Olá, eu sou o Flávio Almeida. Seja bem-vindo ao curso de Webpack da Alura! Começaremos mostrando o projeto que é o ponto de partida da nossa trama.

Inicialmente vamos digitar o seguinte comando no Terminal para instalar o projeto.

client caelum$ npm run watch

Vamos entrar na pasta do projeto que faremos o download mais adiante. O projeto está usando o Babel para fazer a compilação dos arquivos para podermos escrever o código com um código JavaScript mais atual e conseguir fazer transcompilação para uma versão mais antiga. Desta forma, ele funcionará em diferentes navegadores.

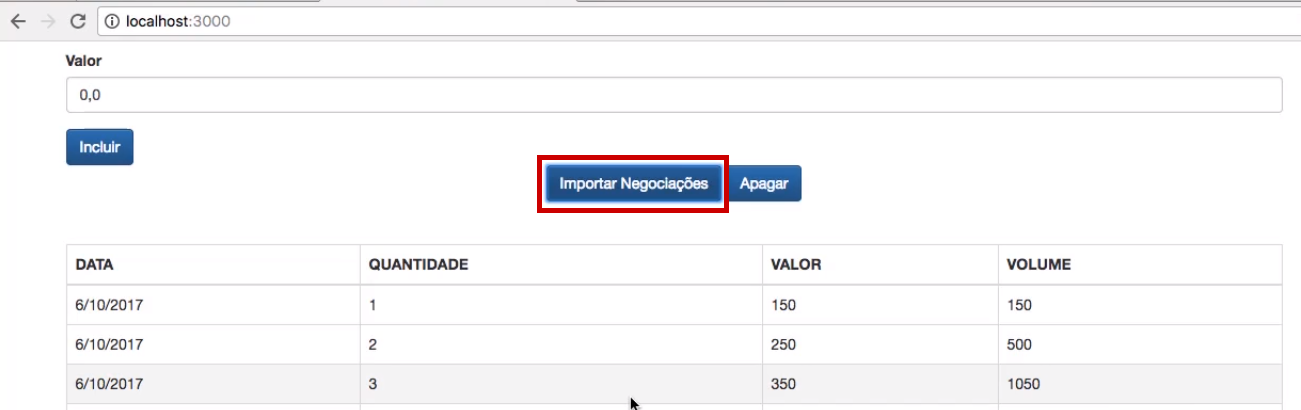
Para acessar o arquivo, antes, precisaremos subir o servidor usando o seguinte comando:

server caelum$ npm start

Após subirmos o servidor, o projeto se tornará público no navegador e será disponibilizado no endereço localhost:3000.



Nosso projeto é um sistema de bolsa de valores no qual podemos cadastrar ações e movimentações. Ao clicarmos no botão "Importar Negociações", ele importará as negociações do servidor.



Porém, a aplicação não está funcionando corretamente. Nós apresentamos um servidor criado por mim, que disponibiliza a API e será ele o responsável por tornar a aplicação acessível no navegador. Em um projeto profissional, é preferível utilizar um servidor que nos torne mais produtivo em um ambiente de desenvolvimento. Se analisarmos o código, estamos usando os módulos do ECMAScript 2015 (ES 6). Para carregar os módulos, usaremos o SystemJS - um carregador universal de módulos que será útil, mas possui a característica de carregar primeiramente o arquivo app.js.

Após carregar este primeiro módulo, ele resolverá todas as suas dependências. Em seguida, ele fará o download de cada um dos demais módulos separadamente, ou seja, se trabalharmos com 300 módulos, teremos 300 requisições.

Levaremos em consideração outro ponto, os arquivos bootstrap-theme.css e bootstrap.css foram baixados manualmente do [site do Bootstrap](https://getbootstrap.com/) e salvos dentro da pasta css. Se estamos usando o NodeJS para subir o servidor e o Babel também usa o NodeJS, podemos utilizar o npm (*Node package manager*, o gerenciador de pacotes do Node) para gerenciar as bibliotecas Front-end.

Não há um distinção entre builds de produção, por exemplo, podemos querer unificar nossos arquivos ou trocar a urlda API - do ambiente de desenvolvimento para o de produção. A Webpack é uma ferramenta que nos ajudará a resolver todos esses problemas. Trata-se de ***module bundler*** que "empacota" módulos para carregá-los na aplicação. Ele é voltado para a criação de single pages application (SPA), sendo utilizado pelo [Angular CLI](https://cli.angular.io/), [Vue CLI](https://github.com/vuejs/vue-cli) e [Create React App](https://github.com/facebookincubator/create-react-app).

É um pedaço de código recorrente nos frameworks famosos de SPA. Neste curso, aprenderemos o essencial de Webpack, veremos como ele funciona, sem utilizar um framework. Trabalharemos com o protótipo de uma SPA e consolidaremos o conhecimento na prática, desta forma, você se sentirá mais seguro sempre que for usar uma Webpack.

Acesse a [primeira atividade](https://cursos.alura.com.br/course/webpack/task/30470) desta aula, atenda os requisitos de infraestrutura e faça o [download do projeto](https://s3.amazonaws.com/caelum-online-public/webpack/stages/01-projeto-webpack.zip).

Até lá!

Para utilizarmos Webpack em ambiente de desenvolvimento é necessário ter a plataforma Node.js instalada. O [Node.js](https://nodejs.org/en/) é um ambiente JavaScript multiplataforma disponível para Linux, Mac e Windows. Para instalá-lo, siga as instruções abaixo:

**Caso você já tenha o Node instalado**

Se você já tem o Node instalado em sua máquina, verifique se ele está na versão **8.6.X**através do comando **node -v** no seu terminal, pois precisamos de uma versão atualizada do mesmo para podermos ter builds mais eficazes com Webpack. Não instale versões ímpares, essas não são tão estáveis quanto as versões pares.

**Instalação Node.js no Linux (Ubuntu)**

No Ubuntu, através do terminal (permissão de administrador é necessária) execute o comando abaixo:

sudo apt-get install -y nodejs

**ATENÇÃO USUÁRIOS LINUX!** em algumas distribuições Linux, pode haver um conflito de nomes quando o Node é instalado pelo apt-get. Nesse caso específico, no lugar do binário ser node, ele passa a se chamar **nodejs**. Isso gera problemas, pois a instrução **npm start** não funcionará, pois ela procura o binário node e não nodejs. Para resolver, use a seguinte instrução no terminal para subir o servidor:

nodejs server

Ou no Ubuntu:

sudo ln -s /usr/bin/nodejs /usr/bin/node

Depois o comando **npm start** funcionará conforme esperado.

É uma pena haver essa discrepância, mas fica aqui essa dica!

**Instalação Node.js no Windows**

Baixe o instalador clicando no grande botão de **Download**, diretamente da página do [Node.js](https://nodejs.org/en/). Durante a instalação, você apenas clicará nos botões para continuar o assistente. Não troque a pasta padrão do Node.js durante a instalação, a não ser que você saiba exatamente o que está fazendo.

**Instalação Node.js no MAC**

O [homebrew][2] é a maneira mais recomendada para instalar o Node.js em sua máquina, através do comando:

brew update

brew install node

Não usa homebrew? Sem problema, baixe o instalador clicando no grande botão de **Download**, diretamente da página do [Node.js](https://nodejs.org/en/).

**Editor recomendado: Visual Studio Code (Gratuito)**

Recomendo extremamente o uso do **Visual Studio Code** (**VSCode**), um editor **gratuito**criado pela Microsoft, disponível claro, para Windows, Linux e MAC.

Você pode baixá-lo [aqui](https://code.visualstudio.com/download).

**Carta branca para nossa jornada**

Agora que você já tem a infraestrutura necessária, já pode baixar o projeto no próximo exercício.

Precisamos de um projeto pronto para que possamos focar exclusivamente no Webpack. O instrutor Flávio Almeida (@flaviohalmeida) disponibilizou o projeto que utilizaremos no curso, diretamente do seu livro [Cangaceiro JavaScript - uma aventura no sertão da programação](https://www.casadocodigo.com.br/products/livro-cangaceiro-javascript).

O aluno curioso pode querer analisar o projeto, apesar desse não ser o foco do curso, e pode encontrar técnicas e códigos que não foram abordados aqui na plataforma. Não se preocupe, você não precisará entender o código do projeto, ele existe apenas para que motive o uso do Webpack.

Baixe o projeto [aqui](https://s3.amazonaws.com/caelum-online-public/webpack/stages/01-projeto-webpack.zip) descompactando-o logo em seguida.Através do seu terminal favorito entre dentro da pasta projeto-webpack/client e execute o comando:

npm install

Este comando baixará todas as dependências utilizadas pelo projeto. Depois de baixadas, execute o script:

npm run watch

Esse script compilará todos os arquivos e travará o terminal aguardando futuras atualizações nos arquivos. Porém, para que possamos acessar a aplicação ainda será necessário levantar o servidor. É um servidor disponibilizado com o projeto, mas você aprenderá ao longo do curso a substituí-lo pelo Webpack Dev Server, dando um ar ainda mais profissional ao projeto.

Para levantar o servidor é necessário abrir uma nova aba no terminal ou um novo terminal. Neste novo terminal, acesse a pasta projeto-webpack/server. Dentro da pasta, execute o comando:

npm start

Isso fará com que um servidor disponibilize nossa aplicação para o navegador. Para acessá-la, basta entrar com o endereço http://localhost:3000. Pronto, já temos o projeto e o servidor iniciado. Não se preocupe com essa complexidade do levantar o projeto, com auxilio do Webpack simplificaremos em muito esse processo.

Agora que tudo esta em seu lugar, podemos iniciar nosso no próximo vídeo. Aliás, ele questionará diversos pontos do projeto que baixamos.

Apesar de o Webpack ser altamente customizável, sua aplicação é voltada para Single Page Applications. Essas páginas que não recarregam durante seu uso tendem a carregar todos os scripts e arquivos CSS que dependem logo no primeiro carregamento e toda mudança de troca de página é feita por um sistema de rotas que manipula a única página carregada trocando seu conteúdo. Para o usuário, essa aplicação se comporta como se fosse uma aplicação nativa, pelo não carregamento durante o uso.

Por mais fabulosa que essa abordagem possa ser para o usuário, se o escopo da aplicação for grande, o primeiro carregamento será sofrível, primeiro pelo tempo de download de arquivos JS e CSS gigantes (concatenados e minificados) além do tempo para que o interpretador JavaScript realize o parser desses arquivos. Esse problema já não acontece com aplicações web tradicionais, pois cada página carrega os scripts e arquivos CSS de que precisam. Webpack soluciona esse problema das Single Page Applications.

Webpack permite dividir um bundle da aplicação em outros bundles menores que são carregados apenas quando forem utilizados pela aplicação, diminuindo assim o tempo de carregamento dessas aplicações e por conseguinte melhorando a experiência do usuário. Aprenderemos a realizar essas divisões, mas ainda há muita coisa para aprendemos antes de chegarmos a este ponto.

Vamos abrir o Terminal, acessaremos a pasta do desktop digitando os seguintes comando:

Caelums-iMac:~caelum$ cd Desktop/projeto-webpack/

Caelums-iMac:projeto-webpack caelum$ ls

client server

Vamos listarmos com ls, lembrando que temos a pasta client, na qual está o projeto, assim como a server do servidor disponibilizado para o projeto.

Em seguida, entraremos dentro da pasta client também com um comando no servidor:

npm run watch

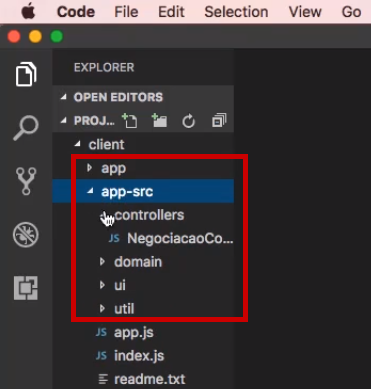
Com npm run watch, rodaremos o compilador em tempo real do Babel. Depois, abriremos uma nova aba do Terminal e entraremos dentro da pasta server:

npm start

Com o npm start, subiremos o novo servidor. Vamos digitar o endereço localhost:3000 no navegador e acessaremos o projeto. Se clicarmos em "Importar Negociações", importaremos as negociações da API.



Para introduzirmos o Webpack, começaremos a preparar a nossa estrutura. Vamos abrir o editor de texto (recomendamos o [Visual Studio Code](https://code.visualstudio.com/download) e apagaremos a pasta app, que é o resultado da compilação de app-src, em que ficam armazenados os arquivos originais do projeto.



Se você trabalha com ECMAScript ou React, vai encontrar uma pasta semelhante. Apagaremos a pasta app, porque a geração dos arquivos compilados pelo Babel será feita com o auxílio do Webpack mais adiante.

