Introduction

* **Static module bundler** for modern **javascript applications**. When processes application, it recursively builds a dependency graph that includes every module. And the bundles into **one or more**.
* **Order** // If we use a js functions in one file in another we have to put the scripts in the order that they can access it. what if we have a lot of js files?

<script src="./src/hello-world.js"></script>

<script src="./src/index.js"></script>

* <https://github.com/vp-online-courses/webpack-tutorial>

Type of files it can handle

* JavaScript
* Typescript
* Coffeescript
* CSS
* Sass
* Less
* Images

Tools:

* [**https://cmder.app/**](https://cmder.app/)
* Grunt // concatenate all are js files into one
* Gulp // they still **could not** figure out **dependencies** between files.
* Require.js // it can manage dependencies but not as powerful

Usefull command line instructions

* rm -rf dist // in git bash

install

* Package.json // keeps track of all dependencies

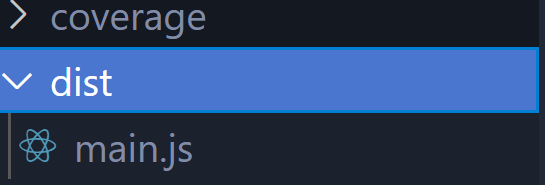
npm init -y

* npm install **webpack** **webpack-cli --save-dev**
  + webpack // core functionality
  + webpack-cli // command line tool, to run webpack from terminal
  + --save-dev // adds webpack to package.json as dev dependency. We need the package during build process. We don't usually need webpack plugins in production environment.

npm install webpack webpack-cli --save-dev

* Run webpack

npx webpack



* Run webpack more conveniently by providing a script in package.json. everytime we run this script, it will run webpack for us.

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"build": "webpack"

},

Configuration file for webpack

* Webpack 5 and 4 uses default config unless provided one
* webpack.config.js // default name. You can choose another name too.
* **module.exports = {}** // basically a js module that **exports the configuration object**

Options:

* mode
* entry: '' // Entry point, default src/index.js. starts from this file for the build process
* output: { filename: 'bundle.js', path: './dist' } // creates them if they don't exist. **Path should be absolute!**
  + Const path = require('path') // old way of importing modules sinc in config file we can not use ecmascript modules! A package that generates the absolute path. It converts a sequence of passed segments into one absolute path.
  + **publicPath: 'dist/'** // where all generated files are located. Or the root domain name of our website
* mode: 'none' // mandatory

// basically a js module

const path = require("path")

module.exports = {

entry: "./src/index.js",

output: {

filename: "bundle.js",

path: path.resolve(\_\_dirname, "./dist")

}, // creates them if they don't exist

mode: "none", // mandatory

}

* module: {rules: [] } // define the loaders
  + **rules** // have an array of specific rules. Each rule is an objects with two properties at least:
    - **test**: regular expression to match certain files formats.
    - **use**: [] // which loader should be used.
      * **use**: **{** loader: "babel-loader", options: { presets: [], plugins: [] }**}** // have options that loader supports them
    - **exclude**: /node\_modules/

module: {

rules:[

{

test: /\.(png|jpg)$/,

use: [

'file-loader'

]

}

]

}

* + We can have more than one rule
* plugins:[] // create a new instance

Plugins: {

new TerserPlugin()

}

packages

loaders

* file-loader
* xml-loader
* style-loader
* css-loader
* sass-loader
* node-sass
* handlebars-loader

plugins

* mini-css-extract-plugin
* Clean-webpack-plugin

Packages

* Handlebars
* webpack-dev-server

webpack loaders

* usually bundle javascript files that require each other
* load css, sass, less, xml, images files directly to javascript files
* javascript libraries that help you load all that stuff
* whenever we want to import a specific file, webpack checks if it has a rule for it, if not, it gives an error.
* By default, it knows how to import:
  + Javascript
  + Json
* Be careful with the name of loaders, there should be no space at the end

Images

* npm install file-loader --save-dev
* file-loader // add a special rule

module: {

rules:[

{

test: /\.(png|jpg)$/,

use: [

'file-loader'

