Project Setup

Proper tooling and setup is good for any project, but it's especially important for Angular 2 due to all of the pieces that are involved. We've decided to use webpack, a powerful tool that attempts to handle our complex integrations. Due to the number of parts of our project that webpack touches, it's important to go over the configuration to get a good understanding of what gets generated client-side.

Webpack

A modern JavaScript web application includes a lot of different packages and dependencies, and it's important to have something that makes sense of it all in a simple way.

Angular 2 takes the approach of breaking your application apart into many different components, each of which can have several files. Separating application logic this way is good for the programmer, but can detract from user experience since doing this can increase page loading time. HTTP2 aims to solve this problem in one way, but until more is known about its effects we will want to bundle different parts of our application together and compress it.

Our platform, the browser, must continue to provide backwards compatibility for all existing code and this necessitates slow movement of additions to the base functionality of HTML/CSS/JS. The community has created different tools that transform their preferred syntax/feature set to what the browser supports to avoid binding themselves to the constraints of the web platform. This is especially evident in Angular 2 applications, where TypeScript is used heavily. Although we don't do this in our course, projects may also involve different CSS preprocessors (sass, stylus) or templating engines (jade, Mustache, EJS) that must be integrated.

Webpack solves these problems by providing a common interface to integrate all of these tools and that allows us to streamline our workflow and avoid complexity.

Installation

The easiest way to include webpack and its plugins is through NPM and save it to your `devDependencies`:

```
npm install -D webpack ts-loader html-webpack-plugin tslint-loader
```

Setup and Usage

The most common way to use webpack is through the CLI. By default, running the command executes `webpack.config.js` which is the configuration file for your webpack setup.

Bundle

The core concept of webpack is the *bundle*. A bundle is simply a collection of modules, where we define the boundaries for how they are separated. In this project, we have two bundles:

- `app` for our application-specific client-side logic
- `vendor` for third party libraries

In webpack, bundles are configured through *entry points*. Webpack goes through each entry point one by one. It maps out a dependency graph by going through each module's references. All the dependencies that it encounters are then packaged into that bundle.

Packages installed through NPM are referenced using *CommonJS* module resolution. In a JavaScript file, this would look like:

```
const app = require('./src/index.ts');
```

or TypeScript/ES6 file:

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

We will use those string values as the module names we pass to webpack.

Let's look at the entry points we have defined in our sample app:

```
{
  ...
  entry: {
    app: './src/index.ts',
    vendor: [
      '@angular/core',
      '@angular/compiler',
      '@angular/common',
      '@angular/http',
      '@angular/platform-browser',
      '@angular/platform-browser-dynamic',
      '@angular/router',
      'es6-shim',
      'redux',
      'redux-thunk',
      'redux-logger',
      'reflect-metadata',
      'ng2-redux',
      'zone.js',
    ]
  }
  ...
}
```

The entry point for `app` , `./src/index.ts` , is the base file of our Angular 2 application. If we've defined the dependencies of each module correctly, those references should connect all the parts of our application from here. The entry point for `vendor` is a list of modules that we need for our application code to work correctly. Even if these files are referenced by some module in our app bundle, we want to separate these resources in a bundle just for third party code.

Output Configuration

In most cases we don't just want to configure how webpack generates bundles - we also want to configure how those bundles are output.

- Often, we will want to re-route where files are saved. For example into a `bin` or `dist` folder. This is because we want to optimize our builds for production.
- Webpack transforms the code when bundling our modules and outputting them. We want to have a way of connecting the code that's been generated by webpack and the code that we've written.
- Server routes can be configured in many different ways. We probably want some way of configuring webpack to take our server routing setup into consideration.

All of these configuration options are handled by the config's `output` property. Let's look at how we've set up our config to address these issues:

```
{
  ...
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].[hash].js',
    publicPath: "/",
    sourceMapFilename: '[name].[hash].js.map'
  }
  ...
}
```

> Some options have words wrapped in square brackets. Webpack has the ability to parse parameters for these properties, with each property having a different set of parameters available for substitution. Here, we're using `name` (the name of the bundle) and `hash` (a hash value of the bundle's content).

To save bundled files in a different folder, we use the `path` property. Here, `path` tells webpack that all of the output files must be saved to `path.resolve(__dirname, 'dist')`. In our case, we save each bundle into a separate file. The name of this file is specified by the `filename` property.

Linking these bundled files and the files we've actually coded is done using what's known as source maps. There are different ways to configure source maps. What we want is to save these source maps in a separate file specified by the `sourceMapFilename` property. The way the server accesses the files might not directly follow the filesystem tree. For us, we want to use the files saved under *dist* as the root folder for our server. To let webpack know this, we've set the `publicPath` property to `/`.

Loaders

TypeScript isn't core JavaScript so webpack needs a bit of extra help to parse the `.ts` files. It does this through the use of *loaders*. Loaders are a way of configuring how webpack transforms the outputs of specific files in our bundles. Our `ts-loader` package is handling this transformation for TypeScript files.

Inline

Loaders can be configured – inline – when requiring/importing a module:

```
const app = require('ts!./src/index.ts');
```

The loader is specified by using the `!` character to separate the module reference and the loader that it will be run through. More than one loader can be used and those are separated with `!` in the same way. Loaders are executed right to left.

```
const app = require('ts!tslint!./src/index.ts');
```

> Although the packages are named `ts-loader`, `tslint-loader`, `style-loader`, we don't need to include the `-loader` part in our config.

Be careful when configuring loaders this way – it couples implementation details of different stages of your application together so it might not be the right choice in a lot of cases.