A seguir, acessaremos o index.html, abaixo da tag <div> com o id="negociacoes"encontraremos algumas tags <script> com o SystemJS:

<div id="negociacoes"></div>

<script src="node\_modules/reflect-metadata/Reflect.js"></script>

<script src="node\_modules/systemjs/dist/system.js"></script>

<script>

System

.import('app/app.js')

.catch(err => console.error(err))

</script>

Nós iremos remover o SystemJS e o comando que importa os módulos. O trecho será substituído pela tag <script> que importará o dist/bundle.js:

<script src="node\_modules/reflect-metadata/Reflect.js"></script>

<script src="dist/bundle.js"></script>

No caminho, especificaremos a pasta dist. Você pode questionar a existência da pasta dist e do arquivo recém adicionado, mas eles serão gerados futuramente pelo Webpack. Estamos preparando as bases para a aplicação funcionar bem com o Webpack.

O próximo passo será parar o servidor no Terminal, com o comando "Control + C" e teremos o seguinte retorno:

server caelum$

Acabamos de parar o processo de compilação em tempo real. Em seguida, dentro da pasta client, acessaremos o arquivo package.json:

"author": "",

"license": "ISC",

"dependencies": {

"reflect-metada": "^0.1.10",

"systemjs": "^0.20.12"

},

"devDependencies": {

"babel-cli": "^6.24.1",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-plugin-transform-es2015-modules-systemjs": "^6.24.1",

"babel-preset-es2017": "^6.24.1"

}

Em seguida, removeremos o systemjs e o babel-cli - a ferramenta de linha de comando que permite trabalharmos com o Babel. Nós não trabalharemos com essa ferramenta. Quem vai ser responsável por chamar o Babel e compilar o código, será o Webpack. Realizaremos a remoção de maneira elegante, por meio do Terminal, usando o comando a seguir:

client caelum$ npm uninstall babel-cli --save-dev

Removemos primeiro o babel-cli que está dentro de devDependencies. se retornarmos ao package.json, veremos que ele foi removido.

"devDependencies": {

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-plugin-transform-es2015-modules-systemjs": "^6.24.1",

"babel-preset-es2017": "^6.24.1"

}

Em seguida, vamos apagar o systemjs que está dentro de dependencies. De volta ao Terminal, digitaremos:

client caelum$ npm uninstall systemjs --save

Por enquanto, estamos apenas adaptando o projeto para trabalharmos com o Webpack. Se você usa o Angular CLI, react create app, Vue CLI, esses detalhes ficaram escondidos, neste curso, nós aprenderemos a essência do Webpack para entender cada um dos frameworks que o utilizam e nos sentirmos mais seguros. Por isso, começaremos por baixo.

Após as duas remoções, o próximo passo será instalar o Webpack. No caso, usaremos a versão 3.1.0, cuja homologação foi feita por mim. Não use versões anteriores, porque elas possuem bugs. Instalaremos também babel-core.

client caelum$ npm install webpack@3.1.0 babel-core@6.25.0 --save-dev

Nós desinstalamos o babel-cli e estamos este outro porque o Babel ainda será usado para a compilação dos arquivos. Nós vamos instalar o core, na versão 6.25.0 do Babel, que será chamado pelo Webpack para automatizar o processo. No Terminal, digitamos o --save-dev, por se tratar de uma dependêcia de desenvolvimento. Usaríamos apenas --save por ser uma dependência em tempo de produção.

Será feito rapidamente o download, em seguida, se acessarmos o arquivo package.json, veremos que o babel-core e o webpack foram instalados corretamente.

"devDependencies": {

"babel-core": "^6.25.0",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-plugin-transform-es2015-modules-systemjs": "^6.24.1",

"babel-preset-es2017": "^6.24.1",

"webpack": "^3.1.0"

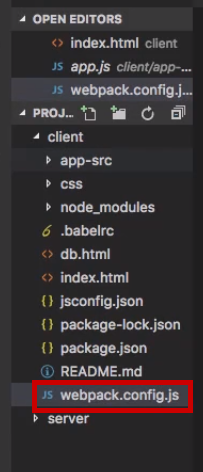
}

Temos o projeto, que não possui carregador de módulo, ou seja, sem o systemjs. Também não temos arquivos compilados, é o momento de configurar o Webpack para construir e empacotar os módulos da aplicação.

Faremos a primeira configuração do Webpack, na qual indicaremos qual módulo será carregado primeiramente pela aplicação. Desta forma, o Webpack conseguirá ver todas as dependências do módulo e assim, ele construirá o bundle, que é o arquivo final com todos os módulos da aplicação. No caso, o módulo que será carregado primeiramente é o app.js.

Frameworks como AngularJS, React ou VueJs possuem um módulo de aplicação que será carregado antes dos demais. Depois, com a ajuda do Webpack, serão desenvolvidas as dependências do módulo.

A seguir, dentro da pasta client, criaremos o arquivo webpack.config.js.



Se estamos criando um módulo no NodeJS, devemos adicionar a seguinte linha de código dentro webpack.config.js:

module.exports = {

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

}

Configuramos o módulo, que no caso, foi feito em um objeto. Nele, passamos o entry - o ponto de entrada - e especificamos qual será o primeiro módulo carregado. Definiremos ainda onde será gravado o bundle do final, que será resolvido no módulo inicial. Lembrando que o Webpack vai procurar todas as dependências e resolverá isso por nós, gerando um grande arquivo no fim. Isso será feito em output, que receberá o objeto filename: 'bundle.js'.

module.exports = {

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

output: {

filename: 'bundle.js',

}

}

Além disso, passaremos o path dentro do objeto, mas antes, criaremos a constante path com o caminho absoluto:

const path = require('path');

module.exports = {

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

output: {

filename: 'bundle.js',

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

}

}

Agora, no path do output, definiremos que ele é o resultado de path.resolve. Ele receberá a variável do NodeJS, chamada \_\_dirname, que informa qual a diretório do webpack.config. Em seguida, queremos concatenar a pasta dist, adicionada como segundo parâmetro do resolve(). Então o path.resolve() criará o caminho completo até a pasta dist, com base na pasta client.

Recapitulando: Se analisarmos o index.html, em que especificamos que o carregamento deve ser feito desde dist/bundle.js, com isso o bundle.js será criado dentro da pasta dist - localizada dentro de client, conforme especificamos no webpack.config.js.

Configuramos uma entrada e uma saída, agora, precisaremos encontrar um forma de executar o script. Para isto, faremos novos ajustes no package.json.

"scripts": {

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

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

},

O próximo passo será apagar o build e o watch, porque não estamos mais usando o Babel CLI, logo, não serão utilizados. Em seguida, adicionaremos o script build-dev, que é um build de desenvolvimento. Ele será responsável por chamar o webpack --config webpack.config.js. Se chamarmos o script build-dev do package.json, será executado o binário do webpack passando como configuração o arquivo webpack.config.js.

Voltaremos para o Terminal, digitaremos o seguinte comando:

client caelum$ ls

Será exibida uma lista com todos os arquivos da pasta client.

Caelums-iMac:client caelum$ ls

README.md index.html package.json

app-src jsconfig.json webpack.config.js

css node\_modules

db.html package-lock.json

Ainda na pasta client vamos usar o seguinte comando npm:

Caelums-iMac:client caelum$ npm run build-dev

> client@1.0.0 build-dev /Users/caelum/Desktop/projeto-webpack/client

> webpack --config webpack.config.js

Em seguida executaremos o comando e o Webpack não processará o arquivo, como veremos na mensagem exibida logo após no Terminal You may need an appropriate loader to handle this file (Em tradução livre: Você vai precisar de um loader apropriado para lidar com este arquivo):

ERROR in ./app-src/controllers/NegociacaoController.js

Module parse failed: /Users/flavioalmeida/Desktop/20/client/app-src/controllers/NegociacaoController.js Unexpected character '@' (5:0)

You may need an appropriate loader to handle this file type.

| import { getNegociacaoDao, Bind, debounce, controller, bindEvent } from '../util/index.js';

|

| @controller('#data', '#quantidade', '#valor')

| export class NegociacaoController {

|

@ ./app-src/app.js 1:0-77

No caso, o arquivo será @controller. O problema é que este projeto está utilizando os decorators existentes no Typescript, mas que podemos utilizar no JavaScript porque estamos rodando com Babel.

Porém, como o Webpack não entende que estamos utilizando a sintaxe do Babel, ele deixa de funcionar. Para que isso não aconteça, precisamos configurar no Webpack o Babel, para processar o arquivo antes de o Webpack tentar criar e processar os módulos.

Raissa, alterou seu arquivo package.json, porém ao executar o comando npm run build-dev o terminal indicou um erro no arquivo.

Vejamos parte do seu conteúdo:

/\* código anterior omitido \*/

"scripts": {

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

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

},

/\* código posterior omitido \*/

Consegue descobrir o erro de Raissa? Depois de pensar a respeito, compare seus achados com a resposta do instrutor.

O problema é que Raissa utilizou aspas simples e não aspas duplas para a propriedade build-dev e seu valor. Uma exigência é que as chaves e valores estejam entre as duplas, caso contrário, o arquivo será inválido.

Corrigindo o código:

/\* código anterior omitido \*/

"scripts": {

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

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

},

/\* código posterior omitido \*/

A ponte de ligação entre o Webpack e o babel-core é o **babel-loader**(<https://github.com/babel/babel-loader>), um carregador exclusivo voltado para o Babel. Esse loader lerá nossas configurações em client/.babelrc quando for executado.

Vamos instalar o loader através do terminal:

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

Isso ainda não é suficiente, precisamos fazer com que client/webpack.config.js utilize o loader que acabamos de baixar. Fazemos isso adicionando a configuração **module**. Dentro de module, podemos ter várias **rules** (regras), e cada regra pode usar um módulo específico quando aplicada.

Alterando nosso arquivo client/webpack.config.js:

// client/webpack.config.js

const path = require('path');

module.exports = {

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

output: {

filename: 'bundle.js',

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

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader'

}

}

]

}

}

Por enquanto, temos apenas uma regra. A propriedade test indica a condição na qual nosso loader será aplicado. Usamos a expressão regular /\.js$/ para considerar todos os arquivos que terminam com a extensão .js. Durante este processo, excluímos a pasta node\_modules, pois não faz sentido processar os arquivos dela. Por fim, dentro de use, indicamos o **loader** que será utilizado, em nosso caso o babel-loader.

Agora já podemos tentar executar mais uma vez nosso processo de build através do terminal:

npm run build-dev

E mais uma vez, recebemos um aviso nele:

Hash: b9e23a2ce60c22bf8d05

Version: webpack 3.0.0

Time: 286ms

Asset Size Chunks Chunk Names

bundle.js 3.87 kB 0 [emitted] main

[0] ./app-src/app.js 1.38 kB {0} [built] [1 warning]

WARNING in ./app-src/app.js

System.register is not supported by webpack.

O que será que foi dessa vez?

Quando usamos System.js, configuramos o client/.babelrc para transcompilar nossos módulos com um padrão compatível com o loader. Contudo, esse formato não é suportado pelo Webpack. A boa notícia é que, a partir da sua versão 2.0, Webpack já suporta por padrão o sistema de módulos do ES2015 (ES6). Dessa forma, não precisamos mais do módulo babel-plugin-transform-es2015-modules-systemjs e podemos removê-lo. Além disso, também podemos atualizar client/.babelrc para não fazer mais uso do plugin.

Removendo o módulo via npm:

npm uninstall babel-plugin-transform-es2015-modules-systemjs --save-dev

Agora, precisamos alterar client/.babelrc. Em plugins, deixaremos apenas o transform-decorators-legacy:

// client/.babelrc

{

"presets":["es2017"],

"plugins" : ["transform-decorators-legacy"]

}

Agora podemos gerar o build do nosso projeto que ele vai até o final, sem problema algum:

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.

Mesmo ainda sem um servidor rodando, se abrirmos client/index.html **diretamente no navegador**, nossa aplicação continuará funcionando.

Aprendemos a realizar um build de desenvolvimento, mas minificar scripts em ambiente de produção é uma boa prática. Aliás, ainda não temos uma separação clara entre esses dois ambientes. Chegou a hora de botar a casa em ordem.

O próximo passo será criar nosso build de produção, que consiste em minificar o bundle.js final. Felizmente, o Webpack já suporta esse processo automaticamente se passarmos o parâmetro adequado.