]

}

]

}

* use image loader

// cactus contains the public url to the file

import Cactus from "./Cactus.jpg"

function addImage() {

const img = document.createElement("img")

img.alt = "Cactus"

img.width = 300

img.src = Cactus

const body = document.body

body.appendChild(img)

}

export default addImage

* the **name** of the **resulting file**. By default is the md5 hash of the content of the file and the original extension.
* If image is not loaded with the correct address you may want to add the publicPath option to output option

xml

* /\.(xml)$/
* xml-loader

Css files

* **npm install css-loader style-loader --save-dev**
* In component based javascript libraries like React js, we want to keep everything in one place so we can track issues, and for future upgrades.
* Two loaders: combine multiple loaders in one single rule. It tells the webpack it needs to use both.
  + **style-loader** // it injects it to the page by style text. This methods combines the result of js and css into one bundle.js file and then dynamically is added to dom by javascript during run time. It can become very big, and needs more time to be loaded. We can separate them too.
  + **css-loader** // it only reads the content and returns it

{

test: /\.css$/,

use: ["style-loader", 'css-loader'],

},

sass

* npm install **sass-loader** **node-sass** --save-dev
  + **node-sass** // binding to LibSass. Libsass is the c version of the popular stylesheet preprocessor called Sass. It allows to convert scss files to css files at an incredible speed.
* Add the sass-loader to the chain. Webpack processes the loaders from right to left. First coverts the sass to css

{

test: /\.scss$/,

use: ["style-loader", 'css-loader', 'sass-loader'],

},

Two files (css and bundle for js) method

* parallel loading
* smaller bundle file
* mini-css-extract-plugin

npm install mini-css-extract-plugin --save-dev

* new MiniCssExtractPlugin() // add to plugin section of webpack.config.js. extracts our css to a separate file

new MiniCssExtractPlugin({

filename: "styles.css",

}),

* change the css and sass loaders

{

test: /\.css$/,

use: [**MiniCssExtractPlugin.loader**, "css-loader"],

},

{

test: /\.scss$/,

use: [**MiniCssExtractPlugin.loader**, "css-loader", "sass-loader"],

},

* import it

const MiniCssExtractPlugin = require("mini-css-extract-plugin")

* add it to index.html

<link rel="stylesheet" href="./dist/styles.css" />

Bibel loader

* Javascript language is based on ecmascript specification and this specification is evolving all the time. When a new version of ecmascript specification comes out browser go ahead and implement them. But it may take some time. But we want to use the new features!
* Covert modern javascript code into older javascript code that is already supported by the browsers.
* Bible is a very famous javascript compiler
* Class properties were not supported by major browsers at some point.
* Apply the rule to all javascript files except the node-modules folder
* **@babel/env** // env is a special preset that compiles ecmascript 6-7-8-9-10 down to ecmascript 5. Env preset support the latest javascript standard defined in the latest ecmascript specification.
* **transform-class-properties** // babel plugins to support specific features that are not supported in the latest ecmascript specification like class properties.
* npm install
  + @babel/core // scoped modules (namespaced).
  + babel-loader
  + @babel/preset-env
  + babel-plugin-transform-class-properties

npm install @babel/core babel-loader @babel/preset-env babel-plugin-transform-class-properties --save-dev

* if the webpack faces a js code that is not supported by current ecmascript specification then webpack will convert such code to the older javascript code which will be understood by all browsers at this time.

