08

Trabalhando com módulos do ES2015!

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/js-avancado/stages/modulo3/07-aluraframe.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

ATENÇÃO: o projeto não possui a pasta `aluraframe/client/node_modules` e você precisará baixar as dependências abrindo o terminal na pasta `aluraframe/client` para em seguida executar o comando **npm install**. Este comando lerá seu arquivo `package.json` e baixará todas dependências listadas nele.

Escopo global e carregamento de scripts = dor de cabeça

Nossa aplicação vem se tornando cada vez mais sofisticada: persistimos dados localmente com IndexedDB, aplicamos padrões de projeto e utilizamos transpiler entre outras coisas. Contudo, desde o primeiro módulo do curso não atacamos dois pontos fracos da linguagem JavaScript: o escopo global e o carregamento de scripts. Primeiro, vamos filosofar sobre o escopo global.

Todas as nossas classes, inclusive a instância de `NegociacaoController` estão no escopo global. Agora, vamos simular o seguinte: baixamos uma lib de terceiros para usar em nossa aplicação. Vamos criar o arquivo `aluraframe/client/js/app/lib/datex.js` com o seguinte código:

```
// aluraframe/client/js/app/lib/datex.js

class DateHelper {

    dateToString(date) {
        /* faz algo */
    }

    stringToDate(string) {
        /* faz algo */
    }
}
```

Vamos importá-lo em `aluraframe/client/index.html` como último script. Quando ele for carregado,por possuir o mesmo nome da nossa classe `DateHelper` que esta no escopo global, haverá colisão de nomes e a nova classe substituirá a já existente. Como a classe agora não possui os método `textoParaData()` e nem `dataParaTexto()` nossa aplicação não funcionará. Faça um teste.

Outro problema, além de colisões no escopo global é a ordem de carregamento de scripts. Você deve lembrar que no primeiro módulo, éramos obrigados a carregar `View.js` antes de `ListaNegociacoesView` e `MensagemView` porque elas herdam de `View` e esta classe obrigatoriamente deve vir primeiro. Veja que esse problema acaba jogando nas costas do desenvolvedor a responsabilidade de verificar em que posição um novo script deve ser incluído.

A plataforma Node.js resolveu este problema adotando padrão CommonJS para criação de módulos, ainda há bibliotecas como `RequireJS` que usam o padrão AMD (*Assincronous Module Definition*). Contudo, o ES2015 especificou seu próprio sistema de módulos que resolve tanto o problema do escopo global quanto o de carregamento de scripts.

ES2015 e módulos

Apesar de fazer parte da especificação, ainda não há consenso a respeito de como os scripts devem ser carregados pelo navegador. É por isso que para usarmos o sistema de módulos oficial do JavaScript precisamos utilizar loaders de terceiros, que nada mais são do que scripts especiais que farão o carregamento dos nossos módulos. Neste treinamento, utilizaremos o `SystemJs`, um carregador de módulos universal que suporta módulos do ES2015.

Além do loader, ajustes em nosso código devem ser feitos para adequá-lo ao loader utilizado. Resumindo: para que possamos utilizar os módulos do ES2015, precisamos utilizar um loader e transcompilar nosso código.

Antes de baixarmos nosso loader, vamos primeiro configuração Babel para que adeque nosso código ao `Systemjs`.

Babel e transcompilação de módulos

Hoje, temos apenas configurado o preset `es2015` no arquivo `.babelrc`. Ele garante a compilação do nosso código para ES5. Contudo, este preset não esta preparado para transcompilar módulos para o `Systemjs`.

No terminal e dentro da pasta `aluraframe/client` vamos instalar o plugin `transform-es2015-modules-systemjs`:

```
npm install babel-plugin-transform-es2015-modules-systemjs@6.9.0 --save-dev
```

Com o módulo baixado, vamos alterar nosso arquivo `aluraframe/client/.babelrc` e adicioná-lo como plugin. Nosso arquivo ficará assim:

```
{
  "presets": ["es2015"],
   "plugins": ["transform-es2015-modules-systemjs"]
}
```

A ideia de um plugin é a seguinte, depois de um preset ser aplicado, plugins são aplicados em seguida realizamos demais transformações.

