Contents

[Mostly Review 1](#_Toc471705192)

[Email 1](#_Toc471705193)

[SendGrid 1](#_Toc471705194)

[Angular 2 3](#_Toc471705195)

[Angular 2 Helloworld 3](#_Toc471705196)

[Angular 2 Helloworld with .NET 8](#_Toc471705197)

[Angular2 Routing 10](#_Toc471705198)

[Modules 11](#_Toc471705199)

[TemplateUrl 13](#_Toc471705200)

[Angular2 Routing with .NET 13](#_Toc471705201)

[Asynchronous Calls Using Observables 15](#_Toc471705202)

# Submission

Please upload this document to the drop box by 4PM January 11.

# Mostly Review

Now that we are back from the break, this document mostly offers a review of some Angular and ASP.NET topics that we covered in the latter part of the fall. You will have seen a lot of this material before but it is organized this time to blend frameworks and to use of JSON in an increasingly dynamic manner.

## Email

Email is practically the only really new topic in this document. Setting up email with a .NET application is easy to do. The code is generally the same wherever you run it but each host may require slightly different configurations. The code below is similar to emailer code that I have used with several different email and web hosts over the years. You can also expect your code will be similar to the example provided here regardless of whatever email platform you use.

### SendGrid

SendGrid is an email service provider that you can use from any host.

https://app.sendgrid.com/signup

I like SendGrid because you can email up to 12,000 times per month free. It seems almost too good to be true – I don’t know where the catch is but I have used it without issue for about a year. Since I don’t send very many emails from my web applications I have not had to pay anything from my SendGrid account. I have not explored their commercial options but I am very happy with the free service and it has been completely reliable.

Example 1: Sending Email via SendGrid

First, a SendGrid account is required. It is easy to sign-up. After, create a console application and place the code below inside it. This code sample is actually taken from <https://sendgrid.com/docs/Integrate/Code_Examples/v2_Mail/csharp.html>

Then, modify the “To” and “From” addresses. You will also have to replace the email and password with the ones you use to sign into SendGrid.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Text;  using System.Net.Mail;  using System.Net.Mime;  namespace SmtpMail {  class Program {  static void Main() {  try {  MailMessage mailMsg = new MailMessage();  // To  mailMsg.To.Add(new MailAddress("emailaddress@home.com", "To Name"));  // From  mailMsg.From = new MailAddress("emailaddress@home.com", "From Name");  // Subject and multipart/alternative Body  mailMsg.Subject = "subject";  string text = "text body";  string html = @"<p>html body</p>";  mailMsg.AlternateViews.Add(  AlternateView.CreateAlternateViewFromString(text,  null, MediaTypeNames.Text.Plain));  mailMsg.AlternateViews.Add(  AlternateView.CreateAlternateViewFromString(html,  null, MediaTypeNames.Text.Html));  // Init SmtpClient and send  SmtpClient smtpClient  = new SmtpClient("smtp.sendgrid.net", Convert.ToInt32(587));  System.Net.NetworkCredential credentials  = new System.Net.NetworkCredential("loginEmail@home.com",  "YourSendGridPassword");  smtpClient.Credentials = credentials;  smtpClient.Send(mailMsg);  }  catch (Exception ex) {  Console.WriteLine(ex.Message);  }  Console.WriteLine("Done!");  Console.ReadLine();  }  }  } |

After running the program, check your email! If you move this code into an ASP.NET application remember to remove the Console.ReadLine() and Console.WriteLine() instructions.

# Angular 2

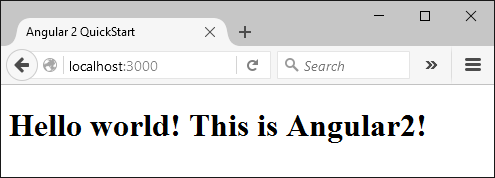
At this point I would understand if you find Angular 2 is only vaguely familiar. Maybe you didn’t like it. Or maybe you loved it. Whatever the case, it is important that you can use a \*big\* JavaScript framework just in case your employer demands it. Or, maybe your employer will ask you to use an easier and smaller framework but with the Angular 2 experience you will be prepared to make the shift in either direction.