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader",

options: {

presets: ["@babel/env"],

plugins: ["transform-class-properties"],

},

},

Webpack Plugins

* [list](https://webpack.js.org/plugins/) of official webpack plugins.
* a plugin is a javascript library that adds functionality to webpack itself. Plugins can do everything that loaders cannot do. Not just import new type of files.
  + define global constants across the whole application
  + minify results in bundle
  + generate other files besides bundle.js

minify bundle

* **terser-webpack-plugin** // Recommended way. It automatically **minifies after npm run build**

npm install terser-webpack-plugin --save-dev

* TerserPlugin

plugins:[

new TerserPlugin()

]

* Import it at the top

const TerserPlugin = require('terser-webpack-plugin')

* uglify-js is another plugin to minify files

Browser Caching

* What if we fix a bug and js has changes?
* Mechanisms?
  + Change the name // browsers remember files by name. and they can download the new version. We don't have to change names manually.
* Webpack.config.js
  + **[contenthash]** // Md5 hash add to the file name. It remains the same if the content don't change

  output: { filename: "bundle**.[contenthash].**js", path: path.resolve(\_\_dirname, "./dist"), publicPath: "dist/" }, // creates them if they don't exist

* For css

plugins: [

new TerserPlugin(),

new MiniCssExtractPlugin({

filename: "styles.**[contenthash].**css",

}),

],

* **clean-webpack-plugin //** Remove all the prev builds before generating new ones (the dist folder can be clut). In each build simply remove all the files from **path**.
  + **npm install clean-webpack-plugin --save-dev**

Const { CleanWebpackPlugin } = require('clean-webpack-plugin')

…

plugins: [

new TerserPlugin(),

new MiniCssExtractPlugin({

filename: "styles..[contenthash].css",

}),

new CleanWebpackPlugin(),

],

* clean other folders // for example you may have an additional plugin that creates some css files in another folder
  + cleanOnceBeforeBuildPatterns:[] // initiate the plugin
    - **'\*\*/\*'** // the default. All the files and subfolders in dist (path)
    - **path.join(process.cwd(), 'build/\*\*/\*')** // other folders like build here.

new CleanWebpackPlugin({

cleanOnceBeforeBuildPatterns:[

'\*\*/\*',

path.join(process.cwd(), 'build/\*\*/\*')

]

}),

Automatic insertion of asset file names in index.html

* Auto generated files names for versioning and minifying. Change manually every time in index.html? No
* html-webpack-plugin // [github link](https://github.com/jantimon/html-webpack-plugin)

npm install html-webpack-plugin --save-dev

const HtmlWebpackPlugin = require('html-webpack-plugin')

…

plugins: [

**new HtmlWebpackPlugin()**

],

}

* the new html file in generated inside dist folder. The assets address in index.thml still have the dist/ prefix. We need to change them. By:

publicPath: ''

* We have two html files. remove the one we made ourselves.

Customize the generated index.html

* Custom options // **title**, the **name of file**, additional **meta tags**

new HtmlWebpackPlugin({

title: "Hello world",

filename: "subfolder/custom\_filename.html",

meta: {

description: "Some description",

},

}),

Complete customized template

* **Template engines**
  + pug
  + ejs
  + underscore
  + handlebars
  + html-loader
* **handlebars** // a template engine for javascript that allows you to separate the business logic from presentation
* **index.hbs** // an extension for handlebar template. Put anything you want inside the handlebar files. Webpack will use it as a template file to generate the html file during the build process
* **{{htmlWebpackPlugin.options.title}} // handlebars variables**

<html>

<head>

<meta charset="utf-8" />

<title>{{htmlWebpackPlugin.options.title}}</title>

<meta name="description" content="{{htmlWebpackPlugin.options.description}}" />

<meta name="viewport" content="width=device-width, initial-scale=1" />

</head>

<body>

</body>

</html>

* Add the template as a variable to HtmlWebpackPlugin. Descriptions variable should be on the same level as the template. Handlebars will the take the variables from here.

new HtmlWebpackPlugin({

title: "Hello world",

template: "src/index.hbs",

description: "Some description",

}),

* Introduce .hbs to the webpack

{

test: /\.hbs$/,

use: [

'handlebars-loader'

]

}

* Install handlebars-loader and handlebars

npm install handlebars --save-dev

Production and Development environment

* In dev [mode](https://webpack.js.org/configuration/mode/#mode-production) we may want additional info like source-map
* But in production we don't need that additional info, and we want our bundle to be as small as possible (load time wise).
* **Mode:**
  + None
  + Development // when we want to develop
  + Production // when you want to deploy the application you need to set it to this.
* Production and development modes handle errors differently. For example if we call a function that does not exist, in dev mode, we see exactly what file (readable version) throws that error, but in production mode, we only see somewhere in bundle.js and the minified version. Because dev mode uses source-map by default.