Webpack Config

The preferred method is to configure loaders through the `webpack.config.js` file. For example, the TypeScript loader task will look something like this:

```
{
  test: /\.ts$/,
  loader: 'ts-loader',
  exclude: /node_modules/
}
```

This runs the typescript compiler which respects our configuration settings as specified above. We want to be able to handle other files and not just TypeScript files, so we need to specify a list of loaders. This is done by creating an array of tasks.

Tasks specified in this array are chained. If a file matches multiple conditions, it will be processed using each task in order.

```
{
  ...
  module: {
    rules: [
      { test: /\.ts$/, loader: 'tslint' },
      { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
      { test: /\.html$/, loader: 'raw' },
      { test: /\.css$/, loader: 'style!css?sourceMap' },
      { test: /\.svg/, loader: 'url' },
      { test: /\.eot/, loader: 'url' },
      { test: /\.woff/, loader: 'url' },
      { test: /\.woff2/, loader: 'url' },
      { test: /\.ttf/, loader: 'url' },
    ],
    noParse: [ /zone\.js\/dist\/.+/, /angular2\/bundles\/.+/ ]
  }
  ...
}
```

Each task has a few configuration options:

- *test* - The file path must match this condition to be handled. This is commonly used to test file extensions eg. `/\.ts$/`.
- *loader* - The loaders that will be used to transform the input. This follows the syntax specified above.
- *exclude* - The file path must not match this condition to be handled. This is commonly used to exclude file folders, e.g. `/node_modules/`.
- *include* - The file path must match this condition to be handled. This is commonly used to include file folders. eg. `path.resolve(__dirname, 'app/src')`.

Pre-Loaders

The preLoaders array works just like the loaders array only it is a separate task chain that is executed before the loaders task chain.

Non JavaScript Assets

Webpack also allows us to load non JavaScript assets such as: CSS, SVG, font files, etc. In order to attach these assets to our bundle we must require/import them within our app modules. For example:

```
import './styles/style.css';

// or

const STYLES = require('./styles/style.css');
```

Other Commonly Used Loaders

- *raw-loader* - returns the file content as a string.
- *url-loader* - returns a base64 encoded data URL if the file size is under a certain threshold, otherwise it just returns the file.
- *css-loader* - resolves `@import` and `url` references in CSS files as modules.
- *style-loader* - injects a style tag with the bundled CSS in the `<head>` tag.

Plugins

Plugins allow us to inject custom build steps during the bundling process.

A commonly used plugin is the `html-webpack-plugin`. This allows us to generate HTML files required for production. For example it can be used to inject script tags for the output bundles.

```
new HtmlWebpackPlugin({
  template: './src/index.html',
  inject: 'body',
  minify: false
});
```

Summary

When we put everything together, our complete `webpack.config.js` file looks something like this:

```
'use strict';

const path = require("path");
const webpack = require('webpack');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const basePlugins = [
  new webpack.optimize.CommonsChunkPlugin('vendor', '[name].[hash].bundle.js'),

  new HtmlWebpackPlugin({
    template: './src/index.html',
    inject: 'body',
    minify: false
  })
];

const envPlugins = {
    production: [
        new webpack.optimize.UglifyJsPlugin({
            compress: {
                warnings: false
            }
        })
    ],
    development: []
};

const plugins = basePlugins.concat(envPlugins[process.env.NODE_ENV] || []);

module.exports = {
  entry: {
    app: './src/index.ts',
    vendor: [
      '@angular/core',
      '@angular/compiler',
      '@angular/common',
      '@angular/http',
      '@angular/platform-browser',
      '@angular/platform-browser-dynamic',
      '@angular/router',
      'es6-shim',
      'redux',
      'redux-thunk',
      'redux-logger',
```

```
        'reflect-metadata',
        'ng2-redux',
        'zone.js',
      ]
    },

    output: {
      path: path.resolve(__dirname, 'dist'),
      filename: '[name].[hash].js',
      publicPath: "/",
      sourceMapFilename: '[name].[hash].js.map'
    },

    devtool: 'source-map',

    resolve: {
      extensions: ['.webpack.js', '.web.js', '.ts', '.js']
    },

    plugins: plugins,

    module: {
      rules: [
        { test: /\.ts$/, loader: 'tslint' },
        { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
        { test: /\.html$/, loader: 'raw' },
        { test: /\.css$/, loader: 'style!css?sourceMap' },
        { test: /\.svg/, loader: 'url' },
        { test: /\.eot/, loader: 'url' },
        { test: /\.woff/, loader: 'url' },
        { test: /\.woff2/, loader: 'url' },
        { test: /\.ttf/, loader: 'url' },
      ],
      noParse: [ /zone\.js\/dist\/.+/, /angular2\/bundles\/.+/ ]
    }
}
```

Going Further

Webpack also does things like hot code reloading and code optimization which we haven't covered. For more information you can check out the official documentation. The source is also available on Github.

NPM Scripts Integration

NPM allows us to define custom scripts in the `package.json` file. These can then execute tasks using the NPM CLI.

We rely on these scripts to manage most of our project tasks and webpack fits in as well.

The scripts are defined in the `scripts` property of the `package.json` file. For example:

```
...
  scripts: {
    "clean": "rimraf dist",
    "prebuild": "npm run clean",
    "build": "NODE_ENV=production webpack",
  }
...
```

NPM allows pre and post task binding by prepending the word `pre` or `post` respectively to the task name. Here, our `prebuild` task is executed before our `build` task.

> We can run an NPM script from inside another NPM script.

To invoke the `build` script we run the command `npm run build`:

1. The `prebuild` task executes.
2. The `prebuild` task runs the `clean` task, which executes the `rimraf dist` command.
3. `rimraf` (an NPM package) recursively deletes everything inside a specified folder.
4. The `build` task is executed. This sets the `NODE_ENV` environment variable to `production` and starts the webpack bundling process.
5. Webpack generates bundles based on the `webpack.config.js` available in the project root folder.