**What is Webpack**

The dynamic web applications usually have lots of javascript files. These files are either created by you or it could be third party libraries like jquery/bootstrap etc. We include these in our **index.html** file using ***<script>*** tag. When a user sends requests to our application, the browser requests and loads these files one at a time. If you have lots of these files, then it will make your application slow. The solution to this problem is to merge all these files into a one or two files so that the browser can download the entire file in one request. This is where **Webpack** is used.

Webpack is a powerful module bundler, which scans your web application looking for javascript files and merges them into one ( or more) big file. Webpack has the ability to bundle any kind of file like JavaScript, CSS, SASS, LESS, images, HTML, & fonts etc. Webpack also comes with Development Server that supports hot module reloading.

**Angular Webpack Tutorial**

In this tutorial, we are going to build an example application which displays the “**Hello and welcome to Angular**” with the logo of Angular below it.

This Tutorial follows the official [Angular Webpack guide](https://v5.angular.io/guide/webpack)   We have used the codes found in that tutorial and made few changes where necessary. We also removed the Testing Framework to make it simpler.

**Prerequisites**

You need to install following before you proceed further with this tutorial

* Visual Studio Code (or any other editor of your choice)
* NPM Package Manager

You can read instruction on how to install from the tutorial [Installing and getting started with Angular](https://www.tektutorialshub.com/angular/angular-installing-settingup-development-environment/)

**Setting up an Angular Application**

The Setting up and angular Application requires the following steps

1. Create an Application folder
2. Create package.json configuration file
3. Installing Dependencies
4. Create the Component
5. Create the root module
6. Bootstrap our application
7. Create the index.html
8. Configure our Application
9. Run the application

**Create an Application Folder**

Open a command prompt and create the folder ***AngularWebpack***.

|  |  |
| --- | --- |
| 1  2  3  4 | md AngularWebpack  cd AngularWebpack |

**package.json Configuration file**

A **package.json** file contains the metadata about our application. It includes the list of Javascript libraries that are used by our application. The NPM package manager reads this file to install/update the libraries.

You can manually create the package.json file and run the command “npm install” to install the dependencies. In this tutorial, we will show it how to do it from the command prompt.

Run the following command to create the package.json file

|  |  |
| --- | --- |
| 1  2  3 | npm init -f |

Open the Visual Studio code and you can see that the following package.json file in the root folder.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | {      "name": "AngualrWebPack",      "version": "1.0.0",      "description": "",      "main": "index.js",      "scripts": {          "test": "echo \"Error: no test specified\" && exit 1"      },      "keywords": [],      "author": "",      "license": "ISC"  } |

**Installing Dependencies**

The next step is to download and install Angular & other libraries. We will use NPM package manager to install them.

**Installing Angular libraries**

Run the following command from the command prompt to install the Angular components.

|  |  |
| --- | --- |
| 1  2  3 | npm install @angular/common @angular/compiler @angular/core @angular/forms  @angular/http @angular/platform-browser @angular/platform-browser-dynamic  @angular/router --save |

The **–save** option ensures that these libraries are saved to package.json file

**Installing third party libraries**

The Angular requires you to install the following dependencies

|  |  |
| --- | --- |
| 1  2  3 | npm install core-js rxjs zone.js --save |

**Rxjs or Reactive Extensions (Rx)** is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators. Read the complete tutorial from [here](https://xgrommx.github.io/rx-book/index.html)

**Zone** is used for change detection. For more read [Zones in Angular](https://blog.thoughtram.io/angular/2016/02/01/zones-in-angular-2.html)

**Core-js** is a ES6 polyfill for typescript. You can read it from [here](https://github.com/zloirock/core-js)

**Installing Development Dependencies**

The development dependencies are those libraries, which required only to develop the application. For Example, javascript libraries for unit tests, minification, module bundles are required only at the time of development of the application.

Our Angular application needs Typescript. Webpack module, loaders,  plugins, and tools for testing the app etc.