Production

* Enables a lot of plugins like
  + Terser // included by default you can remove it from prod.config
* process.env.NODE\_ENV // check this variable to see what mode we are in.

Effectively manage different configs (dev and prod)

* two separate configuration files
  + prod builds
  + dev builds
* two different scripts in package.json
  + --config webpack.production.config.js

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"build": "webpack --config webpack.production.config.js",

"dev": "webpack --config webpack.development.config.js"

},

* change the name of the file with mode dev webpack.production.config.js

webpack.production.config.js

* no need for terser, since by default is in production mode.

webpack.dev.config.js

* No browser caching // remove the **contenthash**
* Don't need tereser, because we don't need to minify in dev
* Optimize the performance of build process:
  + We don't need to extract our css files into a separate file in dev
  + Use **style-loader** instead
  + Make the code faster to be build.

Live server (dev)

* **webpack-dev-server** // a package
* in **webpack.development.config.js:**

devServer: {

static: {

// point to dist folder. no need for index.html. by default looks for it in here:

directory: path.resolve(\_\_dirname, "dist"),

},

// instead of writeToDisk: true,

hot: true,

open: true,

port: 9000,

devMiddleware: {

writeToDisk: true,

index: "index.html",

},

},

* change the package.json
* --hot

"dev": "webpack **server** --config webpack.development.config.js **--hot**"

Multi-page application

* For single-page app we can use one bundle.js file
* In multi-page, server sends different pages to the browser based on url
* Example: Another cactus component and cactus.js file that has the heading and cactus component. Rename index to hello-world.js.
* In webpack-production.config.js
  + **entry**: { x: 'file address', y: 'file address'}. // two entries
  + **[name]** // two different names for their bundles . webpack takes the name from property name of entry and substitute here.
    - We can do this for mini-css too

entry: {

**"hello-world": "./src/hello-world.js",**

**"cactus": "./src/cactus.js",**

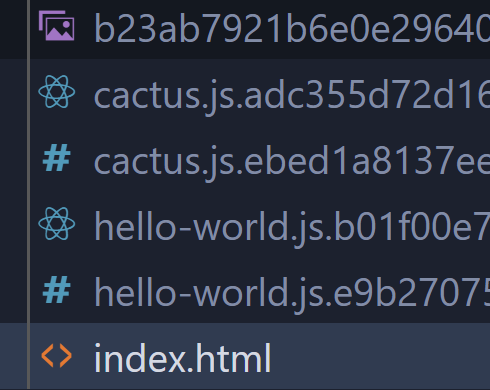
},

output: { filename: "**[name]**.[contenthash].js", path: path.resolve(\_\_dirname, "./dist"), publicPath: "" }, // creates them if they don't exist

new MiniCssExtractPlugin({

filename: "**[name]**.[contenthash].css",

}),



**Different html pages with different bundles**

* By default webpack will include both in one index.html. **Generate multiple html files and include different bundles inside different html files**
* In HtmlWebpackPlugin: two instances instead of one:
  + **filename: "html file name"**
  + **chunks: ["property name of entry points"]** // an array. special option where we can specify which bundles we want to include in what html file.

new HtmlWebpackPlugin({

// filename needed when we have multiple instance of htmlwebpackplugin to have multiple output html files.

filename: "hello-world.html",

chunks: ['hello-world'],

title: "Hello world",

template: "src/page-template.hbs",

description: "Hello World",

}),

new HtmlWebpackPlugin({

filename: "cactus.html",

chunks: ['cactus'],

title: "Cactus",

template: "src/page-template.hbs",

description: "Cactus",

}),

Include common libraries for webpages

* Reasons to use
  + For caching purposes,
  + Extract some external dependencies like react into a separate bundle
  + Multiple html files and reducing their sizes.
* Example: loadash library

import HelloWorldButton from "./components/hello-world-button.js/hello-world-button.js"

