, a template, metadata, and data binding — but it still lacks four other important pieces:

- Multiple components
- Routing
- Services
- Consumption of microservices

Next, you'll create the custom components.

Creating custom components

Stop the Angular process by pressing Ctrl-C (make sure you're in the directory of your Angular project, dw_ng2_app in this case). From your command prompt, run the following commands:

- `ng g c Menu -is --spec false --flat`: Creates the `Menu` component inside the `AppModule` root module, in the same folder.
- `ng g c Weather -is --spec false`: Creates the `Weather` component inside the `AppModule` root module in a subfolder named weather.
- `ng g c Currency -is --spec false`: Creates the `Currency` component inside the `AppModule` root module in a subfolder named currency.
- `ng g c Movie -is --spec false`: Creates the `Movie` component inside `AppModule` root module in a subfolder named movie.

Now, with the new components created — including classes, metadata, and templates — you can see how `AppModule` links to those components. In Listing 3, line 28 contains the declaration of the `AppModule` module. This module contains five components, including the root component and the other four components, as shown in lines 14–18.

Listing 3. app.module.ts

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

import { AppComponent } from './app.component';
import { MenuComponent } from './menu.component';
import { WeatherComponent } from './weather/weather.component';
import { CurrencyComponent } from './currency/currency.component';
import { MovieComponent } from './movie/movie.component';

@NgModule({
  declarations: [
    AppComponent,
    MenuComponent,
    WeatherComponent,
    CurrencyComponent,
    MovieComponent
  ],
  imports: [
    BrowserModule,
```

```
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Creating the routes

Route-creation command

I'm providing instructions here for manual route creation. As of this writing, a CLI command for route creation is under development. You can check the CLI site to see if it's now available.

For Angular to navigate among components, you need to create *routing*. Overwrite the menu.component.html file with the contents of Listing 4 so that the HTML includes the correct menus for each component.

Listing 4. menu.component.html

```
<div class="row">
   <div class="col-xs-12">
    <ul class="nav nav-pills">
     <li routerLinkActive="active"> <a [routerLink]="['/weather']" >Weather</a></li>
     <li routerLinkActive="active"> <a [routerLink]="['/movie']" >Movie Details</a></li>
     <li routerLinkActive="active"> <a [routerLink]="['/currency']" >Currency Rates</a></li>
    </ul>
  </div>
 </div>
```

The code in Listing 4 provides a mapping between the GUI and the URL path. For example, when the user clicks the Movie Details button in the GUI, Angular knows that it needs to run as if the URL path is `http://localhost:4200/movie`.

Next, you'll map the URL paths to the components. In the same folder as the root module, create a config file called app.routing.ts and use the code in Listing 5 as its contents.

Listing 5. app.routing.ts

```
import { Routes, RouterModule } from '@angular/router';
import { CurrencyComponent } from "./currency/currency.component";
import { WeatherComponent } from "./weather/weather.component";
import { MovieComponent } from "./movie/movie.component";
const MAINMENU_ROUTES: Routes = [
    //full : makes sure the path is absolute path
    { path: '', redirectTo: '/weather', pathMatch: 'full' },
    { path: 'weather', component: WeatherComponent },
    { path: 'movie', component: MovieComponent },
    { path: 'currency', component: CurrencyComponent }
];
export const CONST_ROUTING = RouterModule.forRoot(MAINMENU_ROUTES);
```

In this case, if your URL relative path is `movie`, you instruct Angular to call the `MovieComponent` component. In other words, the relative path `movie` maps to the URL `http://localhost:4200/movie`.

Now you need to link this view to its parent component. Overwrite the app.component.html file contents with the following code:

```
<div class="container">
 <app-menu></app-menu>
 <hr>
 <router-outlet></router-outlet>
</div>
```

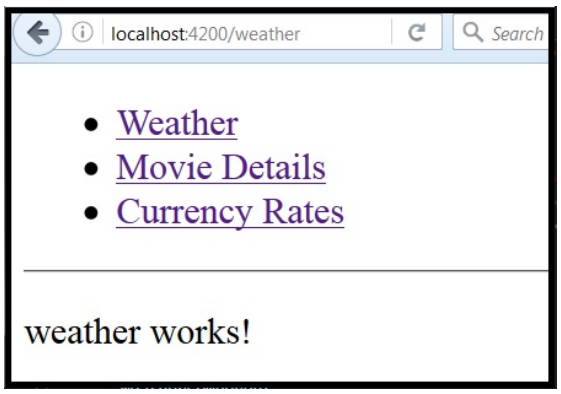
The `<app-menu></app-menu>` selector will contain the menu. The `<router-outlet></router-outlet>` selector is a placeholder for the current component. Depending on the URL path, the value can be any of the three components: weather, movie, or currency.