## Angular 2 Helloworld

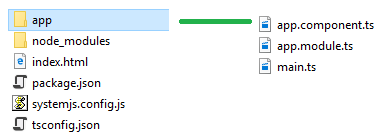
Just for a refresher, here is the Angular 2 helloworld with some basic set-up to get us started. I hope this is not too slow for you but for many of us I think we will need to go over the steps on how to set it up again.

Example 2: AngularJS 2 Hello World!

□ This example shows how to manually build a simple Angular2 application. Once you get it running it should like the following:



This is the file structure once you get it set up.



I am using the three configuration files which are package.json, system.config.js, and tsconfig.json. They are taken from the Angular team’s Angular Quickstart tutorial at

https://github.com/angular/quickstart

The AngularJS team uses the Node package manager to download the required packages. Be sure to download and install a recent version of NodeJS otherwise your project will not build properly:

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

After installing a new version of NodeJS, create a root project folder in a location where you have full permission to read, write and execute instructions. Then, place this package.json file inside your root folder. The configuration here tells the Node package manager which packages to install.

**package.json**

|  |
| --- |
| {  "name": "angular-quickstart",  "version": "1.0.0",  "scripts": {  "start": "tsc && concurrently \"tsc -w\" \"lite-server\" ",  "lite": "lite-server",  "tsc": "tsc",  "tsc:w": "tsc -w"  },  "licenses": [  {  "type": "MIT",  "url": "https://github.com/angular/angular.io/blob/master/LICENSE"  }  ],  "dependencies": {  "@angular/common": "~2.1.1",  "@angular/compiler": "~2.1.1",  "@angular/core": "~2.1.1",  "@angular/forms": "~2.1.1",  "@angular/http": "~2.1.1",  "@angular/platform-browser": "~2.1.1",  "@angular/platform-browser-dynamic": "~2.1.1",  "@angular/router": "~3.1.1",  "@angular/upgrade": "~2.1.1",  "angular-in-memory-web-api": "~0.1.13",  "core-js": "^2.4.1",  "reflect-metadata": "^0.1.8",  "rxjs": "5.0.0-beta.12",  "systemjs": "0.19.39",  "zone.js": "^0.6.25"  },  "devDependencies": {  "@types/core-js": "^0.9.34",  "@types/node": "^6.0.45",  "concurrently": "^3.0.0",  "lite-server": "^2.2.2",  "typescript": "^2.0.3"  }  } |

While there are alternatives, the Angular 2 team has adopted TypeScript as the defacto standard to enable a better code editing and development experience. A TypeScript transpiler is needed to generate JavaScript files from the TypeScript files. To enable the transpilation add the following tsconfig.json file in the root folder of the project.

**tsconfig.json**

|  |
| --- |
| {  "compilerOptions": {  "target": "es5",  "module": "commonjs",  "moduleResolution": "node",  "sourceMap": true,  "emitDecoratorMetadata": true,  "experimentalDecorators": true,  "removeComments": false,  "noImplicitAny": false  }  } |

The systemjs.config.js file creates a map to find the necessary packages and root application module. This file is referenced later by the index.html file.

**systemjs.config.js**

|  |
| --- |
| /\*\*  \* System configuration for Angular samples  \* Adjust as necessary for your application needs.  \*/  (function (global) {  System.config({  "paths": {  // paths serve as alias  "npm:": "node\_modules/"  },  // map tells the System loader where to look for things  "map": {  // our app is within the app folder  "app": "app",  // angular bundles  "@angular/core": "npm:@angular/core/bundles/core.umd.js",  "@angular/common": "npm:@angular/common/bundles/common.umd.js",  "@angular/compiler": "npm:@angular/compiler/bundles/compiler.umd.js",  "@angular/platform-browser": "npm:@angular/platform-browser/bundles/platform-browser.umd.js",  "@angular/platform-browser-dynamic": "npm:@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js",  "@angular/http": "npm:@angular/http/bundles/http.umd.js",  "@angular/router": "npm:@angular/router/bundles/router.umd.js",  "@angular/forms": "npm:@angular/forms/bundles/forms.umd.js",  "@angular/upgrade": "npm:@angular/upgrade/bundles/upgrade.umd.js",  // other libraries  "rxjs": "npm:rxjs",  "angular-in-memory-web-api": "npm:angular-in-memory-web-api/bundles/in-memory-web-api.umd.js"  },  // packages tells the System loader how to load when no filename and/or no extension  "packages": {  "app": {  "main": "./main.js",  "defaultExtension": "js"  },  "rxjs": {  "defaultExtension": "js"  }  }  });  })(this); |

