LAB # 9

Modules Bundling and Webpack

OBJECTIVE

To get familiar with the need of bundling and the role of webpack tool to assist that.

THEORY

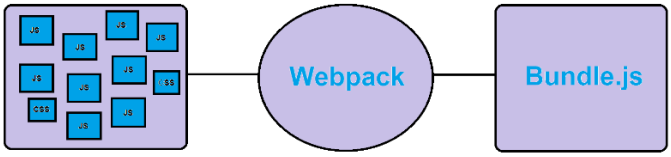
Before we understand what is bundling and how webpack make that process easier, we need to go back in history to understand how we have reached to this point.

Some browser has limit that it does not allow you to open more than 4 connections to the server when a webpage is rendered, and resources are requested from the server to be downloaded. Resources can include JS, CSS, images, and other things. The limit of 4 connections has been put to secure the server from choking down as connections have their own cost to open and close and if they are mostly JS, then they block the webpage rendering until are the JS files are downloaded as it may alter the DOM. So, browser must wait for all of them before it can go and render the DOM.

After the arrival of module patterns in JS (mostly RequireJS made famous by NodeJS), the functionality started to be broken into smaller modules and every module ended up in a separate file. The no. of files grew quicker and all of them had to be sent to the browser for the application to work correctly. As the browsers had limit in how many connections can be opened at a time, this caused the application to wait to render until all the files have been received.

Soon, different bundling solution came up on the horizon to solve that issue. The natural idea was to combine all the files and bundle it in a single or minimum no. of files to improve the client-side performance as a single bundle of files can be sent instead of sending each file in a separate connection. There are other tools in the market that solve this very same issue, but webpack got a lot of traction due to its outstanding features set.

It not only provides bundling JS files but other type of assets as well. It starts off from an entry point and resolves every dependency being used by every file. It generates the dependency graph and then resolves and transforms all the dependencies according to the configuration defined and merge and bundle them into single file called its output. This is not the only thing webpack is capable of. Since, it has grown a lot from its inception, new features have been added into it like multiple entry points, tree-shaking, splitting, and cache busting to name a few.



Pic 1 Simplified Webpack Bundling Process

To get started, there are few concepts you need to be aware of as follows.

- Configuration File
- Entry Point Path
- Output Path and File
- Loaders
- Plugins
- Mode

Configuration File

Although webpack can be used with no configuration at all as it comes pre-configured with default values. But the configuration can be defined if any value needs to be overridden. Here is an example of configuration:

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './path/to/my/module/file.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'university-core.bundle.js',
  }
};
```

In the above code, an object is defined and assigned to module.exports property. This is the module syntax to export any module. In this case, we are exporting a webpack configuration object. There are two properties being defined, one is *entry* and the other one is *output*. The default value for *entry* property is *'./src/index.js'* and *'./dist/main.js'* for *output path* property (nested property) if we don't define the configuration as we have done above.

By default, webpack understands JS and JSON and for the other types of dependencies, it uses a concept called *loaders.* A *loader* can be used to transform CSS and output it as a module in the

final bundle. For the loader configuration to detect dependencies other than JS, it requires a definition of what type of loader (using *use* property) to use and which files (*test* property) need to be transformed.

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './path/to/my/module/file.js',
  output: {
    filename: 'my-first-webpack.bundle.js',
  },
  module: {
    rules: [
          { test: /\.css$/, use: 'css-loader' },
    ]}
};
```

Entry Point Path

It is the entry point of the webpack modules bundling process. Webpack looks and use this property to know which first module to start bundling from. It then works its way to resolve other dependencies found in the entry point module. It resolves all the dependencies recursively and generates a dependency graph.

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './path/to/my/module/file.js',
};
```

Output Path and File

It is the exit point of the webpack modules bundling process. Webpack looks and use this property to know where to emit the bundle. The *output.path* property accepts the path to bundle output directory, and *filename* accepts the name of the generated bundle.

webpack.config.js

```
const path = require('path');

module.exports = {
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-first-webpack.bundle.js',
  }
```

```
};
```

Loaders

Loader is used when webpack needs to process module types other than JS – which is the module type picked up by bundling process automatically without defining any loader. A loader can be defined using top level *rules* property which accepts an array of loaders. Any no. of loaders can be configured, and each configuration object has at-least two properties, *test,* and *use*. Loaders are evaluated from bottom to top and from right to left. The *test* property is defined using regular expression.

webpack.config.js