The module must also be notified about this route. Add two items in the app.module.ts file, as shown on lines 11 and 25 in Listing 6.

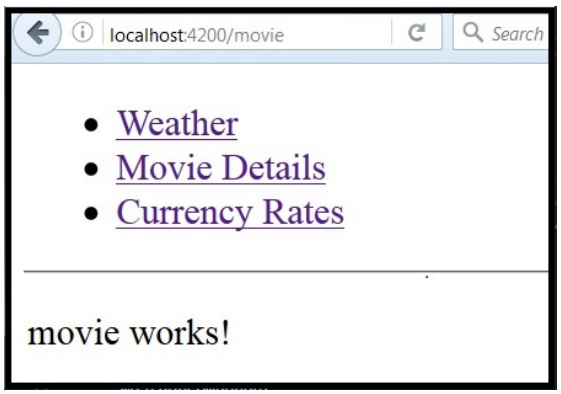
Listing 6. app.module.ts

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

import { AppComponent } from './app.component';
import { MenuComponent } from './menu.component';
import { WeatherComponent } from './weather/weather.component';
import { CurrencyComponent } from './currency/currency.component';
import { MovieComponent } from './movie/movie.component';
import { CONST_ROUTING } from './app.routing';

@NgModule({
  declarations: [
    AppComponent,
    MenuComponent,
    WeatherComponent,
    CurrencyComponent,
    MovieComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    CONST_ROUTING
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now, if you run the application and click the **Weather** link, the app displays **weather works!**:



If you click the **Movie Details** link, the app displays **movie works!**:



And if you click the **Currency Rates** link, the app displays **currency works!**:

You've successfully modified your Angular application to include multiple custom components and routing. Now you're ready for the last two important pieces:

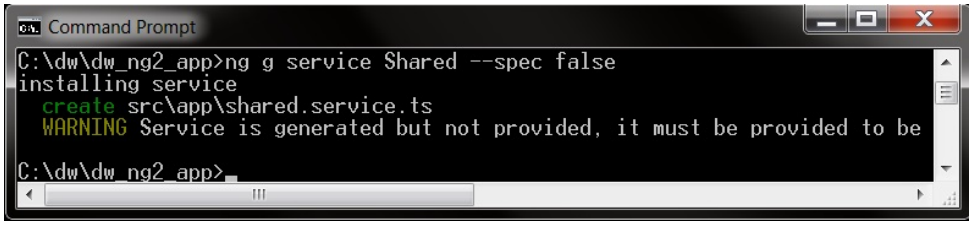
- Create and configure the service
- Consume the microservices

Creating the service

Stop the Angular process by pressing Ctrl-C. Run the following command:

```
ng g service Shared --spec false
```

This command creates the service in the shared.service.ts file, in the the root module folder:



Replace the contents of shared.service.ts with the code in Listing 7.

Listing 7. shared.service.ts

```
import { Injectable } from '@angular/core';
import { Http, Headers, Response } from "@angular/http";
import 'rxjs/Rx';
import { Observable } from "rxjs";

@Injectable()
export class SharedService {
    weatherURL1 = "https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20weather.forecast%20where
%20woeid%20in%20(select%20woeid%20from%20geo.places(1)%20where%20text%3D%22";
    weatherURL2 = "%2C%20";
    weatherURL3 = "%22)&format=json&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeys";
    findMovieURL1 = "http://www.omdbapi.com/?t=";
    findMovieURL2 = "&y=&plot=short&r=json";
    currencyURL = "http://api.fixer.io/latest?symbols=";
    totReqsMade: number = 0;
    constructor(private _http: Http) { }

    findWeather(city, state) {
        this.totReqsMade = this.totReqsMade + 1;
        return this._http.get(this.weatherURL1 + city + this.weatherURL2+ state + this.weatherURL3)
            .map(response => {
                { return response.json() };
            })
            .catch(error => Observable.throw(error.json()));
    }

    findMovie(movie) {
        this.totReqsMade = this.totReqsMade + 1;
        return this._http.get(this.findMovieURL1 + movie + this.findMovieURL2)
            .map(response => {
                { return response.json() };
            })
            .catch(error => Observable.throw(error.json().error));
    }

    getCurrencyExchRate(currency) {
        this.totReqsMade = this.totReqsMade + 1;
        return this._http.get(this.currencyURL + currency)
            .map(response => {
                { return response.json() };
            })
            .catch(error => Observable.throw(error.json()));
    }
}
```

The `import ...` statements in Listing 7 are a must for any service to work. The `@Injectable()` statement is especially important; it indicates that this `service` is injectable in other components — a technique commonly referred to as *dependency injection*.

