Para rodar o webpack, primeiramente precisamos do node na versão 8.6.x pra frente

Versões do node par são mais estáveis

Através do seu terminal favorito entre dentro da pasta projeto-webpack/client e execute o comando:

npm install

Esse script compilara todos os arquivos e travara o termina aguardando futuras atualizações nos arquivos.

npm run watch

Porem esse projeto precisa de um servidor, que está disponibilizado na pasta do projeto mas que sera substituído pelo webpack dev server logo em seguida.

Para levantar o servidor entrar na pasta server e executar o comando

>npm start

Depois disso é só acessar o endereço <http://localhost:3000> no navegador

Para instalar o webpack, rodar o comando abaixo

npm install webpack@3.1.0 babel-core@6.25.0 --save-dev

O babel core está sendo baixado para poder compilar os arquivos

Criar arquivo webpack.config.js para criar um modulo do node

Para criar um modulo do node, executar o comando abaixo. O entry é onde vai conter o ponto de entrada da aplicação, ou seja o primeiro modulo da aplicação, o output é onde vai ser gravado o que foi resolvido a partir desse bundle inicial, o dirname retorna o caminho que o modulo está e sera armazenado em dist.

const path = require('path');

module.exports = {

entry: './app-src/app.js',

output: {

filename: 'bundle.js',

path: path.resolve(\_\_dirname, 'dist')

}

}

Para executar o webpack é necessário alterar o packge.json alterando a parte de scripts como exemplo abaixo, assim chamando o build-dev chamara o binário do webpack passando a configuração do config.js:

"scripts": {

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

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

},

Ao configurar essa parte script, é necessário colocar o código entre aspas duplas, caso contrário não irá funcionar.

Antes do webpack entrar em ação ele precisa pedir para o core do babel (nesse caso) para que ele processe os arquivos. A ligação entre o webpack e o babel é feita através de um loader, tem diversos loaders que fazem esse trabalho

>npm install babel-loader@7.1.0 --save-dev

Abaixo do output (lembrar de colocar, ), colocar o código abaixo

Esse código serve para que o antes do webpack fazer o processamento ele procurar todos os arquivos .js menos os da pasta node-modules e processar esses arquivos com o loader do babel. È possível ter mais de uma regra e usar mais de um loader para diferentes arquivos

module: {

rules: [

{

test: /\.js$/,

exclude: '/node-modules/',

use: {

loader: 'babel-loader'

}

}

]

}

Agora é só executar o comando abaixo e verificar quais os arquivos serão consederados para o bundle final

Npm run build-dev

Dentro da pasta client/dist, foi gerado o arquivo bundle.js. É um arquivo que contém todos os módulos usados pela aplicação concatenados.

O webpack por padrão já vem com o uglyfi para minificar os bundles gerados pela aplicação, para fazer essa operação e diferenciar os arquivos de produção basta junto com o build-dev, criar o build de produção como exemplo abaixo:

"build-prod": "webpack -p --config webpack.config.js"

O Uglify não suporta a sintaxe do ES2015, assim como as versões seguintes. O nosso projeto foi escrito em ES2017, ele certamente não será suportado. Com o Uglify, não conseguiremos minificar os arquivos escritos com o ES mais modernos.

Nós iremos nos abdicar do Uglify, e consequentemente, do parâmetro -p. No caso, o comando de build-prod ficará igual ao de build-dev, mas vamos alterar isso mais adiante.

Nós pediremos para o webpack usar outro módulo na realização desse processo de minificação e suporte a sintaxe do ES2015 (além de versões posteriores), os módulos. Usaremos o **Babili**, faremos a instalação no Terminal, com o seguinte comando:

>npm install babili-webpack-plugin@0.1.1 --save-dev

const babiliPlugin = require('babili-webpack-plugin');

let plugins = [];

if(process.env.NODE\_ENV == 'production') {

plugins.push(new babiliPlugin());

}

A variável process é acessada por qualquer módulo do NodeJS, que nos permite acessar o seu processo.

Em seguida, adicionaremos após process, a propriedade env, com ela teremos acesso a todas as variáveis de ambiente definidas no sistema operacional. Vamos testar se NODE\_ENV é igual a production. Se ele foi setado, com o método push(), jogaremos uma nova instancia new babiliPlugin().

A variável NODE\_END será setada quando chamarmos build project e for igual a production, então, o babiliPlugin() aplicará o processo. Caso você seja usuário de Windows, peço que tenha um pouco mais de paciência.

Para garantirmos que a mudança da variável de ambiente funcionará em qualquer plataforma, pediremos a ajuda de um módulo do NodeJS. No Terminal, acessaremos a pasta projeto webpack client, depois usaremos o comando npm install para o cross-env:

client caelum$ npm install cross-env@5.0.1 --save-dev

O módulo cross-env nos permite setar as variáveis de ambiente cross platform. Após a instalação, encontraremos o cross-env, abaixo do babili-webpack-plugin no package.json.

Depois, faremos um pequena alteração no build-prod, adicionando cross-env:

"scripts": {

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

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

"build-prod": "cross-env NODE\_ENV=production webpack --config webpack.config.js"

},

Quem será responsável por setar a variável de ambiente será cross-env.

Criamos a variável plugins com uma lista que começa vazia, que será adicionada em seguida no module.experts, o módulo de configuração do Webpack. Nele, teremos o entry, output, e abaixo do module, adicionaremos o plugins:

Em seguida, de volta ao Terminal, usaremos o seguinte comando:

client caelum$ npm run build-prod

O processo continuará como o esperado, funcionando tanto no Windows como em outros sistemas operacionais. Com isso, resolvemos o problema da minificação no build de produção.