**Typescript**

Typescript is a superset of Javascript. It extends the Javascript and brings options like static typing, classes, interfaces. The Code written in typescript cannot be used directly in the web browser. It must be compiled to javascript before running in the web browser. This process is known as Transpiling. The Transpiling is done by Typescript compiler (tsc). Read Complete [Typescript tutorial](https://www.typescriptlang.org/docs/tutorial.html)

|  |  |
| --- | --- |
| 1  2  3 | npm install typescript --save-dev |

**Typings**

Typescript forces you to define the types before using them. This has great advantages as any errors are detected at the compile time rather than at the run time.

But our application is going to use a lot of Third Party Javascript libraries in our application. Being written in Javascript, they do not have types defined in them. The typescript compiler will throw an error as it does not recognize them.

This where type definition files come into play. They provide the type information to these javascript libraries. We need to download type definition files for each of these libraries. That is done using the typings.

|  |  |
| --- | --- |
| 1  2  3 | npm install @types/node @types/jasmine --save-dev |

**Webpack**

Webpack along with Webpack dev server can be installed using the following command.

|  |  |
| --- | --- |
| 1  2  3 | npm install webpack webpack-dev-server webpack-merge --save-dev |

The **–save-dev** option ensures that these are installed as development dependencies

**Webpack loaders and plugins**

Webpack supports custom loaders and plugins. A loader is a program that allows you to preprocess files as you “load” them. They extract the content of the file, transform them and then return the transformed content to Webpack for bundling. With the help of leaders, the Webpack can handle any type of files.

You can read more about loaders from [here](https://webpack.js.org/concepts/loaders/)

Run the following command to install the required Webpack loaders

**Webpack loaders**

|  |  |
| --- | --- |
| 1  2  3 | npm install angular2-template-loader awesome-typescript-loader css-loader file-loader html-loader null-loader raw-loader style-loader to-string-loader --save-dev |

**Webpack plugins**

A plugin is a program that changes the behaviour of Webpack

|  |  |
| --- | --- |
| 1  2  3 | npm install html-webpack-plugin webpack-merge extract-text-webpack-plugin --save-dev |

**Others dependencies**

|  |  |
| --- | --- |
| 1  2  3 | npm install rimraf --save-dev |

**Testing tools**

|  |  |
| --- | --- |
| 1  2  3 | npm install jasmine-core karma karma-chrome-launcher karma-jasmine karma-sourcemap-loader karma-webpack --save-dev |

**Creating the Component**

So far we have installed all the required dependencies. The next step is to create our application. Under the root folder of our application create folder called **src**. Under src create a folder called **app**.

**Component class**

First, let us create an Angular Component. Create **app.component.ts** under the**src/app folder** and copy the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { Component } from '@angular/core';  import '../assets/css/styles.css';    @Component({      selector: 'my-app',      templateUrl: './app.component.html',      styleUrls: ['./app.component.css']  })    export class AppComponent { } |

The Component is the most important part of the angular. It controls the region of the screen or View. It consists of three main parts one is class, a class decorator, and an import statement

**Component class**

|  |  |
| --- | --- |
| 1  2  3 | export class AppComponent { } |

A component is a simple class. The class is defined using the Export keyword so that it can be used in other parts of the application. Our component class does not have any methods and properties. The component in real life will have methods and properties, which supply logic to our view.

**@Component decorator**

The AppComponent class is then, decorated with @Component decorator attribute. The @Component (called class decorator) provides Metadata about the component class. The Angular uses this Metadata to create the view

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Component({      selector: 'my-app',      templateUrl: './app.component.html',      styleUrls: ['./app.component.css']  }) |

The Metadata above has three fields. The **selector**, **templateURL** & **styleUrls**

**templateUrl**

The templateUrl contains the path to the HTML file. The Angular uses this HTML file to render the view. In the above example, it points to the app.component.html file. Hence we need to create it. Create **app.component.html** file under the **src/app folder**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <main>      <h1>Hello from Angular Application with Webpack</h1>      <img src="../assets/images/angular.png">  </main> |

**styleUrls**

The styleUrls is an array of Style Sheets that angular2 uses to style our HTML file. In the above example, it points towards to app.component.css style sheet. Create a file **app.component.css** under **src/app folder**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | main {      padding: 1em;      font-family: Arial, Helvetica, sans-serif;      text-align: center;      margin-top: 50px;      display: block;  } |

**Selector**

The selector tells angular, where to display the template. In the example above selector is “’my-app’”. The angular  whenever it encounters the above tag replaces it with the template

**Import statement**

|  |  |
| --- | --- |
| 1  2  3  4 | import { Component } from '@angular/core';  import '../assets/css/styles.css'; |

The import statement is used to import all the libraries that are used in our component class. This statement is similar to C# using statement. The @Component decorator is available in @angular/core module. Hence we need to refer it in our class. This is done using the import method as shown above.