Now that the basic set up for an Angular application is in place it is time to start building our application. To keep our application separate from the rest of the Angular set up, create an **app** folder inside the project root folder.

For this example, the application’s starting component logic is defined within the *AppComponent* class. This class is defined in an export class so it can be used by other classes. The *AppComponent* class stores data in a *title* property. In the component section, a *template* option stores an inline view. Also, a *selector* option there defines a custom element so the component can be included in any HTML view. Notice here that the *title* property value is accessed within the template option inside double curly braces.

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template: `<h1>Hello world! {{title}}</h1>`  })  export class AppComponent {  public title = 'This is Angular2!';  } |

Angular modules group components and directives into libraries. Every application must have at least one module. In this case, our root module is named *AppModule*. The root module tells the application what to load and how to load it. For this case our starting component, AppComponent, is referenced by this root module.

**app/app.module.ts**

|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { FormsModule } from '@angular/forms';  import { AppComponent } from './app.component';  @NgModule({  imports: [  BrowserModule,  FormsModule  ],  declarations: [  AppComponent  ],  bootstrap: [AppComponent]  })  export class AppModule { } |

The code inside main.ts initializes the root module. Main.ts is referenced by the systemjs.config.js file which is referenced in index.html.

**app/main.ts**

|  |
| --- |
| import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { AppModule } from './app.module';  const platform = platformBrowserDynamic();  platform.bootstrapModule(AppModule); |

**index.html**

Finally, here is our index.html page which goes in the root folder above the app folder. There are three noteworthy sections in the document:

1. The first section loads the JavaScript libraries.
2. The second section loads the main file in the format that we specified.
3. The <my-app> tag loads the contents that we defined in app.component.

|  |
| --- |
| <html>  <head>  <title>Angular QuickStart</title>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1">  <!-- 1. Load libraries -->  <!-- Polyfill for older browsers -->  <script src="node\_modules/core-js/client/shim.min.js"></script>  <script src="node\_modules/zone.js/dist/zone.js"></script>  <script src="node\_modules/reflect-metadata/Reflect.js"></script>  <script src="node\_modules/systemjs/dist/system.src.js"></script>  <!-- 2. Configure SystemJS -->  <script src="systemjs.config.js"></script>  <script>  System.import('app').catch(function(err){ console.error(err); });  </script>  </head>  <!-- 3. Display the application -->  <body>  <my-app>Loading...</my-app>  </body>  </html> |

Next, open a command prompt as administrator or terminal and navigate to the project root folder. To load the Angular2 framework and necessary packages, run the command:

**npm install**

Once the packages are loaded, to launch your application in a browser, run the command:

**npm start**

Once the browser launches, the application will stay open and respond to changes as you update the code.

## Angular 2 Helloworld with .NET

You can run Angular2 in or alongside MVC views in a .NET project. For today let’s just look at how to get Angular2 working with an MVC project. A little later in this document we will look at getting and posting JSON from Angular from and to an MVC .NET project.

Example 3: Angular 2 Helloworld with .NET

This example explains the steps needed to create an MVC project that runs Angular 2. First, build an empty MVC project. Build the project and shut it down completely. Open the \*.proj file and add this <TypeScriptCompileBlocked> element inside the <PropertyGroup> element.