// import addImage from "./add-image.js"

import Heading from "./components/heading/heading.js"

**import \_ from "lodash"**

const heading = new Heading()

heading.render(**\_.upperFirst("Hello World")**)

const helloWorldButton = new HelloWorldButton()

helloWorldButton.render()

* By including a library like lodash in the header, webpack will include it in all of the files that have the headers ( two html files that we have here for example). So the size of the files will increase.

optimization: {

splitChunks: {

chunks: "all"

}

},

* By this, we don't have the libraries everywhere that have been imported, but webpack cashes it in a new file for it, and uses (we have to define that) it where it is needed. And when we make a change to any other file that uses it, the users don't need to download the library again, as they already have it.
* how to use the cashed file:
* in HtmlWebpackPlugin // in **newer versions webpack does this by default.** 
  + chuncks // add it to the array

new HtmlWebpackPlugin({

// filename needed when we have multiple instance of htmlwebpackplugin to have multiple output html files.

filename: "hello-world.html",

chunks: ["hello-world", **"vendors~hello-world-kiwi"**],

title: "Hello World",

template: "src/page-template.hbs",

description: "Hello World",

}),

* minSize : 1000 // by default, react extracts common dependencies only when they exceed 30 kBytes before minification. For example it does not work if you include react in two places, which is less than 30 kilobytes.

optimization: {

splitChunks: {

chunks: "all",

minSize: 9000,

automaticNameDelimiter: "\_",

},

},

Dev configs

* Two entries and [name] is the same. And two HtmlWebpackPlugins for each file like before.
* We don't need to split common dependencies into a separate bundle because this is the dev build and we all care about is the speed of build.
  + <http://localhost:9000/hello-world.html>
  + http://localhost:9000/cactus.html

Integrating Express

Single Page

* Express.js

const express = require("express")

const app = express()

const path = require("path")

const fs = require("fs")

// res=and instance of response object

app.get("/", function (req, res) {

const pathToHtmlFile = path.resolve(\_\_dirname, "../dist/index.html")

// read the content of index.html. read sync

const contentFromHtmlFile = fs.readFileSync(pathToHtmlFile, "utf-8")

res.send(contentFromHtmlFile)

})

// start server listen to a port

app.listen(3000, function () {

console.log("Application is running on http://localhost:3000/")

})

* Send the content of static files files back to the browser. // The empty page error occurs because express does not know how handle static files like css and js files. contentFromHtmlFile has the html content, but css and js files are not there! They are not sent by the request.
* All the static files are in the dist folder. We create a new route for static files.

// /static/ is the route. express.static method to serve static files.

app.use("/static", express.static(path.resolve(\_\_dirname, "../dist")))

* Add the /static/ to all the places the static files are addressed (index.html). We do this by webpack config

output: { filename: "bundle.[contenthash].js", path: path.resolve(\_\_dirname, "./dist"), publicPath: "/static/" }, // creates them if they don't exist

Multipage

* everything the same as single but

// res=and instance of response object

app.get("/hello-world/", function (req, res) {

const pathToHtmlFile = path.resolve(\_\_dirname, "../dist/hello-world.html")

// read the content of index.html. read sync

const contentFromHtmlFile = fs.readFileSync(pathToHtmlFile, "utf-8")

res.send(contentFromHtmlFile)

})

app.get("/cactus/", function (req, res) {

const pathToHtmlFile = path.resolve(\_\_dirname, "../dist/cactus.html")

// read the content of index.html. read sync

const contentFromHtmlFile = fs.readFileSync(pathToHtmlFile, "utf-8")

res.send(contentFromHtmlFile)

})

Module Federation - run two separate applications

* Two folders with all the files
* Two ports 9001 9002 for each app

devServer: {

static: {

// point to dist folder. no need for index.html. by default looks for it in here:

directory: path.resolve(\_\_dirname, "dist"),

},

// instead of writeToDisk: true,

hot: true,

open: true,

port: 9001,

},

* in sever.js

app.listen(**9001**, function () {

console.log("Application is running on http://localhost:3000/")