**Root Module**

The Angular follows the modular approach, the application development. Every Angular application must have one module known as root Module. We will name it as **app.module.ts**. Create the file with the name **app.module.ts** under the **folder src/app** and add the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { AppComponent } from './app.component';  @NgModule({      imports: [          BrowserModule      ],      declarations: [          AppComponent      ],      bootstrap: [ AppComponent ]  })  export class AppModule { } |

The Angular module is similar to any other component class we created earlier. Like Component, it consists of three parts. A **class** , **class decorator** and **import statement**

**Module class**

|  |  |
| --- | --- |
| 1  2  3 | export class AppModule { } |

Similar to the component, the Module class is defined with the export keyword. Export class ensures that you can use this class in any other class

|  |  |
| --- | --- |
| 1  2  3 | @NgModule decorator |

We used a @component decorator to define our component. The modules require a @ngModule decorator. @ngModue decorator passed metadata about the module. & bootstrap. The @ngModule Metadata above has three fields. declarations, imports & bootstrap.

**Imports Metadata** tells the angular list of other modules used by this module.

**Declaration Metadata** lists the components, directives, services etc that are part of this module.

**Bootstrap Metadata** identifies the root component of the module. When Angular loads the appModule it looks for bootstrap Metadata and loads all the components listed here. We want our module to AppComponent , hence we have listed it here

**Bootstrapping our root module**

We have so far created AppComponent which is bound to the HTML template app.component.html. We have added the AppComponent to AppModule. In AppModule we indicated that the AppComponent is to be loaded when AppModule is loaded

The Next step is to ask the Angular to load the AppModule when the application is loaded. To do need to create main.ts file

Create **main.ts** in the **src folder** and copy the following content

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { enableProdMode } from '@angular/core';    import { AppModule } from './app/app.module';    if (process.env.ENV === 'production') {    enableProdMode();  }    platformBrowserDynamic().bootstrapModule(AppModule); |

First, we import platformBrowserDynamic library. This library contains all the functions required to bootstrap the angular application. Next, we import our AppModule. Finally, we invoke bootstrapModule method of platformBrowserDynamic library to bootstrap our AppModule

**Index page**

We need a root page for our application. Create index.html under **src folder**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <!DOCTYPE html>  <html>    <head>      <base href="/">      <title>Angular With Webpack</title>      <meta charset="UTF-8">      <meta name="viewport" content="width=device-width, initial-scale=1">    </head>    <body>      <my-app>Loading...</my-app>    </body>  </html> |

**Assets**

We have imported styles.css and used “angular.png” image in our AppComponent.

Create the folder **src/assets/css** and copy the **styles.css** to it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | body {      background: #0147A7;      color: #fff;  } |

Similarly, create the folder **assets/images** under src. Download [Angular Icon](https://www.tektutorialshub.com/wp-content/uploads/2016/10/angular.png) Icon and copy it there.

**Configuring Our Application**

We have successfully built our application. The next step is to run the application. But before that, we need to configure Typescript, Typings and Webpack libraries

**Typescript**

Create the file **tsconfig.json** in the**src folder** our project and copy the following

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | {    "compilerOptions": {      "target": "es5",      "module": "commonjs",      "moduleResolution": "node",      "sourceMap": true,      "emitDecoratorMetadata": true,      "experimentalDecorators": true,      "lib": [ "es2015", "dom" ],      "noImplicitAny": true,      "suppressImplicitAnyIndexErrors": true,      "typeRoots": [        "../node\_modules/@types/"      ]    },    "compileOnSave": true,    "exclude": [      "node\_modules/\*",      "\*\*/\*-aot.ts"    ]  } |

You can read https://www.typescriptlang.org/docs/handbook/tsconfig-json.html to learn about each of those compiler options.