The `totReqsMade` variable is declared here and will be used to pass the value across the three components. This will track the total number of service requests made to get the results of microservices.

You have three methods, with names that indicate their functionality: `findWeather()`, `findMovie()`, and `getCurrencyExchRate()`. During method execution, your Angular application will leave your browser to go out to the web to consume the microservices. Now you'll link components to the creates services.

Replace the movie.component.ts file contents with the code in Listing 8.

Listing 8. movie.component.ts

```
import { Component, OnInit } from '@angular/core';
import { SharedService } from "./../shared.service";

@Component({
  selector: 'app-movie',
  templateUrl: './movie.component.html',
  styles: []
})
export class MovieComponent implements OnInit {
    id_movie: string = "";
    mv_Title: string = "";
    mv_Rated: string = "";
    mv_Released: string = "";
    mv_Director: string = "";
    mv_Actors: string = "";
    mv_Plot: string = "";
    constructor(private _sharedService: SharedService) {
    }

    ngOnInit() {
    }

    callMovieService() {
        this._sharedService.findMovie(this.id_movie)
            .subscribe(
            lstresult => {
                this.mv_Title = lstresult["Title"];
                this.mv_Rated = lstresult["Rated"];

                this.mv_Released = lstresult["Released"];
                this.mv_Director = lstresult["Director"];
                this.mv_Actors = lstresult["Actors"];
                this.mv_Plot = lstresult["Plot"];
            },
            error => {
                console.log("Error. The findMovie result JSON value is as follows:");
                console.log(error);
            }
            );
    }
}
```

This important piece of code calls the service method to get the new data. In this case it is calling `callMovieService()` and then calls the `this._sharedService.findMovie()` method.

Similarly, replace the currency.component.ts file contents with the code in Listing 9.

Listing 9. currency.component.ts

```
import { Component, OnInit } from '@angular/core';
import { SharedService } from "./../shared.service";

@Component({
  selector: 'app-currency',
  templateUrl: './currency.component.html',
  styles: []
})
export class CurrencyComponent implements OnInit {

  id_currency: string = "";
  my_result: any;
  constructor(private _sharedService: SharedService) {
  }

  ngOnInit() {
  }

  callCurrencyService() {
    this._sharedService.getCurrencyExchRate(this.id_currency.toUpperCase())
      .subscribe(
      lstresult => {
                this.my_result = JSON.stringify(lstresult);
      },
      error => {
        console.log("Error. The callCurrencyService result JSON value is as follows:");
        console.log(error);
      }
      );
  }
}
```

And replace the weather.component.ts file contents with the code in Listing 10.

Listing 10. weather.component.ts

```
import { Component, OnInit } from '@angular/core';
import { SharedService } from "./../shared.service";

@Component({
  selector: 'app-weather',
  templateUrl: './weather.component.html',
  styles: []

})
export class WeatherComponent implements OnInit {
  id_city: string = "";
  id_state: string = "";
  op_city: string = "";
  op_region: string = "";
  op_country: string = "";
  op_date: string = "";
  op_text: string = "";
  op_temp: string = "";
  constructor(private _sharedService: SharedService) {
  }

  ngOnInit() {
  }

  callWeatherService() {
    this._sharedService.findWeather(this.id_city, this.id_state)
      .subscribe(
```

```
      lstresult => {
        this.op_city = lstresult["query"]["results"]["channel"]["location"]["city"];
        this.op_region = lstresult["query"]["results"]["channel"]["location"]["region"];
        this.op_country = lstresult["query"]["results"]["channel"]["location"]["country"];
        this.op_date = lstresult["query"]["results"]["channel"]["item"]["condition"]["date"];
        this.op_text = lstresult["query"]["results"]["channel"]["item"]["condition"]["text"];
        this.op_temp = lstresult["query"]["results"]["channel"]["item"]["condition"]["temp"];
      },
      error => {
        console.log("Error. The findWeather result JSON value is as follows:");
        console.log(error);
      }
      );
  }
}
```

Now, update the module to include the services. Edit the app.module.ts file to include the two statements in lines 12 and 28 of Listing 11.