No package.json, adicionamos anteriormente o build de desenvolvimento, falta criar o build de produção. Faremos isso aproveitando a linha com o build-dev, que será semelhante a de build-prod que será adicionado a seguir:

"scripts": {

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

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

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

},

Observe que inserimos o parâmetro de -p e internamente ele vai usar o uglify, que não foi precisaremos baixar por ser uma dependência do Webpack. O Uglify é um módulo famoso, bastante utilizado para minificar arquivos JavaScript. Em seguida, acessaremos o Terminal, digitaremos o comando npm para saber se o build de dev está funcionando.

client caelum$ npm run build-dev

Veremos que tudo está funcionando corretamente. Depois, verificaremos os funcionamento do build-prod:

client caelum$ npm run build-prod

Porém, ao executarmos o programa, ele não será processado. Somos informados que o UglifyJS não aceitou o token:

ERROR in bundle.js from UglifyJs

Unexpected token: name (Negociacao) [bundle.js:75,4]

O Uglify não suporta a sintaxe do ES2015, assim como as versões seguintes. O nosso projeto foi descrito 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:

client caelum$ npm install babili-webpack-plugin@0.1.1 --save-dev

Estamos usando a versão homologada como Webpack. Nós instalamos anteriormente o Babel loader, agora, instalamos o plugin. Existe alguma diferença entre os dois? O **loader** vai trabalhar com cada arquivo separadamente, antes da criação do Bundle final, que pode sua vez, será utilizado pelo **plugin**. Nosso objetivo é minificar o Bundle final, então, faz sentido utilizar o Babili. Após a instalação, ao verificarmos o package.json, encontraremos babili-webpack-plugin:

"devDependencies": {

"babel-core": "^6.25.0",

"babel-loader": "^7.1.0",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-preset-es2017": "^6.24.1",

"babili-webpack-plugin": "^0.1.1",

"webpack": "^3.1.0"

}

O próximo passo será configurar o babili no webpack.config.js, porém, o plugin só será ativado quando estivermos em produção. Em desenvolvimento, ele não poderá ser desativado, porque não podemos gastar tempo com processo de minificação para desordenar o Bundle, e no momento de depurar (ou debugar) durante o desenvolvimento. Resolveremos isso realizando testes? Se uma variável de ambiente, por exemplo, node\_env - lembrando que o webpack é processado pelo NodeJS. Ele possui o valor production, caso contrário não é possível usar o plugin. No package.json, nós removemos -p a da linha em que temos build-prod, agora, incluiremo NODE\_ENV=production.

{

"name": "client",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

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

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

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

}

}

Informamos que a variável de ambiente será de produção (production).

**Atenção**: se você é usuário de Windows, a variável de ambiente não funcionará. Mais adiante mostrarei uma solução universal, que funciona em diferentes sistemas operacionais. Por enquanto, adotaremos essa solução.

Quando chamarmos build project, a variável de ambiente NODE\_ENV será setada. Em seguida, vamos importar o plugin no webpack.config.js, utilizando o mesmo nome do módulo importado do Terminal (babili-webpack-plugin).

const path = require('path');

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

let plugins = [];

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:

module: {

rules: [

{

test:/\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader'

}

},

]

},

plugins: plugins

Podemos dizer que a chave plugins recebe a variável plugins, porém, como estamos trabalhando com o mesmo nome, o ES2015 nos permite omitir o segundo plugins. Com a alteração, nosso código ficou assim:

module.exports = {

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

output: {

filename: 'bundle.js',

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

},

module: {

rules: [

{

test:/\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader'

}

},

]

},

plugins

}

Assim, nosso plugins ficará vazio. Faremos um teste, adicionando um if(process), logo após a variável plugins:

const path = require('path');

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

let plugins = [];

if(process){

}

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().

const path = require('path');

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

let plugins = [];

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

plugins.push(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.

No Terminal, usaremos o comando npm run build-dev, para verificarmos se tudo funciona corretamente:

client caelum$ npm run build-dev

Sabemos que ele não está minificado. O próximo comando será o npm prod:

client caelum$ npm run build-prod

Em seguida, se acessarmos o bundle.js, veremos que o arquivo foi minificado.

Para resolvermos o problema de setarmos variável de ambiente cross plataform, precisaremos de um módulo. No Terminal, usaremos o módulo cross-env, que vai auxiliará as variáveis a mudarem de ambiente. Faremos isso a seguir.

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.

"devDependencies": {

"babel-core": "^6.25.0",

"babel-loader": "^7.1.0",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-preset-es2017": "^6.24.1",

"babili-webpack-plugin": "^0.1.1",

"cross-env": "^5.0.1",

"webpack": "^3.1.0"

}

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.

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. No entanto, falta fazer uma melhoria. Estamos usando um servidor disponibilizado por mim para tornar a aplicação acessível - que é um piloto de uma single page application. Mas queremos criar algo com caracter mais profissional, usando um servidor consolidado pela comunidade para trabalhar com Webpack em ambiente de desenvolvimento. Veremos isso mais adiante.

Para trabalharmos em ambiente de desenvolvimento, quero usar um servidor que se integre com o Webpack, desta forma, não precisaremos criar um servidor próprio para tornar acessível nosso projeto.

No caso, usaremos o Webpack dev server, utilizado por diversos frameworks, single pages applications, pelos Command line interfaces (CLI) dos projetos. Primeiramente, adequaremos o projeto para trabalhados com Webpack dev server. Vale lembrar que a pasta server deve continuar existindo, porque nela encontraremos as APIs consumidas pela aplicação, um protótipo de uma single page applications.

No entanto, esta aplicação não vai tornar os arquivos estáticos acessíveis, como index.html e os arquivos com extensão JS.

Vamos acessar o arquivo server.js, dentro da pasta server.

Depois, entraremos em config/express.js, abaixo da variável express. Vamos comentar a linha referente ao clientPath, em que são compartilhados os arquivos estáticos.

var express = require('express')

,app = express()

,routes = require('../app/routes')

,path = require('path')

,bodyParser = require('body-parser');

/\*

app.set('clientPath', path.join(\_\_dirname, '../..', 'client'));

console.log(app.get('clientPath'));

app.use(express.static(app.get('clientPath')));

\*/

Agora, o back-end compartilhará a API para poder receber e enviar negociações, a pasta não será mais acessível no navegador. Se acessarmos o back-end, veremos que o servidor não serve mais os arquivos. Vamos pará-lo com pressionando "Ctrl + C + Enter" e em seguida, iremos reiniciá-lo com npm start. Se digitarmos localhost:3000 no navegador, nada será exibido. Esse era o nosso objetivo.

Como nossa aplicação é uma single page application, trata-se de um protótipo feito em JavaScript, vai rodar no Weppack server e a API está em outro server, adequaremos os endereços das requisições feitas pela aplicação.

Seguiremos o seguinte caminho: "client > domain > NegociacaoService.js". Em seguida, em todos os endereços da API, adicionaremos http://localhost:3000/, por exemplo, dentro do ObtemNegociacoesDaSemana():

obtemNegociacoesDaSemana() {

return this.\_http

.get('http://localhost:3000/negociacoes/semana')

.then(

dados =>

//código omitido

)

}

Faremos a mesma alteração no método obtemNegociacoesDaSemanaAnterior() e obtemNegociacoesDaSemanaRetrasada(). O próximo passo será fazer alterações no app.js, dentro do fetch(), abaixo da constante config:

const config = {

method,

headers,

body

};

fetch('http://localhost:3000/negociacoes', config)

.then(() => console.log('Dado enviado com sucesso'));

Por enquanto, esse endereço está fixo, por ser de desenvolvimento. Veremos como modificá-lo em produção mais adiante, mas a alteração feita, adequamos o projeto para começarmos a configurar o Webpack server.

A seguir, vamos instalar a Webpack da dev-server. No Terminal, usaremos o seguinte comando:

client caelum$ npm install webpack-dev-server@2.5.1 --save-dev

Após a instalação ser finalizada, quando analisarmos o arquivo package.json, encontraremos o webpack-dev-server listado.

"devDependencies": {

"babel-core": "^6.25.0",

"babel-loader": "^7.1.0",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-preset-es2017": "^6.24.1",

"babili-webpack-plugin": "^0.1.1",

"cross-env": "^5.0.1",

"webpack": "^3.1.0"

"webpack-dev-server": "^2.5.1"

}

Agora, como chamaremos o servidor? Nós adicionaremos um novo npm script, que receberá o nome de start.

"scripts": {

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

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

"build-prod": "NODE\_ENV=production webpack -p --config webpack.config.js",

"start": "webpack-dev-server"

},

O responsável por fazer o build do projeto será webpack-dev-server, que chamará o webpack.config.js. Antes de rodarmos o servidor, vamos **apagar** a pasta dist, que contém o bundle.js. Como ainda não configuramos o ambiente de produção, estamos fazendo um build de desenvolvimento. No terminal, daremos npm start e certficaremos que webpack-dev-server rodou o script do webpack, porque ele rodou o processo executado.

Porém, uma diferença será que não acessaremos mais a aplicação pelo http://localhost:3000, no caso, vamos utilizar http://localhost:8080. No navegador, digitaremos esse endereço, e o arquivo será compartilhado pelo webpack-dev-server. Porém, se clicarmos no botão "Importar Negociações", a tabela com os dados não será exibida.

Se acessarmos o Console, veremos que ele não conseguiu carregar o arquivo bundle.js, porque não criamos uma pasta dist. Isto aconteceu porque o webpack-dev-server cria o arquivo bundle.js em memória. Com isso, qualquer alteração feita na aplicação, o novo arquivo será gerado, sem ser gravado no disco. Porém, o arquivo foi criado diretamente em localhost:8080/bundle.js.

O endereço 8080/bundle.js criado pelo webpack-dev-server, mas o endereço que deve ser levado em consideração é 8080/dist/bundle.js para ficar condizente com o código-fonte.

<script src="node\_modules/reflext-metadata/Reflect.js"><script>

<script src="dist/bundle.js"></script>

Como podemos instruir o webpack-dev-server a criar o arquivo na memória, mas no endereço dist/bundle.js? No webpack.config.js, dentro do output, adicionaremos publicPath. Nele, configuraremos o dist que será acessado pelos usuários.

module.exports = {

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

output: {

filename: 'bundle.js',

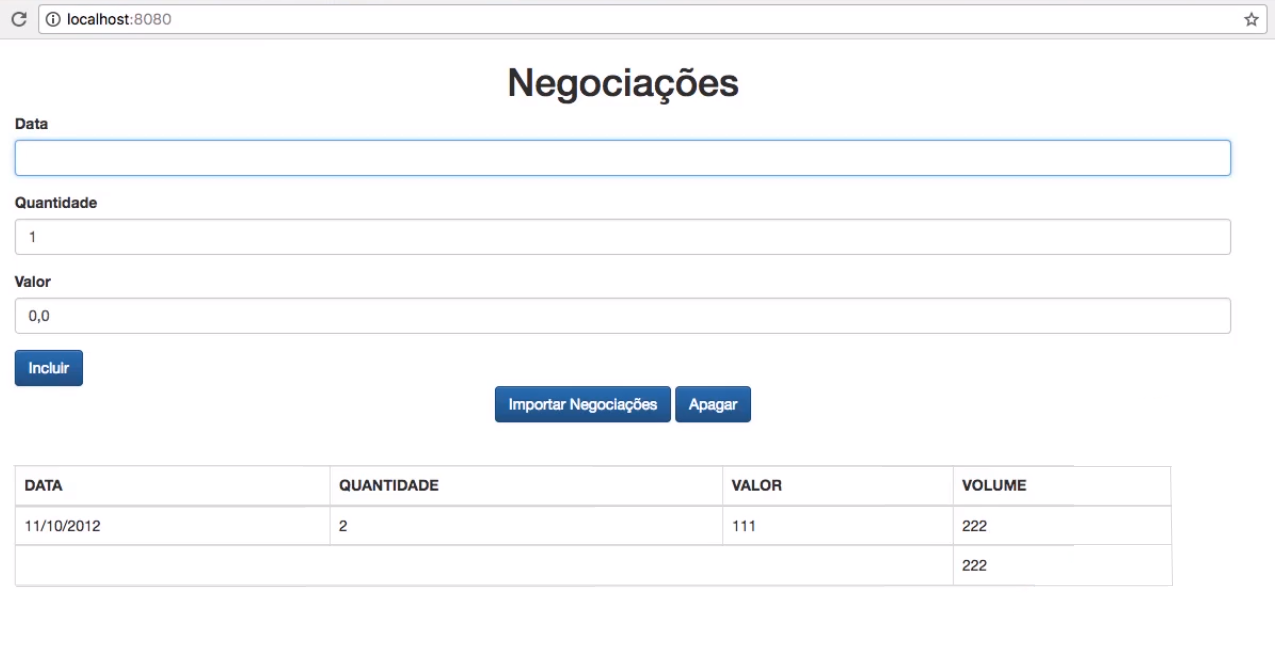
path: path.resolve(\_dirname, 'dist'),

publicPath: 'dist'

},

//...

Agora será considerado será considerada o caminho dist/bundle.js. No Terminal, vamos parar novamente o servidor, e depois, usar o comando client caelum$ npm start. Após subirmos o servidor, tentaremos acessar novamente o endereço localhost:8080 no navegador.



A tabela será exibida e o bundle.js não vai estar na dist, vai ser apenas criado em memória pelo webpack-dev-server. Nós achamos que estamos acessando o bundle.js, porém, estamos acessando uma API que está gerando esse arquivo em memória.

No projeto, abriremos o NegociacaoController.js , e adicionaremos alert() antes do try.