**Webpack Bundle**

The next step is to configure the Webpack. Webpack allows us to bundle all our javascript files into a one or more files. Let us create three bundles in our application

In the first bundle, we add all our application code like components, service, modules etc. We call it as an app. We do not have to create a separate file to that. Our **main.ts** file will be the starting point for this bundle.

We put all the external libraries like Rxjs, Zone etc into a separate bundle. This includes Angular libraries also. Let us call it as the vendor. To do that we need to create the **vendor.ts** and import required libraries. Create the file called the vendor.ts under **src folder** and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // Angular  import '@angular/platform-browser';  import '@angular/platform-browser-dynamic';  import '@angular/core';  import '@angular/common';  import '@angular/http';  import '@angular/router';    // RxJS  import 'rxjs';    // Other vendors for example jQuery, Lodash or Bootstrap  // You can import js, ts, css, sass, ... |

In the third bundle, we include the polyfills we require to run Angular applications in most modern browsers. Create a file called **polyfills.ts** under the **src folder** and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import 'core-js/es6';  import 'core-js/es7/reflect';  require('zone.js/dist/zone');    if (process.env.ENV === 'production') {    // Production  } else {    // Development and test    Error['stackTraceLimit'] = Infinity;    require('zone.js/dist/long-stack-trace-zone');  } |

**Webpack configuration**

The next step is to configure the Webpack.

The Webpack by convention uses the **webpack.config.js** file to read the configuration information. Create the **webpack.config.js** in the **root folder** of our project. Add the following code

|  |  |
| --- | --- |
| 1  2  3 | module.exports = require('./config/webpack.dev.js'); |

The above code tells the Webpack to read the configuration file **webpack.dev.js** from the config folder.

The Webpack can be setup so that you can have a separate configuration option for testing, development, and production. What you need to do is to create separate config files for development. testing and production and then switch between this config file in the main configuration file (webpack.config.js)

Create the folder **“config”** in the root of our project. This is where we are going to put all over Webpack related configuration option

**Helper functions**

Create the file **helpers.js** under the **folder config**and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var path = require('path');    var \_root = path.resolve(\_\_dirname, '..');    function root(args) {    args = Array.prototype.slice.call(arguments, 0);    return path.join.apply(path, [\_root].concat(args));  }    exports.root = root; |

**Common Configurations**

Create the file **webpack.common.js** under the **folder config** and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | var webpack = require('webpack');  var HtmlWebpackPlugin = require('html-webpack-plugin');  var ExtractTextPlugin = require('extract-text-webpack-plugin');  var helpers = require('./helpers');    module.exports = {    entry: {      'polyfills': './src/polyfills.ts',      'vendor': './src/vendor.ts',      'app': './src/main.ts'    },      resolve: {      extensions: ['.ts', '.js']    },      module: {      rules: [        {          test: /\.ts$/,          loaders: [            {              loader: 'awesome-typescript-loader',              options: { configFileName: helpers.root('src', 'tsconfig.json') }            } , 'angular2-template-loader'          ]        },        {          test: /\.html$/,          loader: 'html-loader'        },        {          test: /\.(png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico)$/,          loader: 'file-loader?name=assets/[name].[hash].[ext]'        },        {          test: /\.css$/,          exclude: helpers.root('src', 'app'),          loader: ExtractTextPlugin.extract({ fallbackLoader: 'style-loader', loader: 'css-loader?sourceMap' })        },        {          test: /\.css$/,          include: helpers.root('src', 'app'),          loader: 'raw-loader'        }      ]    },      plugins: [      // Workaround for angular/angular#11580      new webpack.ContextReplacementPlugin(        // The (\\|\/) piece accounts for path separators in \*nix and Windows        /angular(\\|\/)core(\\|\/)@angular/,        helpers.root('./src'), // location of your src        {} // a map of your routes      ),        new webpack.optimize.CommonsChunkPlugin({        name: ['app', 'vendor', 'polyfills']      }),        new HtmlWebpackPlugin({        template: 'src/index.html'      })    ]  }; |

The webpack.common.js config will contain all the configuration settings, which are common to development, testing, and production builds.