Refatorando nosso código com import e export

Agora, precisamos alterar todos os scripts que criamos até o momento para fazerem uso da sintaxe de módulo do ES2015. Precisamos escolher um para começar. Vamos começar por `aluraframe/client/js/app-es6/views/MensagemView.js`.

A classe `MensagemView` depende da classe `View`, tanto isso é verdade que usamos a sintaxe `extends` para herdá-la. Do jeito que esta, não funcionará com o sistema de módulos do ES2015, pois cada script é um módulo que confina o código declarado nele, evitando assim que caia no escopo global.

Precisamos explicitar que queremos usar a classe `View` por meio da instrução `import`.

```
import {View} from './View';

class MensagemView extends View {
```

```
    // código omitido
}
```

Veja que usamos `import` seguido de `{View}`. Colocamos o nome da classe que desejamos importar de um módulo entre chaves. Em seguida, usamos a instrução `from` apontando para o local do módulo. Encare cada script nosso agora como um módulo, ou seja, `View.js` é um módulo. Contudo, do jeito que esta, não funcionará. Porque tudo que estiver entre `{}` deve ser exportado pelo módulo. Se abrirmos `View.js` em nenhum momento deixamos claro que a classe `View` pode ser importada. Corrigimos isso facilmente adicionando a instrução `export` antes da definição da classe:

```
export class View {

    // código omitido
}
```

Veja que o módulo `View.js` agora exporta uma classe: a `View`. Mas atenção, sabemos que `MensagemView` será usada por `NegociacaoController`, sendo assim, precisamos também usar `export` para exportar a classe:

```
import {View} from './View';

export class MensagemView extends View {

    // código omitido
}
```

Agora, vamos fazer a mesma coisa com `NegociacoesView`. Primeiro, vamos verificar suas dependências. Ela depende de `View` e também de `DateHelper`. Então, primeiro, vamos alterar `DateHelper.js` para que exporte a classe `DateHelper`:

```
export class DateHelper {

  // código omitido
}
```

Agora sim, podemos alterar `NegociacoesView.js`:

```
import {DateHelper} from '../helpers/DateHelper';
import {View} from './View';

// exportamos, porque é usada em NegociacaoController.js

export class NegociacoesView extends View {

    // código omitido
}
```

Bom, os módulos da pasta `aluraframe/client/js/app/views` já foram todos alterados. Agora, vamos para a pasta `aluraframe/client/js/app/services`.

Vamos começar pelo módulo `HttpService.js`. Ele não depende de classes de outros módulos, sendo assim, não precisamos usar a instrução `import`. Mas o módulo precisa exportar a classe `HttpService` porque ela é usada por

`NegociacaoService` .

```
export class HttpService {
    /* código omitido */
}
```

Agora, vamos analisar o módulo `NegociacaoService.js` . Ele depende de `HttpService` , de `Negociacao` , pois instancia negociação, de `ConnectionFactory` e `NegociacaoDao` . Primeiro, vamos fazer com que o módulo `Negociacao.js` exporte a classe `Negociacao` .

```
export class Negociacao {
}
```

Agora, vamos atacar o módulo `ConnectionFactory` . Você deve lembrar que implementamos o module pattern para esconder algumas variáveis do programador e exportar apenas a classe. Como estamos usando o sistema de módulos do ES6 podemos simplificar bastante nosso código. Hoje ela está assim:

```
var ConnectionFactory = (function() {

    let stores = ['negociacoes'];
    let version = 9;
    let dbName = 'aluraframe';
    let connection = null;
    let close = null;

    return  class ConnectionFactory {


        // código omitido
    }
})();
```

Como módulos do ES6 já escondem do mundo externos variáveis e classes, podemos simplesmente remover a `IIFE` que utilizamos e exportar apenas a classe como já fizemos com outras classes:

```
let stores = ['negociacoes'];
let version = 9;
let dbName = 'aluraframe';
let connection = null;
let close = null;

export  class ConnectionFactory {

    // código omitido
}
```