Listing 11. app.module.ts

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

import { AppComponent } from './app.component';
import { MenuComponent } from './menu.component';
import { WeatherComponent } from './weather/weather.component';
import { CurrencyComponent } from './currency/currency.component';
import { MovieComponent } from './movie/movie.component';
import { CONST_ROUTING } from './app.routing';
import { SharedService } from "./shared.service";

@NgModule({
  declarations: [
    AppComponent,
    MenuComponent,
    WeatherComponent,
    CurrencyComponent,
    MovieComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    CONST_ROUTING
  ],
  providers: [SharedService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Modifying the components view

Now comes the final piece of the puzzle. You need to instruct the HTML file to call the right service methods. To do so, replace the movie.component.html file's contents with the code in Listing 12.

Listing 12. movie.component.html

```
<h2>Open Movie Database</h2>
<div class="col-md-8 col-md-offset-2">
```

```
 <div class="form-group">
  <input type="text" required [(ngModel)]="id_movie" (change)="callMovieService()" class="form-control"
 placeholder="Enter Movie name ...">
  <br><br>
  <h3>Movie Details</h3>
  <br>
  <p class="well lead">
      <i> Title :</i> {{ this.mv_Title }} <br>
      <i> Plot :</i> {{ this.mv_Plot }} <br>
      <i> Actors :</i> {{ this.mv_Actors }} <br>
      <i> Directed by :</i> {{ this.mv_Director }} <br>
      <i> Rated :</i> {{ this.mv_Rated }} <br>
      <i> Release Date :</i> {{ this.mv_Released }} <br>
  </p>
  <p class="text-info">Total # of all the service requests including Weather, Movie, and Currency is :
      <span class="badge">{{this._sharedService.totReqsMade}}</span>
  </p>
 </div>
</div>
```

Several important things are coded in movie.component.html:

- `{{ this._sharedService.totReqsMade }}`: This is the value that's tracked at the service level and shares its values across all three application components.
- `[(ngModel)]="id_movie"`: The user-entered GUI input is passed to the class that calls this HTML. In this case, the class is `MovieComponent`.
- `(change)="callMovieService()"`: Whenever this field value is changed, you instruct system to invoke the `callMovieService()` method that's present in the movie.component.ts file.
- `{{ this.mv_Title }}, {{ this.mv_Plot }}, {{ this.mv_Actors }}, {{ this.mv_Director }}, {{ this.mv_Rated }}, {{ this.mv_Released }}`: Displays results from the service calls that are made from `callMovieService() -> this._sharedService.findMovie(this.id_movie)`.

Replace the weather.component.html file's contents with the code in Listing 13.

Listing 13. weather.component.html

```
<h2>Yahoo! Weather </h2>
<div class="col-md-8 col-md-offset-2">
 <div class="form-group">
  <input type="text" [(ngModel)]="id_city" class="form-control" placeholder="Enter City name ..."><br>
  <input type="text" [(ngModel)]="id_state" class="form-control" placeholder="Enter State. Example CA for
 California ..."><br>
  <button type="button" class="btn btn-primary" (click)="callWeatherService()">Submit</button>
  <br><br><br>
  <br>
  <p class="well lead">
    <i>City, State, Country :</i> {{ this.op_city }} {{ this.op_region }} {{ this.op_country }} <br>
    <i>Current Condition :</i> {{ this.op_text }} <br>
    <i>Current Temperature :</i> {{ this.op_temp }} <br>
  </p>
  <p class="text-info">Total # of all the service requests including Weather, Movie, and Currency is :
    <span class="badge">{{this._sharedService.totReqsMade}}</span>
  </p>
 </div>
</div>
```

Finally, replace the currency.component.html file's contents with the code in Listing 14.