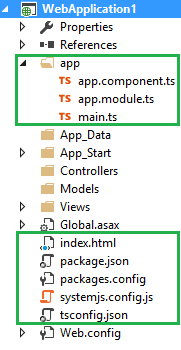
<PropertyGroup>

**<TypeScriptCompileBlocked>true</TypeScriptCompileBlocked>**

</PropertyGroup>

Then, right click the project node and add an **app** folder. After navigate inside the app folder found in the *0\_Angular2\_helloWorld.zip* package. Then drag and drop the typescript files from there into the new app folder. Now your typescript files should be visible inside the app folder.

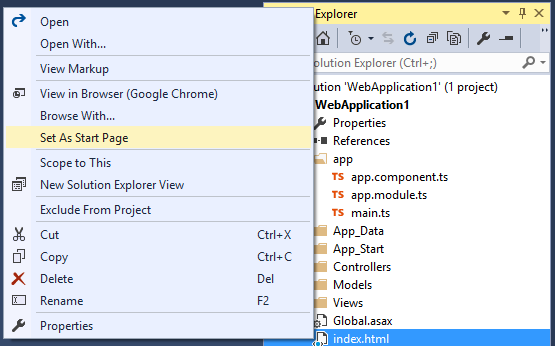
Next, drag and drop the index.html, package.json, tsconfig.json and systemjs.config.js files onto the project node. Your project should look like this:



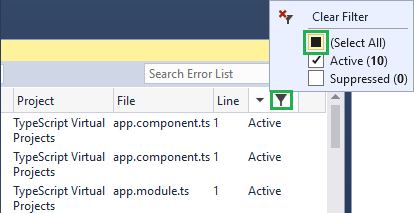
Next, re-open your solution. When you do, VS should start downloading the node\_modules packages for the project. If for some reason node\_modules does not download then navigate to the directory where package.json is with the command prompt and run the command:

**npm install**

While you wait for node\_modules to download, right click the index.html page and choose Set As Start Page.



Once the node\_modules are downloaded, in the error list click on the funnel icon which is the error filter and select all to prevent all typescript errors from showing.



Next, to compile your TypeScript files we will use npm. To do this, navigate to the **app** directory with the command prompt and type:

>**npm start**

This will compile your code and it will also launch the application in a browser. Make the sure application works in the browser. However, later we are going to want to test the application when it runs with .NET so to avoid confusion close the browser which runs at localhost:3000. If you leave it open it can get really confusing since you will want to focus on your .NET code later too. If successful, the HelloWorld page will show:



Exercise 1

In your .NET project, add a second property to the AppComponent class to store a message that says “Happy New Year 2017”. Modify the code in the template option to show this message with the rest of the original content. Make sure the change works properly and then show your revised **app.component.ts** file here:

|  |
| --- |
|  |

## Angular2 Routing

Routing lets us implement navigation in your Angular application so the proper component and child view are selected and displayed whenever a link is clicked.

Example 4: Basic Routing

This demonstration begins with the code inside **0\_Angular2\_ASPNET\_helloWorld.zip**. The code is then transformed to implement hyperlinks that retrieve and display different child views when clicked. You can find the final set of files in **1\_AngularSimpleRouting.zip**.

**app/app.page-a.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is page A.`  })  export class PageAComponent { } |

**app/app.page-b.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is another page.`  })  export class PageBComponent { } |

**app/app.pagedefault.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This page does not exist.`  })  export class PageDefault { } |

Next, replace the code inside app.component.ts file with the following code that contains a template which includes a menu with hyperlinks. The hyperlinks are created using the <routerLink> tag with a value of the path reference from the router. The <router-outlet> element serves as a placeholder for the child view which appears when the appropriate link is selected.

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template:  `<h1>This is the header</h1>  <nav>  <a routerLink="/page-a" routerLinkActive="active">A</a> |  <a routerLink="/page-b" routerLinkActive="active">B</a>  </nav>  <!-- Where router should display a view -->  <router-outlet></router-outlet>`  })  export class AppComponent { } |

With a larger project you may want to store the routing logic in a separate file such as in app.routing.ts. The appRoutes array below defines a series of routing objects which include a hyperlink reference and corresponding component displayed. Since the components are referenced by the router they must first be imported. These components have been loaded through the import statements.