@bindEvent('submit', '.form')

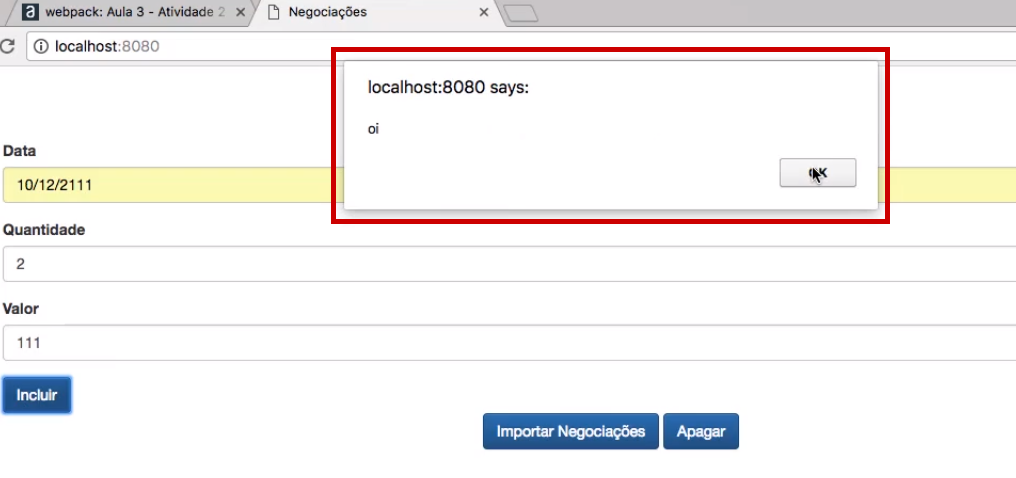
@debounce()

async adiciona(event) {

alert('oi');

//...

Ao realizarmos a gravação, automaticamente o webpack-dev-server irá compilá-lo e gerar um novo bundle. No navegador, será desnecessário inclusive descarregar a página. Se preenchermos os campos do formulário, ao clicarmos no botão "Incluir", será exibida a mensagem de alerta:



Esta geração automática do novo bundle é bastante interessante em ambiente de desenvolvimento, por isso, estamos utilizando o webpack-dev-server. A seguir, removeremos o alert() do NegociacaoController.js e continuaremos com nosso projeto.

Faremos mais melhorias usando o Webpack. No nosso projeto, a pasta css tem arquivos do bootstrap que são importados no index.html.

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width">

<title>Negociações</title>

<link rel="stylesheet" href="css/bootstrap.css">

<link rel="stylesheet" href="css/bootstrap-theme.css">

</head>

Esses arquivos forma baixados manualmente do site do [Bootstrap](https://getbootstrap.com/). Nós baixamos e descompactamos os arquivos que foram descompactados e movidos para a pasta css. Poderíamos ter adicionado um CDN (Contend Delivery Network) e torcer para não ser barrado pelo firewall.

Se o nosso ambiente de desenvolvimento usar o NodeJS, podemos usar o npm - um gerenciador de pacotes do Node - para gerenciar os pacotes Front-End (como Bootstrap e Angular). É o que faremos a seguir: nós vamos configurar o Webpack para que ele entenda a importação de módulos baixados do Node, armazenados na pasta node\_modules. Começaremos fazer os primeiros preparativos para a realização desse processo, mesmo em uma aplicação na qual trabalhamos unicamente com JavaScript. Você vai poder utilizar o conhecimento obtido aqui em outros frameworks.

Para começar, vamos apagar os arquivos bootstrap-theme.css e bootstrap.css da pasta css. Também iremos removê-los do arquivo index.html, ou seja, eles não serão mais importados na tag <head>.

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width">

<title>Negociações</title>

</head>

Em seguida, voltaremos a rodar o servidor com o comando npm start, e depois, carregaremos o projeto no navegador. Agora, o formulário fcará todo desconfigurado.



Usaremos o npm para gerenciar os pacotes do front-end, no passado, era comum a utilização da ferramenta **Bower**. Ela foi descontinuada, não sendo mais recomendada a sua utilização, inclusive, você encontrará essa recomendação no site do Bower - justamente porque npm assumiu essa parte de gerenciamento de dependências do Front-end.

Vamos parar novamente o servidor. Dentro da pasta do projeto Webpack client, usaremos o npm install bootstrap:

client caelum$ npm install bootstrap@3.3.7 --save

Será feito o download, o arquivo booststrap vai ser adicionado na pasta node\_modules. Como faremos para instruir durante o processo de build da aplicação para que o Webpack importe e carregue o bootstrap.

Como carregaremos o bootstrap na aplicação? Acessaremos o seguinte caminho "node\_modules > bootstrap > dist > css", a partir daí, encontraremos os arquivos do boostrap que improtaremos. Se você trabalha com React ou com Angular, arquivos .css são tratados como módulos - isto significa que um componente do Angular importa um CSS, que está atrelado a ele.

No React, podemos importar um CSS que também está atrelado a ele, dentro de um componente. Esta abordagem é interessante porque podemos dizer para um componente quais são os arquivos de estilos que operam sobre ele. Nós faremos o mesmo no projeto.

Se abrirmos a pasta app-src, o arquivo módulo app.js será o primeiro a ser carregado na aplicação. Nele, importaremos os arquivos CSS como se fossem módulos. Adicionaremos import usando uma sintaxe diferente.

import { NegociacaoController } from './controllers/NegociacaoController.js';

import { Negociacao } from './domain/index.js';

import 'bootstrap/dist/css/bootstrap.css';

import 'bootstrap/dist/css/bootstrap-theme.css';

Quando o arquivo bundle.js for gerado, ele verá que foi criado o import que está recebendo a string. Como não utilizamos o ./ antes do endereço - o que indicaria a necessidade de buscar em um determinado diretório, como em import './booststrap/dist/css/bootstrap.css', deveriamos buscar no diretório a pasta bootstrap.

No caso, o Webpack compreenderá que a tarefa será procurar a pasta bootstrap dentro de node\_modules. Isso só será possível porque não especificamos outro caminho no endereço antecedido por ./. Por convenção, ele entende o que deve ser feito.

Ao carregarmos a pasta,o booststrap já seja aplicado. No Terminal, usaremos o comando npm start para subir webpack-dev-server. Porém, no momento de processamento do arquivo e da geração do bundle, receberemos uma mensagem de erro.

ERROR in ./node\_modules/bootstrap/dist/css/bootstrap.css

Module parse failed: /Users/flavioalmeida/Desktop/20/client/node\_modules/bootstrap/dist/css/bootstrap.css Unexpected token (7:5)

You may need an appropriate loader to handle this file type.

Somos informados que não foi possível parsear o booststrap.css, que não é um módulo, mas é tratado como tal. O Webpack nos informa que para a realização da tarefa é necessário um parser, porque ele não é capaz de lidar com CSS como módulos. Para que isso seja feito precisaremos do loader apropriado para lidar com o arquivo.

Por exemplo, quando trabalhamos com React, usamos o react-create-app para montar a infraestrutura por debaixo dos panos.

O próximo passo, será instalar os módulos com o seguinte comando no Terminal:

client caelum$ npm install css-loader@0.28.4 style-loader@0.18.2 --save-dev

Veremos qual é a responsabilidades dos loaders que estamos instalando. O css-loadervai ler o arquivo CSS completamente e, depois, vai transformar todas as suas propriedades em um JSON - um objeto JavaScript. Desta forma, outros loaders do Webpack possam extrair informações do objeto já parseado.

O style-loader vai usar essa informação do objeto transformado em JavaScript e vai torná-lo um CSS inline, que ficará no navegador. Ou seja, os dois elementos trabalharão em conjunto.

Após carregarmos os módulos, acessaremos o package.json:

"devDependencies": {

"babel-core": "^6.25.0",

"babel-loader": "^7.1.0",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-preset-es2017": "^6.24.1",

"babili-webpack-plugin": "^0.1.1",

"cross-env": "^5.0.1",

"css-loader": "^0.28.2",

"style-loader": "^0.18.2",

"uninstall": "0.0.0",

"webpack": "^3.1.0"

"webpack-dev-server": "^2.5.1"

}

Como faremos a configuração dos dois? No webpack.config.js, existe uma regra para lidarmos com babel-loader teremos uma nova regra, na qual teremos um test. Nele, indicaremos a extensão do arquivo que receberá a ação do loader.

module.exports = {

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

output: {

filename: 'bundle.js',

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

publicPath: "dist"

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader'

}

},

/\* NOVA REGRA AQUI \*/

{

test: /\.css$/,

loader: 'style-loader!css-loader'

}

]

},

plugins

}

No caso, teremos dentro do test teremos a extensão .css, sem falar a exclusão da pasta node\_modules porque o bootstrap está lá. Vamos configurar o loader também, no caso, trabalharemos com style-loader!css-loader.

Observe que adicionamos !, esse sinal indica que um loader será aplicado depois do outro, mas vale ressaltar que o processo é feito da direita para esquerda. Ou seja, primeiro será aplicado do css-loader e, depois, style-loader.

Em seguida, vamos testar a aplicação. No Terminal, rodaremos o comando npm start. Porém, quando rodamos o servidor, recebemos outra mensagem de erro. Resolvemos o problema do carregamento do CSS como módulo, porém, agora ele tem dificuldade de carregar os arquivos de fonte do bootstrap, que não são CSS. Isso significa que precisaremos de loader apropriado para carregar fontes, a seguir instruiremos o Webpack para fazer isso.

No momento da importação do bootstrap, ele terá que ler os arquivos, que serão transformados no CSS inline, em seguida, vai extrair os arquivos de imagem e movê-los para pasta dist. Teremos que configurar um loader para cada tipo de fonte trabalhada.

Nós utilizaremos dois loaders documentados no Webpack: url-loader e file-loader. Vamos instalá-los no Terminal:

npm install url-loader@0.5.9 file-loader@0.11.2 --save-dev

Esse dois loaders trabalharão em conjunto porque o url-loader vai ver a url e extrair a fonte, em seguida, irá copiá-la para o local apropriado. Porém, algumas fontes receberão um tratamento especial, por isso, a necessidade do file-loader. Nós criaremos no webpack.config.js várias regras para lidar com diferentes tipos de fontes utilizadas. Esse tipo de configuração é facilmente encontrada por quem trabalha com o Webpack e bootstrap, você irá encontrá-la logo abaixo:

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader'

}

},

{

test: /\.css$/,

loader: 'style-loader!css-loader'

},

{

test: /\.(woff|woff2)(\?v=\d+\.\d+\.\d+)?$/,

loader: 'url-loader?limit=10000&mimetype=application/font-woff'

},

{

test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/,

loader: 'url-loader?limit=10000&mimetype=application/octet-stream'

},

{

test: /\.eot(\?v=\d+\.\d+\.\d+)?$/,

loader: 'file-loader'

},

{

test: /\.svg(\?v=\d+\.\d+\.\d+)?$/,

loader: 'url-loader?limit=10000&mimetype=image/svg+xml'

}

]

},

plugins

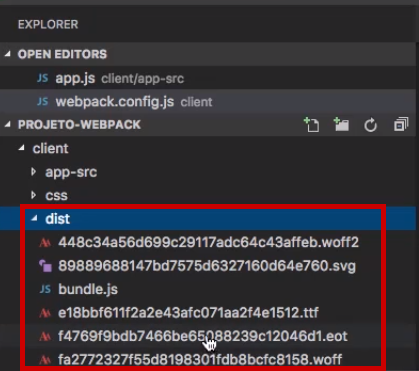
}

Entre as fontes trabalhadas estão: woff, ttf, eot e svg. Algumas usarão url-loader, enquanto outras usarão file-loader, tratam-se de experessões regulares um tanto elabordas, mas já são bastante utilizadas.

Vamos salvar, e no Terminal, rodaremos o webpack-dev-server. Em seguida, testaremos a aplicação no navegador, tudo estará funcionando corretamente.



Se voltarmos ao Terminal e reiniciarmos novamente o servidor com npm run build-dev, veremos que será gerado um arquivo bundle.js dentro da pasta dist, juntamente outros arquivos de cada fonte utilizada.



O Webpack entendeu que além de importar o CSS, queremos ter acesso aos arquivos das fontes. Ele extraiu cada fonte por meio dos loaders e colocou dentro da pasta dist, quando formos distribuir a aplicação copiaremos a pasta dist, levaremos o bundle.jsjuntamente com as fontes.

Nossa aplicação está funcionando conforme o esperado, porém, ainda faltam coisas para fazer. Por enquanto, deixaremos a pasta dist, mesmo trabalhando com webpack-dev-server. Nós só vamos gerar essa pasta quando estivermos em ambiente de desenvolvimento ou no build de produção.

Nós vimos como importar CSS direto da pasta node\_modules, considerando-o como um módulo. Fizemos tudo isso com a mágica dos loaders.

Mas se quisermos importar o CSS que não está na pasta node\_modules, por exemplo, um arquivo gerado por mim. Vamos criar um arquivo novo, que receberá o nome meucss.css, dentro da pasta css:

table {

box-shadow: 5px 5px 5px black

}

Como faremos para que este CSS seja aplicado assim no momento do carregamento. No app.js, adicionaremos um import:

import { NegociacaoController } from './controllers/NegociacaoController.js';

import { Negociacao } from './domain/index.js';

import 'booststrap/dist/css/bootstrap.css';

import 'bootstrap/dist/css/bootstrap-theme.css';

import 'css/meucss.css';

Porém, dá forma como elaboramos o código, ele acreditará que estamos procurando um módulo dentro de node\_modules. No entanto, queremos acessar a pasta css. Teremos que especificar isso no caminho adicionando ../:

import '../css/meucss.css';

O próximo passo ser colocar o servidor do webpack-dev\_server para rodar, recarregaremos a página, e teremos adicionado uma sombra na lateral da tabela.