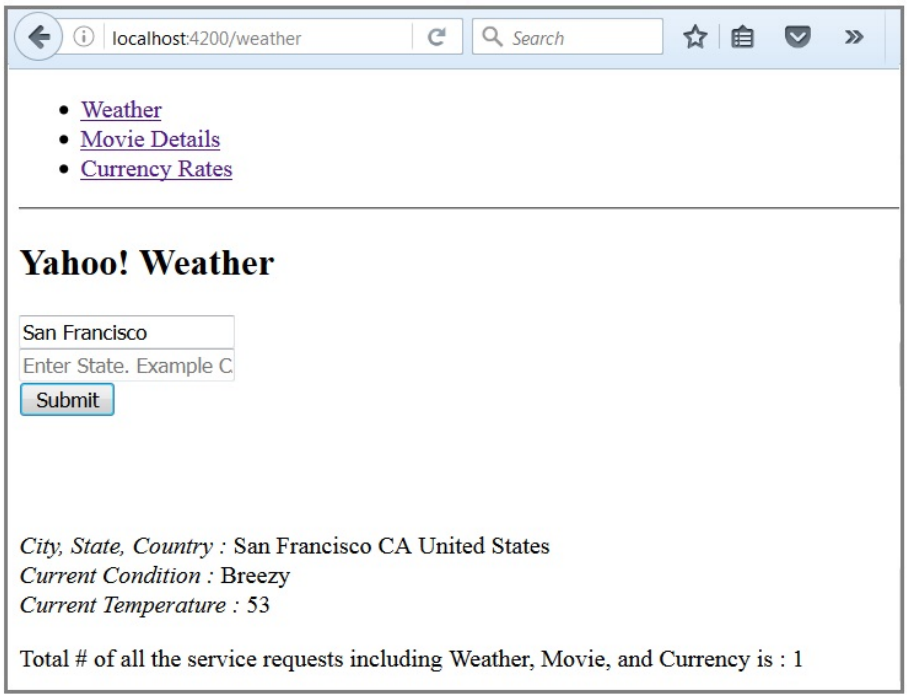
Listing 14. currency.component.html

```
<h2>Currency Exchange Rates</h2>
<div class="col-md-8 col-md-offset-2">
 <div class="form-group">
  <input type="text" [(ngModel)]="id_currency" (change)="callCurrencyService()" class="form-control"
 placeholder="Enter Currency Symbol. Example: GBP(,AUD,INR)...">
  <br><br>
  <h3>Rate Details</h3>
  <br>
  <p class="well lead">Exchange rate relative to Euro in a JSON format: : {{ this.my_result }} </p>
  <p class="text-info">Total # of all the service requests including Weather, Movie, and Currency is :
    <span class="badge">{{this._sharedService.totReqsMade}}</span>
  </p>
 </div>
</div>
```

Now, if all went as expected, the application can accept user input in the browser.

Running the app and improving the UI

Run the application now, enter some values, and view the results. For example, click the Weather link and enter `San Francisco` to see that city's weather conditions:



Test your new Angular skills

Currently, your application's input fields don't implement validation or error handling. On your own, you can try to add these features. Hint: Add methods in the service called **validateMovie(movie-name)**, **validateCurrency(currency-name)**, **validateCity(city-name)**, and **validateState(state-name)** Then call those methods from the corresponding components.

Everything is working well, but the UI could be more appealing. One way to make the GUI nicer is to use Bootstrap. (Angular 2 Material would be an ideal choice, but as of this writing it hasn't been officially released).

Go to the Bootstrap Getting started page and copy the following two lines from that page to your clipboard:

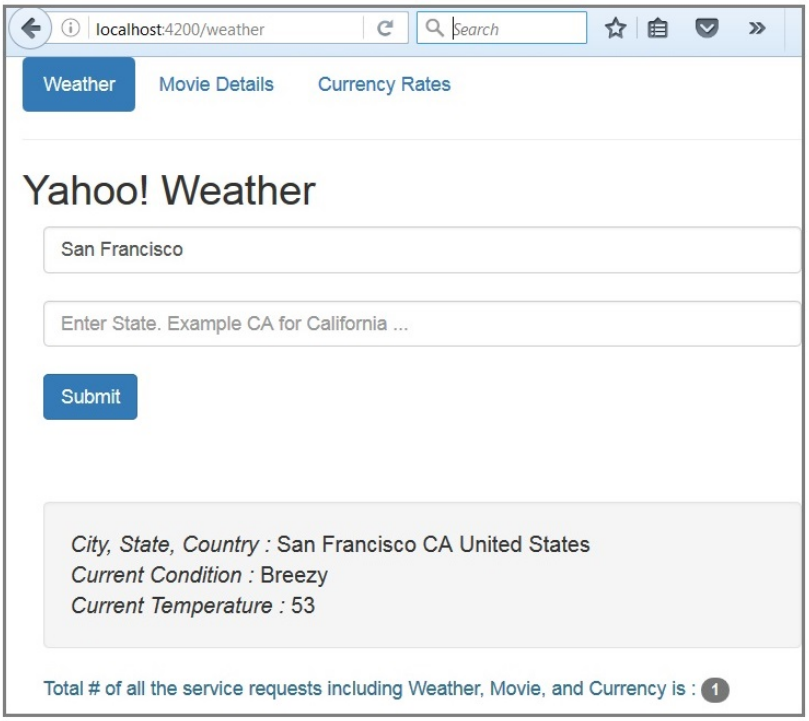
```
<!-- Latest compiled and minified CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
 integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
```

Open the index.html file and paste the statement that you just copied below line 8.