})

Module Federation

* New in webpack 5: upgrade

npm install --save-dev webpack@latest webpack-cli@latest

* Allows one application to dynamically to use modules from another application at the run time.
* Our two applications have different dependencies and can be deployed separately

Module federation plugin

* Its already included.

const { ModuleFederationPlugin } = require("webpack").container

* Use hello-world button in cactus. During the build process, webpack will generate a file that contains everything this application export to the outer wold so theat other applications can use that.
* Filename: 'remoteEntry.js' // is to give this file a name. this name is the convention most people use.
* Exposes // modules that can be exposed

new ModuleFederationPlugin({

name: "HelloWorldApp",

filename: "remoteEntry.js",

exposes: {

"./HelloWorldButton": "./src/components/hello-world-button/hello-world-button.js",

},

}),

* All the othe applications will be referenced in this applicatin using public url, and this public url is baked into a remote entry file and we need to make sure it is correct. During the build process webpack doesn’t' know where we are going to deploy our application. We can build our application on a local machine and then take the generated files and copy them to some cdn. There is no way that webpack magically know the url of that cdn. Somehow we need to tell webpack what url.
  + publicPath

output: { filename: "[name].bundle.js", path: path.resolve(\_\_dirname, "./dist"), publicPath: "http://localhost:9001" }, // creates them if they don't exist

* put the ModuleFederationPlugin in production config too and change the publicPath too.
* In server.js. we no longer want to get the static files from the /static since we changed pbulicPath.

app.use("/", express.static(path.resolve(\_\_dirname, "../dist")))

* To consume ModuleFederationPlugin modules in Cactus application: Be careful of **names**, or you get **can't resolve module error**
  + No need to change the server static files for cactus or the public url, just use moduleFederated and import the module.
  + **remotes**: { } // remotes objects shared by other applications.

new ModuleFederationPlugin({

name: "CactusApp",

remotes: {

HelloWorldApp: "HelloWorldApp@http://localhost:9001/remoteEntry.js",

},

}),

* Consume the button in the code base. In cactus.js file:

// dynamic import, because remote buttons are loaded async

// HelloWorldApp is the name of application we specified in webpack.config ModuleFederatedPlugin

// HelloWorldButton is the name of the component hello-world exposes to the outer world

// since we use the default export to export the helloworldbutton component we need to get the default export

import("HelloWorldApp/HelloWorldButton")

.then((HelloWorldButtonModule) => {

const HelloWorldButton = HelloWorldButtonModule.default

const helloWorldButton = new HelloWorldButton()

// we are using a component that is defined in another application and it's not listed in package.json

// it is loaded dynamically at run time.

helloWorldButton.render()

})

Benefits

* Need to change the color of button based on business requirements, and when we change it hello-world application that has the button, we don't need to start the cactus app again, only refresh. Because that module is loaded during run time.

Micro Front-ends

* We have a **top-navigation app** called dashboard used only for routing, the content is shown below it comes from another url, this kind of architecture is called micro frontend architecture. It's useful when **each and every page is developed by a different team.**
* The dashboard app which serves as a container for multiple independent applications called micro-frontends. The dashboard app includes the routing logic that allows to switch between these independent applications. Other applications like cactus app, are compoletely decoupled from the dashboard application and deployed on a different url. Dashboard app loads some of the components exposed byt the cactus and hello-world apps and renders them on the page.
* **Export everything** in hello-world app **as a componen**t, so we **can consume** it in the dashboard app. In src/components create a component named hello-world-page. Copy everything from hello-world.js (that contains all the imports from heading and button, the final page we show) to it. put all the codes in the render method of a class and export it:

import HelloWorldButton from "../hello-world-button/hello-world-button.js"

// import addImage from "./add-image.js"

import Heading from "../heading/heading.js"

class HelloWorldPage {

render() {

const heading = new Heading()

heading.render("Hello World")

const helloWorldButton = new HelloWorldButton()

helloWorldButton.render()