**app/app.routing.ts**

|  |
| --- |
| import { ModuleWithProviders } from '@angular/core';  import { Routes, RouterModule } from '@angular/router';  import { AppComponent } from './app.component';  import { PageAComponent } from './app.page-a';  import { PageBComponent } from './app.page-b';  import { PageDefault } from './app.pagedefault';  const appRoutes: Routes = [  { path: 'page-a', component: PageAComponent },  { path: 'page-b', component: PageBComponent },  { path: '', redirectTo: '/page-a', pathMatch: 'full' },  { path: '\*\*', component: PageDefault }  ];  export const routing: ModuleWithProviders = RouterModule.forRoot(appRoutes); |

### Modules

Modules group components together. This not only helps to organize your code, the modularization also allows us to split the application up so it does not need to be loaded all at once.

In app.module.ts, we are declaring AppModule which includes four different components and we are also importing the router defined in the file previously discussed.

**app/app.module.ts**

|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { AppComponent } from './app.component';  import { PageDefault } from './app.pagedefault';  import { PageAComponent } from './app.page-a';  import { PageBComponent } from './app.page-b';  import { routing } from './app.routing';  @NgModule({  imports: [ BrowserModule, routing ],  declarations: [ AppComponent, PageDefault,  PageAComponent, PageBComponent ],  bootstrap: [ AppComponent ],  })  export class AppModule { } |

To specify a default target and reference for all tags in the Angular application, include this base tag inside the header of the index.html file.

**index.html**

|  |
| --- |
| <base href="/"> |

When you run the project it will allow you to navigate between pages.



Exercise 2

Add a page-c view to your solution for Example 4. Add a hyperlink to navigate to the new page. Test your application to ensure it works.

Show your app.page-c.ts file here:

|  |
| --- |
|  |

Show your revised app.component.ts here:

|  |
| --- |
|  |

Show your revised app.routing.ts here:

|  |
| --- |
|  |

Show your revised app/app.module.ts here:

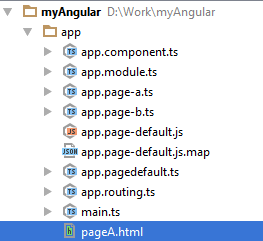
|  |
| --- |
|  |

### TemplateUrl

With routing enabled, it is now possible to separate your HTML into different pages.

Example 5: TemplateUrl

This example shows how to separate your HTML into different pages. To do this, begin with Example 4, then in the app directory create pageA.html.



Paste this code inside the HTML file.

|  |
| --- |
| <**H3**>This is page A.</**H3**> |

Then replace the contents of **app.page-a.ts** with the following code to reference the new HTML page from the app directory of the application.

|  |
| --- |
| **import** { Component } **from '@angular/core'**; @Component({  **templateUrl**:**'./app/pageA.html'** }) **export class** PageAComponent { } |

The output should appear as it did before in Example 4.

## Angular2 Routing with .NET

Example 6: Angular Routing with .NET

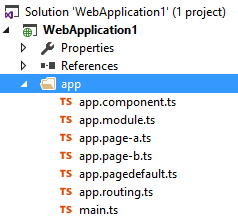
Now we are going to add in some Angular routing capability inside a .NET project. To do this, we will use the same files as Example 4. You can find these files in the package named **1\_AngularSimpleRouting.zip**. To begin, create an empty MVC project. Build the project so it saves and then shut it down completely. Open the \*.proj file and add this <TypeScriptCompileBlocked> element inside the <PropertyGroup> element – this tells the project not to try to compile the TypeScript.