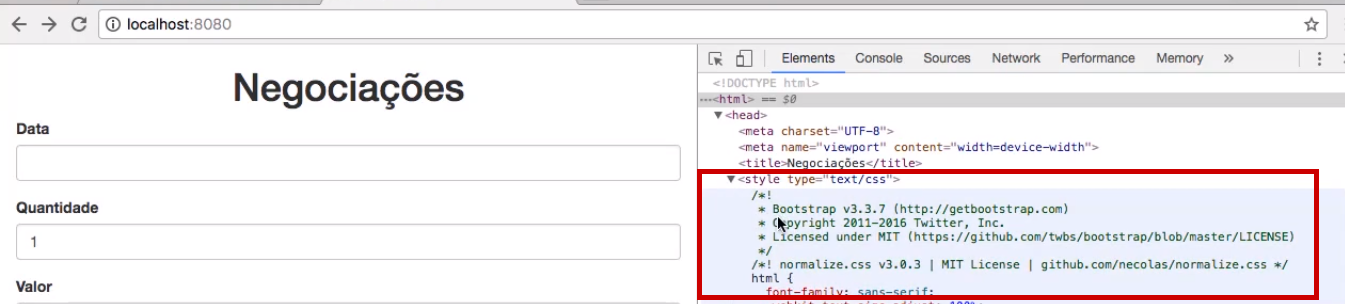
Em uma aplicação que trabalhamos com componentes - que não é uma o nosso caso -, cada um deles pode importar os módulos, tanto JS como CSS, que são suas dependências. Desta forma, ao analisarmos um componente, fica claro do que ele depende, inclusive a parte do estilo.

Porém, existe algo curioso que mostraremos mais adiante.

É o momento de revelarmos um segredo: nosso projeto tem um pequeno problema que pode ter passado despercebido. Se recarregarmos a página no navegador, ela será exibida sem o estilo carregado nos instantes iniciais.

Isto ocorre devido ao **fouc**, (flash of unstyled content, ou seja, flash de um conteúdo não estilizado), o que não deveria acontecer em uma aplicação. O problema é que se analisarmos o arquivo bundle.js da pasta dist, veremos que os arquivos CSS foram adicionados automaticamente. Todo arquivo CSS gerada, o Webpack agrupou em um único Bundle arquivos de diferentes extensões.

No caso, por meio do JS, o CSS foi adicionado como tag <style> do navegador. Se insecionarmos a página, encontraremos os arquivos CSS inline, como o style do bootstrap.css. Em um único bundle.js, ele colocou todos os arquivos empacotados juntos.



O CSS será aplicado pelo JavaScript, a menos que este adicione dinamicamente a string (que é um CSS) incluindo a tag <style> no navegador. Esse processo pode demorar e, por isso, o usuário vê o flash of Unstyled Content. Um especialista em Front-end poderia desistir de usar o Webpack, porque o navegador consegue carregar os arquivos CSS usando a tag link. Existem diversas otimizações que gostaríamos de aproveitar.

Nós temos que instruir o Webpack para que ele trabalhe nesse sentido, sem adicionar o CSS do bundle.js. A solução é deixar os arquivos separados, como style.css, com todos estilos concatenados. Em seguida, podemos importá-lo na página e usufruir do carregamento rápido.

O próximo passo será adicionar uma tag <link> dentro do index.html e apontaremos dist/style.css - que geraremos a seguir.

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width">

<title>Negociações</title>

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

</head>

Porém, durante o processo de build, o Webpack conseguirá separá-los, deixando os arquivos JavaScript dentro do bundle.js. Desta forma, resolveremos com o FOUC. Após adicionarmos <link>, acessaremos o Terminal para instalar o extract-text-webpack-plugin:

npm install extract-text-webpack-plugin@3.0.0 --save-dev

Você deve utilizar a mesma versão usada no curso, porque as anteriores vão ter bugs com o Webpack 3, no qual os CSS serão carregados na ordem contrária.

O próximo passo será fazer a configuração no webpack.config.js, importando extractTextPlugin:

const path = require('path');

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

const extractTextPlugin = require('extract-text-webpack-plugin');

O plugin será usado tanto em desenvolvimento como em produção, por isso, ele será o primeiro a ser listado na lista.

const path = require('path');

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

const extractTextPlugin = require('extract-text-webpack-plugin');

let plugins = []

// ADICIONANDO O PLUGIN,

// MAIS ABAIXO, MODIFICAÇÃO DO LOADER

plugins.push(

new extractTextPlugin("styles.css")

);

Adicionamos o extractTextPlugin() e passamos como parâmetro styles.css. Porém, isso não é o suficiente, teremos que alterar um dos loaders padrões do CSS, localizado no seguinte trecho:

module.exports = {

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

output: {

filename: 'bundle.js',

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

publicPath: "dist"

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader'

}

},

{

test: /\.css$/,

loader: 'style-loader!css-loader'

},

//...

Na linha loader: 'style-loader!css-loader', substituiremos o loader por chave use. Nós vamos usar o extractTextPlugin.extract(). Ele receberá um objeto JavaScript, em que passaremos o fallback informaremos qual loader será usado em casos de falha, além disso, especificaremos o loader padrão no use.

test: /\.css$/,

use: extractTextPlugin.extract({

fallback: 'style-loader',

use: 'css-loader'

})

Nosso use é o resultado do extractTextPlugin, em caso de falha será usado style-loader, se tudo funcionar corretamente css-loader.

Em seguida, usaremos o comando npm run build-dev no Terminal, dentro da pasta dist, ele criará o styles.css. Depois, como comando npm start, a aplicação funcionará corretamente. Se abrirmos o código fonte da página, ele acessará dist/styles.css, arquivo criado recentemente.

Quando fazemos o build, ele vai gerar o arquivo separado.

Você pode ter percebido que surgiu outra pequena falha no projeto. Se fizermos o build em ambiente de produção no Terminal, será gerada a pasta dist, com um bundle.jsminificado e o styles.css. Porém, se abrirmos o arquivo de estilo, veremos que ele não está minificado.

Quando utilizamos o plugins para extrair o extractTextPlugin, consequentemente, extrair os arquivos CSS importado modularmente e colocar todos num arquivo <style>separado e usufruir da tag <link> no navegador - totalmente otimizada para essa finalidade. Nesse processo, o CSS acabou não sendo minificado.

Nós resolveremos isso utilizando dois plugins, o primeiro deles será [optimize-css-assets-webpack-plugin](https://github.com/NMFR/optimize-css-assets-webpack-plugin). Ele receberá a responsabilidade por processar o style.css gerado antes de ser gravado na pasta de build. Porém, ele nao sabe processar o CSS, ainda será preciso ter um minificador CSS, no caso, usaremos o [cssnano](http://cssnano.co/). Podemos combinar os dois plugin com Webpack, juntamente com cssnano, que pode ser usado em qualquer sistema. Faremos a instalação pelo Terminal:

npm install optimize-css-assets-webpack-plugin@2.0.0 --save-dev

npm install cssnano@3.10.0 --save-dev

Os dois serão usados em ambiente de desenvolvimento. Se acessarmos o package.jsonencontraremos o optimize-css-assets-webpack-plugin e o cssnano:

"devDependencies": {

"babel-core": "^6.25.0",

"babel-loader": "^7.1.0",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-preset-es2017": "^6.24.1",

"babili-webpack-plugin": "^0.1.1",

"cross-env": "^5.0.1",

"css-loader": "^0.28.4",

"cssnano": "^3.10.0",

"extract-text-webpack-plugin": "^3.0.0",

"file-loader": "^0.11.2",

"optmize-css-assets-webpack-plugin": "^2.0.0",

"style-loader": "^0.18.2",

"uninstall": "0.0.0",

"url-loader": "^0.5.9",

"webpack": "^3.1.0"

"webpack-dev-server": "^2.5.1"

}

Em seguida, iremos configurar os loaders no webpack.config.js, começaremos adicionando as importações.

const path = require('path');

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

const extractTextPlugin = require('extract-text-webpack-plugin');

const optimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');

let plugins = [];

//...

Por enquanto, não vamos importar cssnano, você deve apenas entender que o optmize-css-assets-webpack-plugin só pode ser adicionado ao Webpack quando for ambiente de produção. Por isso, faremos imediatamente o registro do babiliPlugin dentro do if - assim como qualquer plugin de produção-, caso contrário, ele entrará em desenvolvimento.

Em ambiente de desenvolvimento, não queremos que o arquivo seja minificado, pelo contrário, é melhor poder analisá-lo.

const optimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');

let plugins = []

plugins.push(

new extractTextPlugin("styles.css")

);

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

plugins.push(new babiliPlugin());

plugins.push(new optimizeCSSAssetsPlugin({

cssProcessor: require('cssnano'),

cssProcessorOptions: {

discardComments: {

removeAll: true

}

}

}));

}

O cssProcessor que utilizaremos será o cssnano, por isso, estamos fazendo o require() diretamente. Configuramos também cssProcessorOptions, que recebe um objeto JavaScript com as configuração do pré-processador. Adicionamos dentro do objeto as propriedade discardComments, que receberá o removeAll().

O que foi passado para cssProcessoOptions não estará relacionado com optimizeCSSAssetsPlugin, mas sim com cssProcessor importado. Após pedirmos para que todos os comentários fossem removidos, porque não faz sentido minificá-los em produção.

Temos ainda a propriedade canPrint, que vamos configurar como true, que adicionaremos logo abaixo do optimizeCSSAssetsPlugin. Com ela, indicaremos se o plugin pode exibir informações no Console.

//...

plugins.push(new optimizeCSSAssetsPlugin({

cssProcessor: require('cssnano'),

cssProcessorOptions: {

discardComments: {

removeAll: true

}

},

canPrint: true

}));

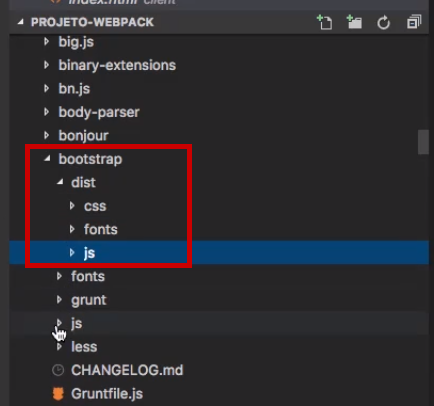
}

No Terminal, rodaremos o comando de build-prod, depois, ao acessarmos style.css, veremos que toda a parte de estilo foi minificada. Se rodarmos os comandos build-dev, o arquivo estará minificado, enquanto com o build-prod, descobriremos que ele não está. Abriremos o arquivo index.html no navegador, confirmaremos que está tudo funcionando.

Resolvemos a questão do Flash of Unstyled Content e a parte de minificação.

Vimos como tratar arquivos CSS como módulos e depois, mantê-los separados do bundle.js, sendo a melhor opção importá-lo pela tag <link>. Porém, o que devemos fazer para importar scripts e tratá-los como módulos da aplicação. Normalmente, os scripts já são importados pela tag <script>. Nós não queremos fazer isso, porque queremos que o Webpack gere o bundle.js, e processe os módulos.

Dentro do node\_modules, temos a pasta bootstrap - que possui a dist, onde encontraremos vários plugins js do bootstrap.



No caso, vamos carregar apenas o plugin do modal, que não será necessário no projeto, mas ficará pronto para o caso de fazermos o modal, ele já deve estar carregado. Por exemplo, veremos como carregar um script do Bootstrap na aplicação.

Assim que a aplicação for carregada no app.js, após importarmos o CSS do bootstrap, será a vez ser de importar o modal.js .

import { NegociacaoController } from './controllers/NegociacaoController.js';

import { Negociacao } from './domain/index.js';

import 'booststrap/dist/css/bootstrap.css';

import 'bootstrap/dist/css/bootstrap-theme.css';

import 'bootstrap/js/modal.js';

import 'css/meucss.css';

O Webpack vai entender que estamos procurando model.js, dentro de "node\_module > bootstrap > js". Se subirmos novamente o servidor no Terminal e recarregarmos a aplicação, veremos que algo deixou de funcionar. Se investigarmos no Console, encontraremos um aviso informando que jQuery não foi definido.

Uncaught Error: Bootstrap's JavaScript requires jQuery

Isto está acontecendo porque quando carregamos o modal do Booststrap, este dependerá do jQuery para que o modal funcione. Trata-se de uma dependência do booststrap. Então precisaremos instalar o jQuery:

npm install jquery@3.2.1 --save

Em seguida, se analisarmos a pasta node\_modules, acessaremos "jquery > dist", e depois, encontraremos os arquivos jquery. Iremos importar o arquivo no app.js, antes do modal.js.

import { NegociacaoController } from './controllers/NegociacaoController.js';

import { Negociacao } from './domain/index.js';

import 'booststrap/dist/css/bootstrap.css';

import 'bootstrap/dist/css/bootstrap-theme.css';

import 'jquery/dist/jquery.js';

import 'bootstrap/js/modal.js';

import 'css/meucss.css';

Porém, se tentarmos rodar novamente a aplicação, receberemos a mesma mensagem de erro. Isto porque estamos tratando o jquery e o modal como módulos. Cada módulos é uma unidade confinada de código, logo, jQuery não está no escopo global. Por estar confinado dentro de um módulo, ele não é acessível para Bootstrap.