// addImage()

}

}

export default HelloWorldPage

* Expose it via ModuleFederatedPlugin

"./HelloWorldPage": "./src/components/hello-world-page/hello-world-page.js",

* Need to do the same for cactus-page and update its publicPath and remove the remotes from ModuleFederatedPlugin since we don't need it here. Also change the server.js so the url for the static files is the root (/).
* Create a folder named **dashboard** and run **npm init -y. in package.json:**

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"build": "webpack --config webpack.production.config.js",

"dev": "webpack serve --config webpack.dev.config.js --hot",

"start": "node src/server.js"

},

* Create the webpack.config and use ModuleFederatedModule to consume the assets from remote apps. Import the modules in dashboard, and render them. Need a express server to run the dashboard app.

…

module.exports = {

entry: "./src/dashboard.js",

output: {

filename: "[name].bundle.js",

path: path.resolve(\_\_dirname, "./dist"),

publicPath: "http://localhost:9000/",

},

mode: "development",

devServer: {

static: {

directory: path.resolve(\_\_dirname, "dist"),

},

hot: true,

open: true,

port: 9000,

devMiddleware: {

writeToDisk: true,

index: "dashboard.html",

},

**// always return dashboard.html no matter what you really put in the browser navigation bar.**

**historyApiFallback: {**

**index: "dashboard.html",**

**},**

},

….

plugins: [

new CleanWebpackPlugin(),

new HtmlWebpackPlugin({

**filename: "dashboard.html",**

**title: "Dashboard",**

}),

new ModuleFederationPlugin({

**name: "App",**

**remotes: {**

**HelloWorldApp: "HelloWorldApp@http://localhost:9001/remoteEntry.js",**

**CactusApp: "CactusApp@http://localhost:9002/remoteEntry.js",**

},}),],}

* In dashboard.js

// this does not include domain and port

const url = window.location.pathname

if (url === "/hello-world-page") {

import("HelloWorldApp/HelloWorldPage").then((HelloWorldPageModule) => {

const HelloWorldPage = HelloWorldPageModule.default

const helloWorldPage = new HelloWorldPage()

helloWorldPage.render()

})

} else if (url === "/cactus-page") {

import("CactusApp/CactusPage").then((CactusPageModule) => {

const CactusPage = CactusPageModule.default

const cactusPage = new CactusPage()

cactusPage.render()

})

}

* In server of dashboard app:

const express = require("express")

const path = require("path")

const fs = require("fs")

const app = express()

// We need to define how to serve static files

app.use("/", express.static(path.resolve(\_\_dirname, "../dist")))

// serve the same html file for any url

app.get("\*", function (req, res) {

const pathToHtmlFile = path.resolve(\_\_dirname, "../dist/dashboard.html")

const contentFromHtmlFile = fs.readFileSync(pathToHtmlFile, "utf-8")

res.send(contentFromHtmlFile)})

app.listen(9000, function () {

console.log("Application is running on http://localhost:9000")

})

* Install dev dependencies. Only used in build process

npm install webpack webpack-cli webpack-dev-server mini-css-extract-plugin html-webpack-plugin clean-webpack-plugin babel-loader @babel/core @babel/preset-env --save-dev

* Install production dependencies. Express is a prod dependency because we use it to serve all the generated files.

npm install express --save

* Build and run two apps and then build and run dashboard

Navigation bar component

* Invoke nav bar component and pass the one argument that is all the pages we have
* In dashboard.js

import NavigationBar from "./components/navigation-bar/navigation-bar.js"

const navigationItems = [

{

url: "/hello-world-page",

title: "Hello World Page",

},

{

url: "/cactus-page",

title: "Cactus Page",

},

]

const navigationBar = new NavigationBar()

navigationBar.render(navigationItems)