```
const path = require('path');

module.exports = {
  module: {
    rules: [
          { test: /\.css$/, use: 'css-loader' },
          { test: /\.ts$/, use: 'ts-loader' },
    ]}
};
```

In the above example, all files ending up in *css* extension will be processed by *css-loader*. The *ts-loader* will be evaluated first and then *css-loader*.

Plugins

Anything which a loader is not capable of doing is done by a *plugin*. A plugin works outside of the bundling process like receiving the progress of bundling, generating an html file and injecting and referencing the bundle in the generated html file. Plugins are defined using top level *plugins* property as an array.

webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm
const webpack = require('webpack'); //to access built-in plugins
const path = require('path');

module.exports = {
  module: {
    plugins: [
          new webpack.ProgressPlugin(),
          new HtmlWebpackPlugin({ template: './src/index.html' }),
    ]
  }
};
```

In the above example, there are two plugins defined. *ProgressPlugin* shows the progress of bundling and emits certain events, and *HtmlWebpackPlugin* generates an *html* file and reference the generated bundle and link with it. The use of *new* keyword indicates that it can accept certain parameters to tweak its behavior. Plugins are expected to define an *apply* method which webpack calls during its invocation of plugin application.

Mode

mode property tells webpack which optimization technique to use. There are few options that can be defined on *mode* property, and depending on that, webpack will perform certain optimization, transformation, and apply validation during and after bundling process. The other values of mode are *development* and *none*.

webpack.config.js

```
const path = require('path');

module.exports = {
  mode: ‘production’,
};
```

In the above example, the *mode* is set to *production* which will enable webpack to optimize the bundle in terms of size and other things. If the mode were set to *development*, a *source map* would also be produced among other things to make it easier to debug the code.

There are many features offered by webpack which are kind of advance at this stage but being listed down here to give you an idea of how capable and rich webpack is in terms of features offerings.

- Module Federation
- Bundling Per Page or Splitting
- Hot Module Replacement
- Modules Caching
- Module Cache Busting Hash
- Tree Shaking
- Lazy Loading Modules
- Etc

Webpack Application (NodeJS and VS Code is needed to develop this app**)**

Here is a step-by-step instruction given on how to bundle a sample application files and use it to be consumed by the *html page*. The idea is to reduce the no. of connections opening-up to the individual files (modules actually) and use a single bundle only or minimum no. of files instead. In production apps, this helps reduce the no. of connections to the server as there is only one bundle to be loaded by the browser referenced by html page and thereby improving the overall application performance.

Activity 1: Creating package.json file and installing webpack module

1. Go to command prompt and create a project directory called ***students***.
2. Switch to that directory by typing following command ***cd students***.
3. Type ***npm init*** and press *Enter* for every option. This will create a *package.json* file to maintain npm packages with some other project information.
4. Type following command ***npm install –save-dev webpack webpack-cli.*** This will install the latest webpack module inside the *node_modules* directory. It will also update the package.json file *devDependencies* section to record the newly installed webpack module. The reason that it added the webpack module under *devDependencies* and not under *dependencies* section is because of the –save-dev switch provided in the preceding command. We need this only during development time and not during production when app is running live.
5. Type ***code .*** This will open *Visual Studio Code* editor with the current directory loaded (*students* directory).

Activity 2: Adding the webpack execution command under package.json scripts section

6. In Visual Studio Code, open the package.json file by clicking it.
7. Add following property under package.json scripts section. Here the command is called *build*. This command will be used to kick off the bundling process using webpack.

package.json

```
…
scripts: {
    “build”: “webpack”
},
…
```

8. We need to create a webpack configuration file for bundling instructions.

Activity 3: Adding webpack configuration file

9. Create a new file inside *students* folder called *webpack.config.js* from Visual Studio Code.
10. Type following code inside the webpack.config.js file.

webpack.config.js

```
const path = require('path');

module.exports = {
   entry: './src/index.js',
   output: {
      path: path.resolve(__dirname, 'bundle'),
```

```
        filename: 'students.js'
    }
}
```

11. Basic webpack configuration has been set. Now we will create our actual app modules.

Activity 4: Creating app modules

12. Create a new folder inside *students* folder called *src* from Visual Studio Code.
13. Inside *src* folder, create a new file called *student-profile.js*.

student-profile.js

```
const name = 'xyz';
const email = 'xyz@ssuet.com';
const dob = new Date(2000, 1, 1);
const profile = {
    name,
    email,
    dob
};

export { profile };
```

14. Inside *src* folder, create a new file called *student-courses.js*.

student-courses.js

```
const courses = ['js', 'html', 'css'];
const logCourses = () => {
    for(let course of courses) {
        console.log(course);
    }
};

export { logCourses };
```

15. Inside *src* folder, create a new file called *index.js*.

index.js

```
import { profile } from './student-profile.js';
import { logCourses } from './student-courses.js';
```

```
console.log('STUDENT PROFILE ----------------------');
console.log(profile.name);
console.log(profile.email);
console.log(profile.dob);

console.log('STUDENT COURSES ----------------------');
logCourses();
```

Activity 5: Creating index.html and linking the bundle file.