Now let us look at each part

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | entry: {      'polyfills': './src/polyfills.ts',      'vendor': './src/vendor.ts',      'app': './src/main.ts'  }, |

First, we let Webpack know our entry points. Remember that we have decided to create three bundles of our application. Our three entry points are **polyfills.ts** , **vendor.ts**, and **main.ts** all located in the src folder.

The Webpack starts from these files and traverses through it to find dependencies and merges all of them one bundle per each entry.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | module: {      rules: [        {          test: /\.ts$/,          loaders: [            {              loader: 'awesome-typescript-loader',              options: { configFileName: helpers.root('src', 'tsconfig.json') }            } , 'angular2-template-loader'          ]        },        {          test: /\.html$/,          loader: 'html-loader'        },        {          test: /\.(png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico)$/,          loader: 'file-loader?name=assets/[name].[hash].[ext]'        },        {          test: /\.css$/,          exclude: helpers.root('src', 'app'),          loader: ExtractTextPlugin.extract({ fallbackLoader: 'style-loader', loader: 'css-loader?sourceMap' })        },        {          test: /\.css$/,          include: helpers.root('src', 'app'),          loader: 'raw-loader'        }      ]    }, |

Webpack then uses **leaders** to transform our files. For example, the Typescript files (ts extension) are passed through “angular2-template-loade” and then to “awesome-typescript-loader” (Right to left)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | plugins: [      // Workaround for angular/angular#11580      new webpack.ContextReplacementPlugin(        // The (\\|\/) piece accounts for path separators in \*nix and Windows        /angular(\\|\/)core(\\|\/)@angular/,        helpers.root('./src'), // location of your src        {} // a map of your routes      ),        new webpack.optimize.CommonsChunkPlugin({        name: ['app', 'vendor', 'polyfills']      }),        new HtmlWebpackPlugin({        template: 'src/index.html'      })    ] |

The *CommonsChunkPlugin* removes all the multiple used chunks of code and uses it only once.

The *HtmlWebpackPlugin* adds a script tag to our index.html for the each of the bundle created.

**Development specific configuration**

Create **webpack.dev.js** under the **config folder** and add the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | var webpackMerge = require('webpack-merge');  var ExtractTextPlugin = require('extract-text-webpack-plugin');  var commonConfig = require('./webpack.common.js');  var helpers = require('./helpers');    module.exports = webpackMerge(commonConfig, {    devtool: 'cheap-module-eval-source-map',      output: {      path: helpers.root('dist'),      publicPath: '/',      filename: '[name].js',      chunkFilename: '[id].chunk.js'    },      plugins: [      new ExtractTextPlugin('[name].css')    ],      devServer: {      historyApiFallback: true,      stats: 'minimal'    }  }); |

The **webpack.dev.js** file imports the **webpack.common.js** and uses additional configuration options required only for the development

|  |  |
| --- | --- |
| 1  2  3 | devtool: 'cheap-module-eval-source-map', |

The devtool defines how the source map is created. The source maps help in debugging our applications in the browser.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | output: {      path: path.resolve(rootDir, 'dist'),      publicPath: 'http://localhost:8080/',      filename: '[name].js',      chunkFilename: '[id].chunk.js'  }, |

Output configuration has options that affect the output of the Webpack compilation. You can configure location on disk where the compiled files are written to (path), the name of the bundle (filename), the name of the chunk file (chunkfilename) and public URL path (publicPath) etc.

|  |  |
| --- | --- |
| 1  2  3  4  5 | plugins: [      new ExtractTextPlugin('[name].css')  ], |

You call any development environment specific plugin here. The ***extract-text-***webpack***-plugin*** removes the compiled CSS from the bundle and emits is as a separate file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | devServer: {      historyApiFallback: true,      stats: 'minimal'  } |

The devServer is used to configure the behavior of web pack-dev-server

**Package.json**

Finally, Open the package.json and replace the scripts options with the code below

|  |  |
| --- | --- |
| 1  2  3  4  5 | "scripts": {       "start": "webpack-dev-server --inline --progress --port 8080"    }, |

**Run the Application**

From the command prompt type, **npm start** to run the application. Open the browser and type http://localhost:8080/. You should be able to see “Hello and welcome to Angular” message on the screen