Mas podemos permanecer tranquilos. Nós não queremos que o jQuery fique em uma closure, mas sim no escopo global - uma exigência do modal do `Bootstrap. Em uma situação na qual precisamos de uma biblioteca que esteja no escopo global, precisaremos fazer uma "manobra" para que isso seja possível no Webpack.

Faremos isso a seguir.

Como faremos para disponibilizar de uma maneira que seja acessível para todos os módulos da aplicação? Nosso objetivo é que ele fique em um escopo, que seja acessível por todos os módulos gerados no bundle.js.

A solução será utilizar um plugin que já está embutido no Webpack: [webpack.ProvidePlugin](https://webpack.js.org/plugins/provide-plugin). Para isto, iremos importar o objeto que representa o webpack.

const path = require('path');

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

const extractTextPlugin = require('extract-text-webpack-plugin');

const optimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');

const webpack = require('webpack');

A forma como disponibilizaremos o jQuery deve estar disponpivel tanto para ambiente de desenvolvimento como de produção, por isso, adicionaremos abaixo do extractTextPlugin, push() com webpack.ProvidePlugin.

let plugins = []

plugins.push(

new extractTextPlugin("styles.css")

);

plugins.push(

new webpack.ProvidePlugin({

'$': 'jquery/dist/jquery.js',

'jQuery': 'jquery/dist/jquery.js'

})

);

O que nós queremos disponibilizar para todos os módulos serem acessíveis, associamos ao $(cifão, alias do jQuery ); O jQuery também nos permitirá acessar o jQuery. O valor das duas variáveis disponibilizadas será o caminho do módulo 'jquery/dist/jquery.js', que será carregado pelo ProvidePlugin. As variáveis estarão no escopo acessível por cada módulo da aplicação, por isso, devemos utilizar um nome compreensível pelos plugins do jQuery (como $ e jQuery).

Após essas alterações, removeremos a importação do jquery no client/app-src/app.js.

import { NegociacaoController } from './controllers/NegociacaoController.js';

import { Negociacao } from './domain/index.js';

import 'booststrap/dist/css/bootstrap.css';

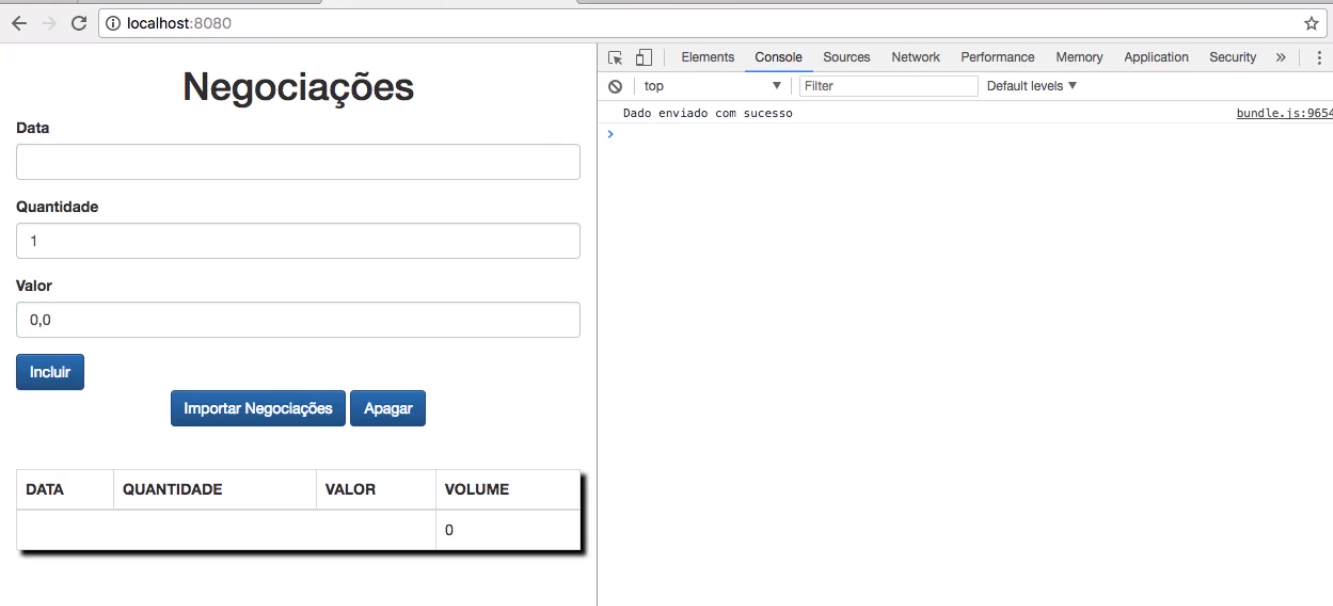
import 'bootstrap/dist/css/bootstrap-theme.css';

import 'bootstrap/js/modal.js';

import 'css/meucss.css';

Sempre que for feita uma alteração no webpack.config.js, devemos parar e rodar novamente o servidor.

Se atualizarmos a aplicação no navegador, a mensagem de erro não será mais exibida no Console.



**Atenção:** Se digitarmos $ no Console, o retorno será o seguinte

> $

f $(selector, [startNode]) { [Comand Line API] }

O Chrome possui uma assistente de linha de comando que recebe o nome de **$**. Apesar de que o jQuery ser acessível pelos módulos da aplicação, existe uma closure envolvendo a jQuery que não permite que escape para o escopo global.

Para sabermos se ele está disponível, realizaremos um teste no client/app-src/app.js. Adicionaremos o jQuery, com um alert():

import 'bootstrap/dist/css/bootstrap.css';

import 'bootstrap/dist/css/bootstrap-theme.css';

import 'bootstrap/js/modal.js';

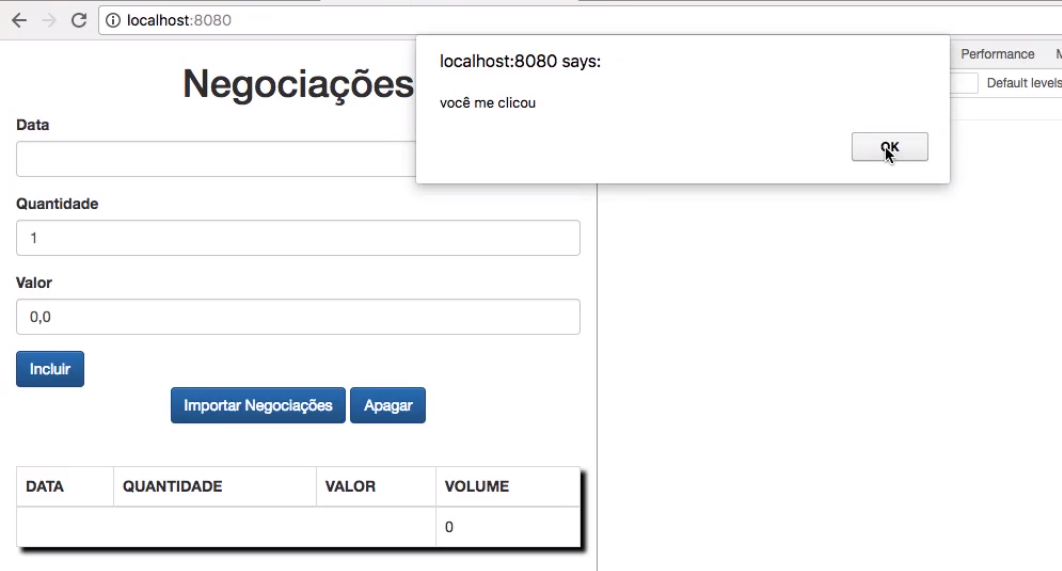
import '../css/meucss.css';

import { NegociacaoController } from './controllers/NegociacaoController.js';

import { Negociacao } from './domain/index.js';

$('h1').on('click',() => alert('você me clicou'));

Agora conseguiremos confirmar se o jQuery está realmente funcionando. Ao recarregarmos a página novamente no navegador, veremos a mensagem do jQuery será exibida.



O modalfoi disponibilizado do Bootstrap, faremos um console.log() com uma pequena "gambiarra", adicionando jQuery com h1 e todo elemento do jQuery - se o modal do Bootstrap está carregado -, vai contar a função modal.

import { NegociacaoController } from './controllers/NegociacaoController.js';

import { Negociacao } from './domain/index.js';

import 'bootstrap/dist/css/bootstrap.css';

import 'bootstrap/dist/css/bootstrap-theme.css';

import 'bootstrap/js/modal.js';

import '../css/meucss.css';

$('h1').click(() => alert('você me clicou'));

console.log($('h1').modal);

No Console, foi exibido o código da função Modal, no qual veremos que o plugin foi colocado. Conseguimos carregar o jQuery, além do plugin do Bootstrap, ou seja, tudo está funcionando como esperávamos.

Vamos aplicar uma série de boas práticas. O Webpack, no passado, colocava cada módulo em uma closure e realizava um processo bem trabalhoso. Porém, é possível diminuir a quantidade de closures durante a criação dos módulos, isso acelera o tipo de processamento e carregamento no navegador.

Vale ressaltar que falamos do tempo no qual os módulos são parseados, executados e processados. Como podemos fazer isso? Nosso escopo é pequeno e por isso, a diferença será quase imperceptível. Mas é uma boa prática adotar essa abordagem na sua aplicação. É o que faremos em modo de produção.

Dentro do if de production, adicionaremos optimize.ModuleConcatenationPlugin()antes do babiliPlugin.

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

plugins.push(new webpack.optimize.ModuleConcatenationPlugin());

plugins.push(new babiliPlugin());

plugins.push(new optimizeCSSAssetsPlugin({

cssProcessor: require('cssnano'),

cssProcessorOptions: {

discardComments: {

removeAll: true

}

},

canPrint: true

}));

}

Com isso, as alterações serão feitas no módulo, fazendo com que seu processamento ocorra mais rapidamente. Esta é uma novidade introduzida a partir da versão Webpack 3 e que estamos usando na nossa aplicação.

Em seguida, no Terminal, usaremos o comando npm run build-prod. O build de produção será mais demorado dessa vez, no entanto, o bundle.js será gerado já com a otimização.

Cada módulo do nosso bundle é envolvido por um **wrapper**, que resumidamente se trata de uma função. Contudo, a existência desses wrappers tornam a execução do script um pouco mais lenta no navegador .

Entretanto, a partir do Webpack 3, podemos ativar o **Scope Hoisting**. Ele consiste em concatenar o escopo de todos os módulos em um único wrapper, permitindo assim que nosso código seja executado mais rapidamente no navegador.

Vamos ativar esse recurso apenas no build de produção, adicionando uma instância de ModuleConcatenationPlugin em nossa lista de plugins:

// client/webpack.config.js

// código anterior omitido

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

// ATIVANDO O SCOPE HOISTING

plugins.push(new webpack.optimize.ModuleConcatenationPlugin());

plugins.push(new babiliPlugin());

plugins.push(new optimizeCSSAssetsPlugin({

cssProcessor: require('cssnano'),

cssProcessorOptions: {

discardComments: {

removeAll: true

}

},

canPrint: true

}));

}

// código posterior omitido

Isso já é suficiente. Contudo, devido à dimensão reduzida da nossa aplicação, teremos dificuldade de ver na prática as mudanças no tempo de execução da nossa aplicação.

Se analisarmos o bundle.js, veremos que tem o tamanho de 106 kb. Dentro dele, encontraremos misturados os arquivos criados por mim com as bibliotecas do sistema.

À medida que a aplicação for avançando, teremos novas bibliotecas e mais código, o arquivo ficará imenso até ser necessário dividi-lo. Por isso, é um boa prática manter o código escrito por nós no bundle.js, e as demais bibliotecas podem ficar em outro arquivo. O Webpack nos permite fazer isso de forma apropriada e elegante.

Precisamos adicionar CommonsChunkPlugin no nosso código, localizado logo abaixo dos plugins incluídos anteriormente.

let plugins = [];

plugins.push(new extractTextPlugin('styles.css'))

plugins.push(new webpack.ProvidePlugin({

'$': 'jquery/dist/jquery.js',

'jQuery': 'jquery/dist/jquery.js'

}));

plugins.push(

new webpack.optimize.CommonsChunkPlugin({

}));

O CommonsChunkPlugin vai nos ajudar a fatiar a nossa aplicação, com ele, será criado um módulo que receberá o nome de vendor. Nele, todas as bibliotecas criadas por terceiros e o bundle.js será um arquivo enxuto.