<PropertyGroup>

**<TypeScriptCompileBlocked>true</TypeScriptCompileBlocked>**

</PropertyGroup>

Re-open the project and create an empty app folder in the project. Next, drag and drop the files from the *app* folder in 1\_AngularSimpleRouting.zip onto the *app* folder in the .NET project. Your app folder should now look like the following:



Then, drag and drop the configuration files and the index.html file in the root of the extracted zip archive onto the project node. Then, check and make sure this base tag exists in the head tag inside index.html:

|  |
| --- |
| <base href="/"> |

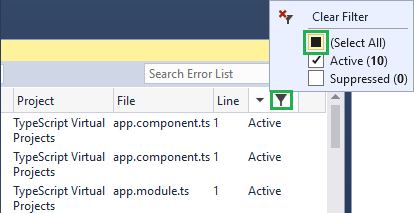
To tell the project to launch using index.html, right click the index.html page and choose **Set As Start Page**. Next, to ensure the project keeps returning back to the Angular application when pages are refreshed, add the following method to the class inside the Global.asax.cs file:

|  |
| --- |
| protected void Application\_BeginRequest(object sender, EventArgs e)  {  switch(Request.Url.AbsolutePath) {  case "/page-b":  case "/page-a":  Server.TransferRequest("index.html");  break;  }  } |

You need your node modules to download. Navigate to the directory where package.json is with the command prompt and run the command:

**npm install**

Once the node\_modules are downloaded, in the error list click on the funnel icon which is the error filter and select all to prevent all typescript errors from showing.



Next, to compile your TypeScript files we will use npm. To do this, navigate to the **app** directory with the command prompt and type:

>**npm start**

This will compile your code and it will also launch the application in a browser. Make the sure application works in the browser. However, remember to close the browser which runs at localhost:3000 to avoid confusing it for the application that will be running from the .NET server.

Now run your .NET project. When you run the project, it should display the different pages when selecting the different angular links. It will look similar to the output from Example 4: Basic Routing except it will be served from ASP.NET.

## Asynchronous Calls Using Observables

You should recognize the duplication in this next section but the refresher leads us back to the important topic of working with JSON and .NET from a JavaScript client.

Often, web applications will need to make requests for remote resources. Waiting for the resources can take several seconds and there is no guarantee the resource will arrive. There are several ways to manage this request so the call can be made on a separate thread and the resource can be received by the main thread in the application when the response arrives. One way to manage this is with observables. You can think of an Observable as a stream of events published by some source. We listen for events in this stream by subscribing to the Observable. In these subscriptions we specify the actions to take when the web request produces a success event or a fail event.

[https://angular.io/docs/ts/latest/guide/server-communication.html#!#http-client](https://angular.io/docs/ts/latest/guide/server-communication.html#!)

Example 7: Asynchronous Calls

To demonstrate how to use an observable, this example shows how to perform an asynchronous call for JSON. To begin, you could start the code in 0\_Angular2\_helloWorld.zip and add the following file.

**app/test.json**

|  |
| --- |
| [  {  "first": "Jane",  "last": "Chan"  },  {  "first": "Bill",  "last": "Good"  }  ] |

Next, the following code can be added to your application. This code imports a third party library, rxjs, which has been endorsed for managing observables by the AngularJS team. The code in app.mynameservice.ts is basically a template that can easily be modified to enable get, post, put or delete actions with a remote service. For this case, the Observable is set up to retrieve JSON from the test.json file using a get request. Also notice that we are passing a reference through the constructor to the Http class which has access to the get, post, put and delete functions. This reference is passed from the module a little later.

**app/app.mynameservice.ts**

|  |
| --- |
| import { Injectable } from '@angular/core';  import { Http, Response } from '@angular/http';  import { Observable } from 'rxjs/Observable';  import 'rxjs/add/operator/map';  import 'rxjs/add/operator/catch';  import 'rxjs/add/observable/throw';  @Injectable()  export class MyNameService {  private dataUrl = './app/test.json'; // URL to web API  constructor(private http: Http) { }  getNames(): Observable<string[]> {  return this.http.get(this.dataUrl)  .map(this.extractData)  .catch(this.handleError);  }  private extractData(res: Response) {  let body = res.json();  return body || {};  }  private handleError(error: any) {  // In a real world app, we might use a remote logging infrastructure  // We'd also dig deeper into the error to get a better message  let errMsg = (error.message) ? error.message :  error.status ? `${error.status} - ${error.statusText}` : 'Server error';  console.error(errMsg); // log to console instead  return Observable.throw(errMsg);  }  } |

Next, replace the code inside app.component.ts with the following code. This revised AppComponent creates an instance of the service through it’s constructor. The instance can be created in the constructor header with the help of the providers option. Note too how, since we are using a service that returns an Observable we can set up a subscription with handlers for the following cases:

1. Data is returned.
2. The request experiences an error.
3. A final block is entered regardless of whether 1 or 2 is selected.