* Install

npm install style-loader css-loader sass-loader node-sass -save-dev

* Then in navigation-bar.js

import "./navigation-bar.scss"

class NavigationBar {

render(navigationItems) {

const liItems = navigationItems.map((navigationItem) => {

return `

<li>

<a href="${navigationItem.url}">${navigationItem.title} </a>

</li>

`

})

const ul = document.createElement("ul")

ul.innerHTML = liItems.join("")

ul.classList.add("navigation-bar")

document.querySelector("body").appendChild(ul)

}

}

export default NavigationBar

Integrate Jquery into Webpack

* npm install jquery --save // production dependency
* import $ from 'jquery'
* $('<h1>') // created a h1 tag

Use custom font with webpack

* Google font open sans. Download include in the app (font folder in src). Ttf.
* Online font converter: ttf woff woff2
* Make default font: index.scss

//webface fonts are commonly available, @font-face helps to have our own font

@font-face{

font-family: 'Open Sans';

font-weight: 300;

src: url('./fonts/OpenSans-Light.woff2') format('woff2'),

src: url('./fonts/OpenSans-Light.woff') format('woff'),

src: url('./fonts/OpenSans-Light.ttf) format(ttf)

}

@font-face{

font-family: 'Open Sans';

font-weight: 400;

src: url('./fonts/OpenSans-Regular.woff2') format('woff2'),

src: url('./fonts/OpenSans-Regular.woff') format('woff'),

src: url('./fonts/OpenSans-Regular.ttf) format(ttf)

}

@font-face{

font-family: 'Open Sans';

font-weight: 600;

src: url('./fonts/OpenSans-SemiBold.woff2') format('woff2'),

src: url('./fonts/OpenSans-SemiBold.woff') format('woff'),

src: url('./fonts/OpenSans-SemiBold.ttf) format(ttf)

}

* Specify the custom font as default font

html{

body{

font-family:'Open Sans', Arial, sans-serif;

font-weight: 300;

}

}

* import './index.scss' // put this in index.js before everything else
* load fonts in webpack(dev and prod). This will create a folder in dist called fonts. Npm install file-loader --save-dev

{

test: /\.(woff2|woff|ttf)$/,

use: [

loader: 'file-loader',

options: {

name: '[name].[ext]',

outputPath: 'fonts/'

}

]

}

Font Awsome

* SVG javascript framework the recommended way to integrate font awesome

npm install --save-dev @fontawsome/fortawsome-svg-core @fortawsome/free-solid-svg-icons @fortawsome/free-regualr-svg-icons @fortawsome/free-brands-svg-icons

* Import fontawsome
  + Library object allows us to specify which icons we want to use
  + Dom // replace any existing i tag with svg

import {library, dom } from '@fontawsome/fortawsome-svg-core'

import {faSpinner } from '@fortawsome/free-regualr-svg-icons'

* Use the icons

library.add(faSpinner);

* In template html file // fas font awesome solid set

<i class="fas fa-spinner fa-spin"> </i>

* The library will replace the id with svg. Watches dynamically, even at some point we add another i to the dom.

dom.watch();

Lint

* Linter refers to tools that analyze the source code to flag programming errors, bugs, stlylistic errors, and suspicious constructs.
* It can warn about syntax errors, uses of undeclared vars, calls to deprecated functions, formatting conventions, and much more.
* Static code analysis, that can help debug during production. It also helps with coding standards.
* ESLint // for javascript

npm insall eslint --save-dev

* In package.jso, write a script:

"lint": "eslint ."

* Create a config file for eslint. A file that is called .eslintrc and uses json format. we can write our rules, and eslint will inspect our codes. You can check the rules some of the are very common that we can check all by one line.
* ecmaVersion // By default 5
* "node": true // don’t give errors on require
* "browser"// not complain for document var
* "parser" // don't complain about class properties. npm install babel-estlint --save-dev
* "no-console" // 0 means turn it off ignore the console.log errors
* Linter plugin and eslint // A plugin for any editor for eslint

{

"extends": "eslint:recommended",

"parser": "babel-estlint",

"parserOptions": {

"ecmaVersion": 6,

"sourceType:: "module"},

"env":{

"node": true,

"browser": true, },

"rules":{

"no-console": 0

}}