Mas precisaremos adicionar um name e filename.

plugins.push(

new webpack.optimize.CommonsChunkPlugin(

{

name: 'vendor',

filename: 'vendor.bundle.js'

}

)

);

O CommonsChunkPlugin vai ter um plugin identificador para o bundle que se chama vendor - que será usado mais adiante. Além disso, todos os arquivos indicados para as bibliotecas de terceiros, ficarão dentro vendor.bundle.js. O próximo passo será configurar quem integrarão vendor.bundle.js, no entry do module.exports que atualmente está assim:

module.exports = {

entry: {

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

vendor: ['jquery', 'bootstrap', 'reflect-metadata']

},

Observe começaremos com jquery, por ser um dependência do boostrap.

Agora o ponto de entrada será app, e teremos que quebrá-lo. Passaremos também vendor, considerando que também usamos o vendorno CommonsChunkPlugin. No caso, devem ser carregados o boostrap e jquery, reflect-metada - como este foi adicionado, podemos remover o <script> com reflect. Faremos primeiro isso no index.html, ficando apenas com um script acima do fechamento da tag <body>.

<div id="negociacoes"></div>

<script src="dist/bundle.js"></script>

</body>

</html>

Quando fizermos o build do projeto, o Webpackverá que temos dois pontos de entrada. No caso do app, será aproveitado todo o conteúdo do app.js, resolverá as dependências e vai gerar o bundle.js. No vendor, será considerado todos os módulos indicados e a gravação será feita no vendor.bundle.js.

Você pode questionar o fato de que estamos importando os arquivos modal e bootstrapno app.js. Porém, eles não integraram o bundle.js, onde estarão os arquivos que nós programamos. O Webpack saberá que eles fazem parte do módulo do Bootstrap - por termos indicado isso no vendor.

module.exports = {

entry: {

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

vendor: ['jquery', 'bootstrap', 'reflect-metadata']

},

A seguir, vamos importar o vendor.bundle.js dentro do index.html.

<div id="negociacoes"></div>

<script src="dist/vendor.bundle.js"></script>

<script src="dist/bundle.js"></script>

Com esse tipo de abordagem, se tivermos mais um bundle, teremos que importá-lo manualmente nesse arquivo. Faremos essa melhoria mais adiante.

No Terminal, vamos rodar npm run build-dev, depois, no projeto, veremos bundle.js e o vendor.bundle.js dentro da pasta dist. Se minificarmos o arquivo, ele ficará com um tamanho menor.

Se testarmos a aplicação no navegador, veremos que está tudo funcionando. Porém, desta vez, se analisarmos o código-fonte, veremos que estão sendo carregados dois arquivos bundle.js. É uma boa prática fazer essa separação.

Falta algo ainda no projeto antes de finalizá-lo. Quando criamos o arquivo vendor.bundle.js, nós tivemos que importá-lo no index.html. Outra alteração feita foi que adicionamos a tag <link> para que style.css fosse importado também. Porém, queremos que os artefatos gerados pelo Webpack sejam importados automaticamente. É possível fazer isso com o auxilio do plugin [html-webpack-plugin](https://github.com/jantimon/html-webpack-plugin), que instalaremos pelo Terminal:

npm install html-webpack-plugin@2.29.0 --save-dev

Recomento que você use a versão html-webpack-plugin@2.29.0, porque sei que ela está funcionando. Depois, veremos o novo plugin listado no package.json.

"devDependencies": {

"babel-core": "^6.25.0",

"babel-loader": "^7.1.0",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-preset-es2017": "^6.24.1",

"babili-webpack-plugin": "^0.1.1",

"cross-env": "^5.0.1",

"css-loader": "^0.28.4",

"cssnano": "^3.10.0",

"extract-text-webpack-plugin": "^3.0.0",

"file-loader": "^0.11.2",

"html-webpack-plugin": "^2.29.0",

"optmize-css-assets-webpack-plugin": "^2.0.0",

"style-loader": "^0.18.2",

"uninstall": "0.0.0",

"url-loader": "^0.5.9",

"webpack": "^3.1.0"

"webpack-dev-server": "^2.5.1"

}

Em seguida, acessaremos index.html para apagar as tags <script>, com import dos dois bundle.

//...

<div id="negociacoes"></div>

</body>

</html>

O arquivo index.html será usado como template gerar outro index.html, por isso, é recomendável alterar o nome do arquivo do primeiro para main.html. Quando o Webpack fizer o build do projeto, ele criará o arquivo index.html baseado no main.html, vai inserir todos os artefetos gerados por Webpack e vai gravar dentro da pasta dist.

Lembrando que atualmente o index.html ainda não vai ser gravado na pasta dist, mas com essa mudança, o public path configurado no webpack.config.js não poderá mais conter a a pasta dist. Porque se estamos dentro desta pasta e importamos os bundles.js da mesma pasta, teremos um problema. Não existe a pasta dist dentro de dist. A solução será remover o public path, porque os arquivos serão importado na raiz de dist.

O próximo passo será importar HtmlWebpackPlugin no webpack.config@js. Iremos ainda adicionar plugin logo abaixo, passando dentro as configurações:

const path = require('path');

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

const extractTextPlugin = require('extract-text-webpack-plugin');

const optimizeCSSAssetsPlugin = require('optmize-css-assets-webpack-plugin');

const webpack = require('webpack');

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

let plugins = [];

plugins.push(new HtmlWebpackPlugin({

hash: true

}))

Quando ele adicionar no index.html que será criado baseado no main.html, o CSS e o script do bundle, incluirá hash no fim para ajudar a invalidar o cash. É desnecessário termos o nome do arquivo no hash. Podemos colocar como parâmetro outro hash, que invalidará o cash. Por enquanto, basta entender que nosso objetivo é gerar o cash.

Vamos pedir que ele minifique, ao incluirmos o minify. Por enquanto, passaremos algumas instruções informando que o HTML5 é true, desta forma ele poderá realizar ações como a compactação de arquivos. Teremos também o collapseWhitespace, que iremos configurar como true também, ele será responsável pela remoção de espaço entre as tags. Adicionaremos ainda removeComments para que sejam removidos os comentários.

plugins.push(new HtmlWebpackPlugin({

hash: true,

minify: {

html5: true,

collapseWhitespace: true,

removeComments: true,

},

filename: 'index.html',

template: \_\_dirname + '/main.html'

}));

Observe que especificamos o filename e o template que ele utilizará como referência. Se testarmos no Terminal, após o build, veremos que o index.html foi gerado dentro de dist, que teve o código compactado. Além disso, se analisarmos o arquivo, veremos que style.css foi importado, juntamente com o hash. Com isso, se o CSS for alterado, o hash mudará, invalidando o cash do navegador.

No fim do arquivo, vemos que ele importou o bundle.js e o vendor.bundle.js. Com o comando npm start subiremos o webpack-dev-server e, no navegador, veremos que a aplicação está funcionando corretamente.

Com esta nova organização, todo os arquivos na pasta dist são referentes ao que colocaremos em produção. Mais adiante, veremos outras boas práticas que adotaremos.

A aplicação que estamos utilizando para trabalhar com Webpack é o protótipo de um single page application. Nós sabemos que é normal utilizar frameworks do mercado em aplicações deste tipo, como Angular e React.

No entanto, existe algo que devemos considerar. Você percebeu que todos os módulos da aplicação ficarão no bundle.js, ou seja, em um único arquivo? Isto significa que se a aplicação tiver muitos módulos, o bundle.js vai ficar imenso.

No entanto, isto não é bom. Nós evitamos o problema de fazer múltiplas requisições para fazermos o download de vários arquivos. Mas, agora, gastamos um bom tempo para baixar o bundle.js e para só então, ele será processado pela aplicação.

Um estratégia utilizada para a resolução de problemas como esse é o **code splitting**(separação de código) e o **lazy loading** (carregamento preguiçoso). Em aplicação como AngularJS ou VueJS, em seus sistemas de rotas, podemos informar que queremos carregar um módulo no momento em que for necessário.

A aplicação não será carregada com esse módulo, ele ficará em back-end separado, e quando acessarmos o módulo pela primeira vez, ele será carregado pelo framework uma única vez.

Desta forma, se queremos fazer um carregamento sob demanda (lazy loading), devemos ter a aplicação divida em partes menores. Para o escopo do projeto, vamos escolher esse módulo - que define a classe NegociacaoService.js - para ser a parte carregada sob demanda.

Com isso, o Webpack saberá que no momento da criação dos bundles, será gerado um arquivo bundle para a aplicação e outro separado destinado ao módulo carregado sob demanda.

Estes conceitos serão utilizados se você trabalha com AngularJS e React, ainda que existam alguns detalhes específicos de quando estes fazem o lazy loading. Mas é isso que faremos basicamente.

Primeiramente, levaremos em conta que não deve haver na nossa aplicação um importestático do módulo NegociacaoService. Sabemos que nesta aplicação, temos o index.js dentro de domain. Neste arquivo, estamos exportando o NegociacoesService.js:

export \* from './negociacao/Negociacao.js';

export \* from './negociacao/NegociacaoDao.js';

export \* from './negociacao/NegociacaoService.js';

export \* from './negociacao/Negociacoes.js';

Esta é uma técnica que chamamos de **barrel**. Nós removeremos o export referente ao NegociacaoService, depois, faremos o mesmo em NegociacaoController.js. No caso, dentro do primeiro import:

import { Negociacoes, Negociacao } from '../domain';

import { NegociacoesView, MensagemView, Mensagem, DateConverter } from '../ui';

import { getNegociacaoDao, Bind, getExceptionMessage, debounce, controller, bindEvent } from '../util';

Vamos removê-lo porque apagaremos todas as importações estáticas do módulo realizadas em nossa aplicação. Como retiramos o import, removeremos também o atributo service responsável pela criação da instância de NegociacaoService(). Logo abaixo do this.\_mensagem, removeremos this.\_service.

this.\_service = new NegociacaoService();

Em seguida, no método importaNegociacoes(), acessamos o service apesar de não existir mais a propriedade da classe, logo, teremos que criar a variável. A seguir, nós usaremos o **destructuring** do ECMAScript 6 para importar o NegociacaoService, que receberá o resultado de await System.import(). E dentro dos parênteses, passaremos o caminho do módulo.

@bindEvent('click', '#botao-importa')

@debounce()

async importaNegociacoes() {

try {

const { NegociacaoService } = await System.import('../domain/negociacao/NegociacaoService');

const service = new NegociacaoService();

const negociacoes = await service.obtemNegociacoesDoPeriodo();

console.log(negociacoes);

negociacoes.filter(novaNegociacao =>

!this.\_negociacoes.paraArray().some(negociacaoExistente =>

novaNegociacao.equals(negociacaoExistente)))

.forEach(negociacao => this.\_negociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações do período importadas com sucesso';

} catch (err) {

this.\_mensagem.texto = getExceptionMessage(err);

}

}

Adicionamos ainda ao try, const service.

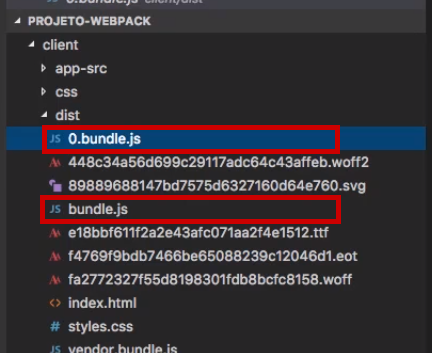
Como o projeto utilizou o ES8, utilizamos o async await, com isso podemos chamar um método que seja assíncrono, e esperar o seu resultado.

Se você quer se aprofundar nesta sintaxe, pode fazer o [curso de Typescript](https://cursos.alura.com.br/course/typescript-parte1/task/26739) da Alura.

Agora é possível realizar o carregamento "preguiçoso", com o System.import() - que é disponibilizado pelo Webpack. Você pode se perguntar sobre a maneira como estamos fazendo lazy loading do módulo NegociacaoService. Ele será carregado pela primeira vez, em seguida, será utilizado pela variável, depois, quando chamarmos o mesmo método, o módulo não será mais recarregado e o System.import não será feito novamente.