**app/app.component.ts**

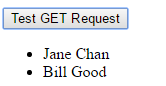
|  |
| --- |
| import { Component } from '@angular/core';  import { MyNameService } from './app.mynameservice';  // This component consumes the re-usable service.  @Component({  selector: 'my-app',  template: `<button (click)="getSomeData()">Test GET Request</button>  <ul>  <li \*ngFor="let myData of myNames">{{myData.first}} {{myData.last}}</li>  </ul>`,  // Providers allow us to inject an object instance through the constroctor.  // In this case we enable injection of MyDataService into AppComponent.  providers: [MyNameService]  })  export class AppComponent {  myNames: Array<any>;  \_myDataService: MyNameService;  // Since we are using a provider above we can receive  // an instance through an instructor.  constructor(myDataService: MyNameService) {  // Store local reference to MyDataService.  this.\_myDataService = myDataService;  }  getSomeData() {  this.\_myDataService.getNames()  // Subscribe to changes in the observable object  // that is returned by getRemoteData.  .subscribe(  // You basically get three handlers.  // 1. Handle successful data.  data => {  this.myNames = data  console.log(JSON.stringify(data))  },  // 2. Handle error.  error => {  alert(error)  },  // 3. Execute final instructions when successful.  () => {  console.log("Finished")  });  }  } |

To enable the HTTP get request we have to reference the HttpModule in our main module.

**app/app.module.ts**

|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { FormsModule } from '@angular/forms';  import { AppComponent } from './app.component';  import { HttpModule } from '@angular/http';  @NgModule({  imports: [  BrowserModule,  FormsModule,  HttpModule,  ],  declarations: [  AppComponent,  ],  bootstrap: [AppComponent]  })  export class AppModule { } |

The output from running the program after these changes is as follows:



Exercise 3

What is the role of the providers option in AppComponent?

|  |
| --- |
|  |

Exercise 4

Modify Example 7: Asynchronous Calls with the following changes and do not delete anything from the original solution for Example 7. Add the following JSON file to the project.

**app/car.json**

|  |
| --- |
| {  "make": "Toyota",  "model": "Prius"  } |

Next, add a second button to your component. Create an additional new function inside the service to make an asynchronous call to read this object. Then create a new function in your AppComponent class to call the new method and retrieve the data. Display the contents of this object after it is retrieved. Show your revised app.mynameservice.ts file here:

|  |
| --- |
|  |

Show your revised app.component.ts file here:

|  |
| --- |
|  |

Exercise 5

Run the code in **3\_Angular\_JSON\_From\_NETMVC.zip**. Explain how server side validation is implemented in the controller of the POST enabled method.

|  |
| --- |
|  |

Exercise 6

Run the code in **3\_Angular\_JSON\_From\_NETMVC.zip**. Set a break point at the last line of the GetNames() method in the NameRepository class. When halted, make sure the autos window is available. You can get there from the debug menu. Show a screenshot of the autos window which displays the JSON object that has been created.

|  |
| --- |
|  |

Exercise 7

Name the library that allows us to create ad hoc JSON objects in the code from **3\_Angular\_JSON\_From\_NETMVC.zip.**

|  |
| --- |
|  |

Exercise 8

When running the code from **3\_Angular\_JSON\_From\_NETMVC.zip**, take and show a screenshot of the Chrome debug window while halted at a breakpoint inside the last line of the extractData function inside app.nameservice.ts.

|  |
| --- |
|  |

Exercise 9

When running the code in **3\_Angular\_JSON\_From\_NETMVC.zip**, take and show a screenshot of the Chrome debug window while halted at the breakpoint inside the last line of the success block inside the getNames() function in the app.component.ts.

|  |
| --- |
|  |