03

Executando o Babel

Transcrição

Realizaremos um teste manual para entender o que estamos fazendo. Com o Terminal ainda aberto no pasta `aluraframe/client` execute o seguinte comando:

```
./node_modules/babel-cli/bin/babel.js js/app-es6 -d js/app
```

Em versões mais novas:

```
./node_modules/.bin/babel js/app-es6 -d js/app
```

(testado com babel-cli 6.26.0 e babel-preset-es2015 6.24.1)

Estamos executando o `babel.js` , passando como parâmetro a pasta de origem que passará pelo processo de transcompilação: `js/app-es6` . Nós usamos o parâmetro `-d` indicando qual será o diretório de destino, no caso `js/app` . E se compararmos o código em ES6 com o gerado pelo ES5 ?

Atualmente o código do `aluraframe/client/js/app-es6/views/MensagemView.js` está assim:

```
class View {

    constructor(elemento) {

        this._elemento = elemento;
    }

    template() {

        throw new Error('O método template deve ser implementado');
    }

    update(model) {
        this._elemento.innerHTML = this.template(model);
    }
}
```

A versão em ES 5 gerada pelo Babel (`aluraframe/client/js/app/views/MensagemView.js`) ficou assim:

```
'use strict';

var _createClass = function () { function defineProperties(target, props) { for (var i = 0; i <

function _classCallCheck(instance, Constructor) { if (!(instance instanceof Constructor)) { thr

var View = function () {
    function View(elemento) {
```

```
            _classCallCheck(this, View);

            this._elemento = elemento;
        }

        _createClass(View, [{
            key: 'template',
            value: function template() {

                throw new Error('O método template deve ser implementado');
            }
        }, {
            key: 'update',
            value: function update(model) {
                this._elemento.innerHTML = this.template(model);
            }
        }]);

        return View;
    }();
```

Não se preocupe com a legibilidade do código... O mais importante é entender que ele gerou um código compatível e funcional em ES5. Podemos testá-lo, abrindo outro terminal e subindo o servidor a partir da pasta `aluraframe/server` .

```
npm start
```

Agora, acesse a URL da aplicação `localhost:3000` .

Tudo continua funcionando perfeitamente. Mas é um tanto trabalhoso rodarmos o comando gigante manualmente sempre que quisermos compilar o nosso código. Será que temos uma opção mais simples? Veremos a seguir.

Vamos adicionar um script em `aluraframe/client/package.json` , que será um atalho para o comando recém executado.

```
{
  "name": "client",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
      "build": "babel js/app-es6 -d js/app "
    },
    "author": "",
    "license": "ISC"
    "devDependencies": {
      "babel-cli": "^6.10.1",
      "babel-preset-es2015": "^6.9.0"
  }
}
```

Veja que adicionei dentro de `script` s a chave `build` . Eu poderia ter usado outro nome para a chave, mas o relevante é o seu valor. Observe que usamos como valor "babel js/app-es6 -d js/app", praticamente igual ao que executamos no

terminal. A diferença é que não usamos o caminho completo.

Agora, para executar nosso script no Terminal executaremos o seguinte comando:

```
npm run build
```

Bem mais simples...

Podemos pedir para que Babel gere um `sourcemap`, ao compilar nossos arquivos. Trata-se de um arquivo que liga o arquivo resultante da compilação com o seu original para efeito de depuração, ou seja, para uso do *debugger*.

Vamos alterar `aluraframe/client/package.json` e adicionar o parâmetro `--source-maps` na chamada dos `script`s:

```
{
  "name": "client",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
      "build": "babel js/app-es6 -d js/app --source-maps"
    },
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "babel-cli": "^6.10.1",
      "babel-preset-es2015": "^6.9.0"
  }
}
```

Agora, o processo será exibido no Terminal. Em seguida, executaremos novamente:

```
npm run build
```

Dentro de `aluraframe/client/js/app`, foi criado o arquivo `.map` para cada arquivo resultante. Os navegadores carregam automaticamente os arquivos `.map`.

> **Atenção**: o Visual Studio Code e outros editores de texto podem ter dificuldade em exibir em seu sidebar os arquivos gerados em `aluraframe/client/js/app`, isto ocorre porque eles são criados fora de seu ambiente. Não é preciso se preocupar com o fato, considerando que jamais poderemos alterar esses arquivos. Se quisermos alterá-los, devemos alterar os arquivos originais e, então, o Babel irá gerar novamente um código transcompilado. Se você quiser verificar se a pasta `app` foi criada, você pode fechar e abrir o editor novamente ou verificar pelo Terminal.

Depois, faremos um teste. Vamos alterar o método `importaNegociacoes()`, do arquivo `NegociacaoController.js` da pasta `app-es6`. A alteração será feita na seguinte linha:

```
this._service = new NegociacaoService();
```

Após a mudança, ela ficará assim:

```
this._service = new xNegociacaoService();
```

Observe que o nome da classe está errado. Após, rodamos novamente o comando "npm run build" para o Babel transcompilar o arquivo.

Vamos recarregar o navegador e novo arquivo será utilizado. Veremos que as negociações não foram importadas. Por isso, verificaremos no Console a mensagem de erro:

```
Uncaught ReferenceError: xNegociacaoService is not defined
```

Ele também nos indicará em qual linha está o problema, no nosso caso, a linha `22` do arquivo `NegociacaoController.js`. Vamos acessá-la e seremos direcionados para a linha do erro no arquivo original. Mas o arquivo carregado pelo navegador foi o transcompilado. Desta forma, resolvemos o assunto sobre transcompilação.