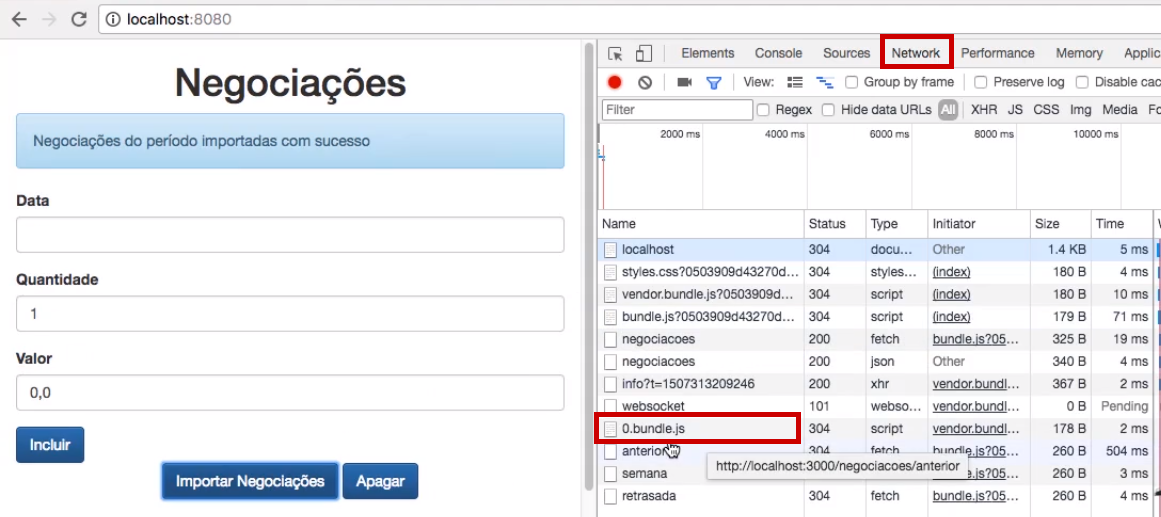
Porém, como será feita a separação por bundles diferentes?

Se fizermos o build de desenvolvimento no Terminal, o Webpack criará dois novos arquivos dentro da pasta dist: bundle.js e 0.bundle.js.



No arquivo 0.bundle.js, encontraremos o módulo que será carregado primeiramente, por meio do lazy loading. Ao perceber que está sendo realizada a importação dinâmica do módulo, o Webpack sabe quem deverá ser isolado no novo bundle, gerando o código perfeito.

Feito isso, não precisamos nos preocupar em importar o arquino no index.html. Após rodarmos o comando npm start no Terminal, veremos que a aplicação está rodando normalmente. Se abrirmos o Inspect e acessarmos a aba "Network", na primeira vez que a página for carregada, o arquivo 0.bundle.js não será listado. Em seguida, quando clicarmos no botão "Importar Negociações", o arquivo já será baixado na aplicação.



Esta forma de fazer carregamento preguiçoso e com códigos separados é bastante adotada em single pages applications. O sucesso do funcionamento dos frameworks depende do bom uso do Webpack.

O System.import() é uma maneira utilizada pelo Webpack para fazer o carregamento assíncrono sob demanda de módulos. Porém, o ECMAScript nas versões mais atuais, usa uma sintaxe específica para carregamento nestes casos, fazendo o uso da função import().

Tanto o Chrome como outros navegadores já suportam esta função, o Webpack ao detectá-la fará por debaixo dos panos, ele fará o System.import(), mas aceitando a sintaxe do ES que se tornará vigente.

Há uma proposta para introduzir importações dinâmicas na linguagem que está bem avançada e que pode ser acompanhada em https://github.com/tc39/proposal-dynamic-import.

O Webpack 3 está preparado para trabalhar com o import(), mas se salvamos e damos o build no projeto, teremos um erro na compilação. Somos informados que ele tem um erro na sintaxe, no entanto, ele não é entendido pelo Babel - utilizado por frameworks como React. Isto significa que o Babel entende como inválido o usuo do import().

Você tem opção de alterar o código e usar o System.import(), ou pode utilizar o que a sintaxe mais moderna, ue será adotada como padrão no futuro. No segundo caso, se você utilizar o Babel, terá que instalar o plugin babel-plugin-syntax-dynamic-import.

No Terminal, rodaremos o seguinte comando:

npm install babel-plugin-syntax-dynamic-import@6.18.0 --save-dev

Com isso, não será realizado o parse do Babel e deixar de reclamar sobre a sintaxe. Depois de fazemos o download do plugin, acessaremos o arquivo .babelrc para adicionar o novo plugin.

{

"presets":["es2017"],

"plugins" : ["transform-decorators-legacy", "babel-plugin-syntax-dynamic-import"]

}

Agora ele será processado também durante o processo de compilação. Compilaremos a aplicação novamente, e ao recarregarmos a aplicação, a aplicação conseguirá importar as negociações como o esperado.



Nós utilizamos a abordagem mais moderna para a realização do carregamento assíncrono (lazy loading). Este ajuste era necessário apenas em projetos que utilizam o Babel.

Tem algo que pode ter deixado dúvidas para você. Vamos rodar novamente no Terminal o comando npm run build-prod, depois, será realizado o build de produção. Se estivéssemos trabalhando com Angular, em uma Single Page Application, quais arquivos colocaremos em produção? No caso, o conteúdo completo da pasta dist será colocado em produção. Cada comand line interface de framework SPA, criará um pasta, mas vale ressaltar que vai ser oferecido uma comando para que seja gerada a pasta d distribuição.

Por exemplo, se colocarmos todos esses arquivos dentro da página do GitHub Pages, vai funcionar.

O que queríamos esclarecer que dist é a pasta de distribuição, caso você trabalhe com um framework da sua preferência, basta verificar qual pasta será gerada no fim.

Se analisarmos o domain/NegociacaoService.js, veremos que acessamos http://localhost:3000. Isso faz sentido se estamos em ambiente de desenvolvimento, porém, quando fizermos o build de produção, vamos querer adicionar o endereço da API em produção - que seria diferente do atual.

Veremos como é possível trocar valores da URL, adequando-os ao tipo de build. Novamente, utilizaremos o plugin webpack.DefinePlugin do webpack.config.js e já vem cokm Webpack.

Começaremos adicionando a variável SERVICE\_URL que vai receber um endereço padrão http://localhost:3000. Trabalharemos com essa URL no build de desenvolvimento. Se estamos no build de produção, entraremos no if e teremos outro endereço - no caso, usaremos um inexistente.

let SERVICE\_URL = JSON.stringify('http://localhost:3000');

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

SERVICE\_URL = JSON.stringify('http://endereco-da-sua-api');

plugins.push(new webpack.optimize.ModuleConcatenationPlugin());

plugins.push(new babiliPlugin());

plugins.push(new optimizeCSSAssetsPlugin({

cssProcessor: require('cssnano'),

cssProcessorOptions: {

discardComments: {

removeAll: true

}

},

canPrint: true

}));

}

O endereço http://endereco-da-sua-api não existe e, por isso, ocorrerá um erro quando estivermos em ambiente de produção, mas o objetivo é demonstrar que ele foi modificado. Nós devemos entender que o valor que será utilizado pelo webpack.DefinePlugin deve ser o resultado de JSON.stringify(), caso contrário, teremos um erro de parse.

Se estamos fazemos de build de desenvolvimento trabalharemos com um valor, se for ambiente de produção, será outro.

Depois do if, já teremos um valor definido, adicionaremos plugins.push().

plugins.push(new webpack.DefinePlugin({

SERVICE\_URL: SERVICE\_URL

}));

O DefinePlugin()recebe como parâmetro o objeto JS e os valores que varreremos os nosso módulos e trocar. Sempre que for encontrado SERVICE\_URL no módulos deverá ser trocado pelo valor SERVICE\_URL. Vimos anteriormente que no ES, quando o nome da propriedade tem o mesmo nome da variável, podemos deixar apenas uma referência. Podemos deixar uma versão mais enxuta:

plugins.push(new webpack.DefinePlugin({ SERVICE\_URL }));

Fizemos as alterações necessárias no webpack.config.js. O próximo passo será modificar em NegociacaoService.js o endereço do get adotando template string, substituindo as as aspas simples ("'") por crase ("`"), porque queremos interpolar as bvariáveis.

obtemNegociacoesDaSemana() {

return this.\_http

.get(`${SERVICE\_URL}/negociacoes/semana`)

.the(

dados => dados.map(objeto =>

new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)),

err => {

throw new ApplicationException('Não foi possível obteras negociações')

}

);

Faremos o mesmo com em obtemNegociacoesDaSemanaAnterior(), obtemNegociacoesDaSemanaRetrasada().

return this.\_http

.get(`${SERVICE\_URL}/negociacoes/anterior`)

// código omitido

}

Observe que adicionamos ${SERVICE\_URL}. Em seguida, transformaremos em template string o fetch() em app.js.

fetch(`${SERVICE\_URL}/negociacoes`, config)

.then(() => console.log('Dado enviado com sucesso'));

Fizemos ajustes nos endereços, que serão diferentes no contextos de build de produção e de desenvolvimento. No Terminal, vamos testar se está tudo funcionando corretamente. Após fazermos o build de produção, dentro da pasta dist, encontraremos 0.bundle.js onde encontraremos o endereco-da-sua-api configurado.

Na vez de testarmos o build de desenvolvimento, veremos que o localhost:3000 foi utilizado. Se subirmos novamente o servidor e recarregarmos a página, conseguiremos importar as negociações normalmnete.

Vimos como utilizar em nossas aplicações o Webpack, inclusive, nos casos em que você adotar AngularJS, React, com VueJS.

Nossa aplicação é um projeto embrionário de uma Single Page Application, aquele tipo de aplicação que não recarrega a página durante seu uso e que depende única e exclusivamente de JavaScript para funcionar. Não temos um sistema de roteamento e outros recursos de uma aplicação mais completa criada por frameworks como Angular, React, Ember entre outros. Mas isso não nos impede de colocá-la no ar.

Vale lembrar que nossa aplicação é constituída de duas partes. A primeira, a aplicação cliente, aquela que roda no navegador, a segunda, a API do backend. Focaremos apenas o processo de deploy (colocar no ar) da aplicação cliente, foco desta obra. Todavia, nada impede que o leitor implemente a API utilizada pelo curso utilizando outras linguagens como PHP, Java, C#, etc. realizando o deploy de sua API utilizando o serviço de hospedagem que mais se familiarizar. Aliás, é muito comum aplicações Single Page Application realizarem essa divisão marcante entre o deploy do cliente e da API.

Chegamos ao fim de mais um curso da Alura. Espero que você tenha gostado do formato do treinamento, no qual não deixamos o Webpack "preso" a nenhum framework. Desta forma, conseguimos fixar os conceitos fundamentais. Assim, é possível utilizar o Webpack no seu projeto trabalhando apenas com JAvaScript, ou aproveitar para se aperfeiçoar em frameworks como AngularJS, React, VueJS e outros.

Obrigado, alunos e alunos. Vamos continuar para o próximo curso...

Deixo meus votos de sucesso para todos!

Durante o vídeo, renomeamos o arquivo client/index.html para client/main.html. Nosso objetivo era gerar o arquivo index.html dentro de client/dist com todos os outros arquivos gerados pelo Webpack.

Além de renomear o arquivo, foi preciso remover todas as tags <link> e <script> que importam arquivos CSS e JavaScript respectivamente.

Tendo a certeza sobre as alterações feitas anteriormente, modificamos o arquivo client/webpack.config.js. O primeiro passo foi importar e adicionar o plugin como primeiro da nossa lista:

// client/webpack.config.js

const path = require('path');

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

const extractTextPlugin = require('extract-text-webpack-plugin');

const optimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');

const webpack = require('webpack');

// importou o novo plugin

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

let plugins = []

// adicionou como primeiro da lista

plugins.push(new HtmlWebpackPlugin({

hash: true,

minify: {

html5: true,

collapseWhitespace: true,

removeComments: true,

},

filename: 'index.html',

template: \_\_dirname + '/main.html',

}));

// código posterior omitido

O plugin recebeu um objeto como parâmetro com as seguintes configurações:

**hash**: quando true, adiciona um hash no final da URL dos arquivos script e CSS importados no arquivo HTML gerado, importante para versionamento e cache no navegador. Quando um bundle diferente for gerado, o hash será diferente e isso é suficiente para invalidar o cache do navegador, fazendo-o carregar o arquivo mais novo.

**minify**: recebe um objeto como parâmetro com as configurações utilizadas para minificar o HTML. Podemos consultar todas as configurações possíveis no endereço <https://github.com/kangax/html-minifier#options-quick-reference>.

**filename**: o nome do arquivo HTML que será gerado. Respeitará o valor do path de output que já configuramos logo no início da criação do arquivo webpack.config.js.

**template**: caminho do arquivo que servirá como template para geração de index.html.

Por fim, fizemos outra alteração. Como o arquivo index.html gerado ficou em client/dist/index.html, ajustamos o caminho dos imports dos scripts e dos estilos. Para isso, removemos a propriedade publicPath da chave output:

// client/webpack.config.js

// código anterior omitido

module.exports = {

entry: {

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

vendor: ['jquery', 'bootstrap', 'reflect-metadata']

},

output: {

filename: 'bundle.js',

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

/\* removeu publicPath \*/

},

// código posterior omitido

Pronto. Ao realizarmos um teste com a criação de um build de desenvolvimento, por exemplo, com o comando npm:

npm run build-dev

Analisando client/dist, vemos que o arquivo index.html foi gerado:

├── 448c34a56d699c29117adc64c43affeb.woff2

├── 89889688147bd7575d6327160d64e760.svg

├── bundle.js

├── e18bbf611f2a2e43afc071aa2f4e1512.ttf

├── f4769f9bdb7466be65088239c12046d1.eot

├── fa2772327f55d8198301fdb8bcfc8158.woff

├── index.html

├── styles.css

└── vendor.bundle.js

Nossa solução funciona para o build de produção e também quando estamos usando o Webpack Dev Server.