16. Under *students* folder, create a new file called *index.html*.

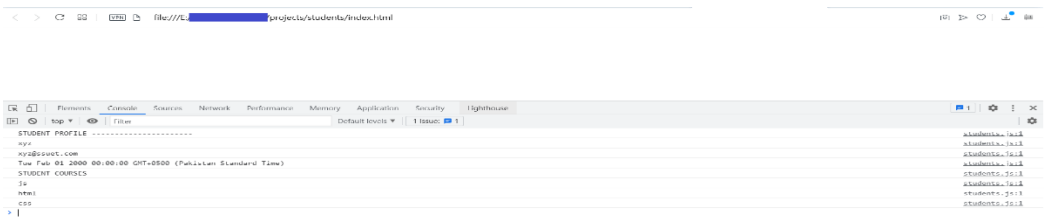
index.html

```
<html>
   <head>
      <script src="./bundle/students.js"></script>
   </head>
   <body></body>
</html>
```

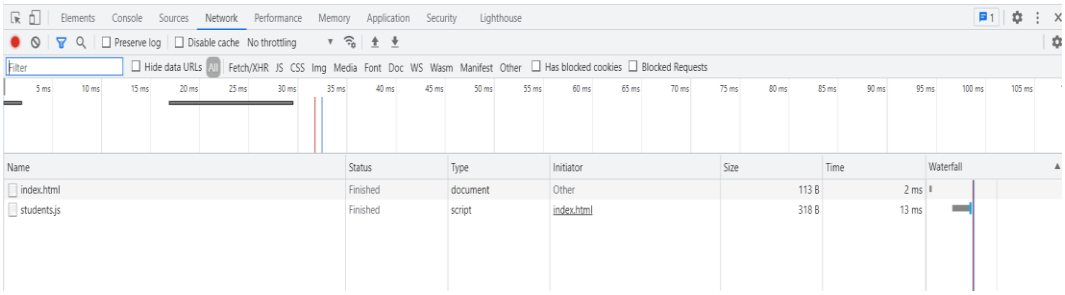
Activity 6: Kicking off webpack bundling process and running the index.html file.

17. From Visual Studio Code main menu, click on *Terminal* menu and then *New Terminal* sub-menu.
18. Type following command ***npm run build***. This command will start the bundling process and then emit the bundle output under *./bundle* inside *students* folder using webpack configuration *output.path* property defined above. The bundled file will be named as *students.js* as was defined under *output.filename* property.
19. Go to inside *students* folder and run *index.html* in any browser.
20. Open the browser *Developer Console Tab* where the code has been executed to log the information.

console output



students.s bundle



21. Now you can see that only single *students.js* bundle file is being referenced from *index.html* file instead of referencing *index.js, student-profile.js,* and *student-courses.js* files. When rendering the *index.html* file, browser will only open one connection to request the *students.js* bundle and not three connections to request each individual file separately resulting in much more improved app performance.

Lab Task

JS

1. Create an app called *university management*.
2. Define three app feature modules called *students.js, faculties.js* and *events.js*, and one main app module called *app.js*.
3. Add functionality to each app feature modules and call them from the main app module.
4. Define configuration for webpack bundling process and change the entry point to *app.js*

CSS

5. Add an app styling SASS file called *app-style.scss* and add relevant styles for students, faculties, and events section to be rendered on html page.
6. Define appropriate loader in webpack config to transform the app-style.scss to normal CSS file and output it under *assets* folder, and name it *style.css*. You will need to install the *sass-loader* and *css-loader* using *npm install* command. You will also need to require in *mini-css-extract-plugin* plugin and define it under *plugins* array property of webpack config file.

HTML

7. Create an *index.html* file and reference the JS output bundle and CSS files.
8. Using the information returned by referenced bundle inside html page, layout the information utilizing html tags.

RUN and VERIFY

9. Run the webpack config to output the bundle.
10. Open the html page to view its result and verify that there is only one single bundle and that CSS styling has been applied.

Home Task

SPLIT BUNDLES

1. Using the lab assignment, split the *node_modules* packages (modules) into separate *vendor.js* and app.js files.
2. Reference the split modules from the html page.
3. Verify the results.