Listing 15. index.html

```
 <!doctype html>
<html>

<head>
  <meta charset="utf-8">
  <title>DwNg2App</title>
  <base href="/">

  <!-- Latest compiled and minified CSS -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
 integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>

<body>
  <app-root>Loading...</app-root>
</body>

</html>
```

Now the app looks better in a browser, with more readable styles and menu buttons instead of links for Weather, Movie Details, and Currency Rates:



Nested index.html

Take a minute now to appreciate why the application that you just finished is rightly called a SPA.

When the Angular application is started, the server sends the index.html file to the browser, and index.html is the only file that the browser displays. Anything Angular does to the page is inserted into this view:

The `<app-root>` selector at the end of index.html is replaced by the contents of app.component.html. In turn, app.component.html contains two selectors: `<app-menu>` and `<router-outlet>`. The `<app-menu>` selector is filled up with the contents of menu.component.html, and `<router-outlet>` is filled up dynamically depending on the menu selection — that is, with the contents of weather.component.html, currency.component.html, or movie.component.html.

All of the selectors are static except for the Angular reserved selector `<router-outlet></router-outlet>`. This selector is filled during runtime, depending on the router value. Only index.html is displayed, and all of the other HTML files that you coded are nested inside the index.html file.

Simulating the server

Your Angular project is successfully running in a development computer. If you have access to a remote sandbox server, you can move the code there to see how the app behaves when a user runs it. (Otherwise, you can skip to the tutorial Conclusion.)

Make sure that Node.js and Angular CLI are installed in the remote sandbox. Compress everything in your local project folder except the node_modules directory and its contents. Copy the compressed project file to the sandbox server and decompress it. Go to the server directory that contains package.json and run the `npm install` command. The package.json file enables the `npm install` command to go to the NPM public repository and install all the required package versions. Running this command also automatically creates the node_modules directory and its contents on the server.

Run the `ng serve` command to start the application in the sandbox server, as you did on your development computer. Press Ctrl-C to stop the process. Again, if you want to learn other options on the `ng serve`, run the `ng help` command.

Run the application with the `ng serve --port sandbox-port# --host sandbox-hostname` command.

Now the Angular application is available at the URL http://*sandbox-hostname*:*sandbox-port#*. While you run the app at that URL in your development computer browser, stop the sandbox server Angular process by pressing Ctrl-C on the server. Notice that the entire application is running in the development computer's browser even though the server Angular process is down. This tells you that SPA technology is in action. Once the application is in the browser, the control never goes to the server unless the application needs new data.

In the world of the SPA, browsers are the new servers. If 10 users are running the app, 10 browsers are handling the load. If 1,000 are running the app, 1,000 browsers are handling the load. The entire Angular application is under the control of the browser except for any logic — such as authentication and database operations — that's running in the server.

Better performance and reduction in server stress can ultimately improve the user experience and user satisfaction — keys to any business success.

“ In the world of the SPA, browsers are the new servers. ”

Conclusion

You've learned how to code and run a SPA using Angular 2 in your development computer and sandbox server. For production requirements, consult with your IT department. One of main pieces that most production applications have is authentication and authorization, which should be handled in the server mainly for security reasons. You'll likely need a dedicated server to handle these operations. For that server, you could use Node.js, which can act as a server that runs on the back end while Angular 2 runs on the front end. (Both Node and Angular originated from Google, so they work well together.)

Other techniques that you can consider using to improve application performance include:

- Bundling: A process that combines many of your programs into a single file.
- Minifying: Compressing the bundled file to keep the project size to a minimum.
- Ahead-of-time (AoT) compilation: The server takes care of compilation ahead of time during the build process, instead of the browser doing Just-in-time (JIT) compilation during runtime.

Acknowledgments

I want to thank the wonderful IBMers Guy Huinen, Dean P Cummings, Don Turner, and Mark Shade for their review effort and support.

Downloadable resources

Description	Name	Size
Sample app for sandbox	dw_ng2_app_for_sandbox.zip	18KB

Related topics

- Mastering MEAN
- Modularize Angular applications with webpack
- Angular architecture overview
- Tutorial: Tour of heroes
- Explore Angular resources
- Component interaction