Agora, precisamos fazer com que o módulo `NegociacaoDao.js` importe `Negociacao` e exporte a classe `NegociacaoDao` :

```
import {Negociacao} from '../models/Negociacao';
```

```
export class NegociacaoDao {

    // código omitido
}
```

Então, agora podemos alterar `NegociacaoService`:

```
import {HttpService} from './HttpService';
import {Negociacao} from '../models/Negociacao';
import {ConnectionFactory} from './ConnectionFactory';
import {NegociacaoDao} from '../dao/NegociacaoDao';

export class NegociacaoService {
// código omitido
}
```

Vamos alterar `ProxyFactory`. Ela não depende de outra classe, só deve exportar:

```
export class ProxyFactory {
    // código omitido
}


export class Mensagem {
    // código omitido
}


export class ListaNegociacoes {
    // código omitido
}
```

Dentro de `aluraframe/client/js/app/helpers` faltou modificar o módulo `Bind`. Ele depende de `ProxyFactory`:

```
import {ProxyFactory} from '../services/ProxyFactory';

export class Bind {
    // código omitido
}
```

Por fim, falta o módulo `aluraframe/client/js/app/controllers/NegociacaoController.js`. Esta sim, dependerá de vários módulos:

```
import {Mensagem} from '../models/Mensagem';
import {Negociacao} from '../models/Negociacao';
import {ListaNegociacoes} from '../models/ListaNegociacoes';
import {NegociacoesView} from '../views/NegociacoesView';
import {MensagemView} from '../views/MensagemView';
import {NegociacaoService} from '../services/NegociacaoService';
import {DateHelper} from '../helpers/DateHelper';
```

```
import {Bind} from '../helpers/Bind';

export class NegociacaoController {

    // código omitido
}
```

Bom, agora, precisamos remover a instância de `negociacaoController` de `index.html`, pois ela não ficará mais no escopo global.

Veja que com isso, não podemos mais associar o evento clique do botão `adiciona` com o `apaga` ou com a controller, porque não há outra `negociacaoController` no escopo global. A página `index.html` ficará assim:

```
<!DOCTYPE html>
<html>
<head>
   <!-- código omitido -->
</head>
<body class="container">

    <!-- código omitido -->

    <form class="form"">

        <!-- código omitido -->

        <button class="btn btn-primary" type="submit">Incluir</button>
    </form>

    <div class="text-center">
        <button class="btn btn-primary text-center" type="button">
            Apagar
        </button>
    </div>
    <br>
    <br>

    <div id="negociacoesView"></div>
</body>
</html>
```

Mas como associaremos a submissão do formulário à chamada do método `adiciona()` de `NegociacaoController`?

O primeiro passo é indicarmos qual será o primeiro módulo a ser carregado pela aplicação. Criaremos o módulo `aluframe/client/js/app-es6/boot.js`. Ele importará `NegociacaoController`:

```
import {NegociacaoController} from './controllers/NegociacaoController';
let negociacaoController = new NegociacaoController();
```

A instância de `NegociacaoController` não estará no escopo global, sendo assim, precisaremos adicionar manualmente para o formulário e o botão que apaga as negociações a chamada dos métodos da nossa instância:

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

var negociacaoController = new NegociacaoController();

document.querySelector('.form').onsubmit = negociacaoController.adiciona;
document.querySelector('button[type=button]').onclick = negociacaoController.apaga;
```

Veja que para o evento `onsubmit` do formulário associamos o método `adiciona()`, contudo nosso código não funcionará. O `this` dos métodos deixaram de ser a instância `negociacaoController` e passarão a ser o elemento do DOM ao qual foram associados. Para resolvermos isso, podemos usar a conhecida função `bind()`:

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

var negociacaoController = new NegociacaoController();

document.querySelector('.form').onsubmit = negociacaoController.adiciona.bind(negociacaoControll
document.querySelector('button[type=button]').onclick = negociacaoController.apaga.bind(negociac
```

Perfeito. Agora que temos todos os módulos configurados, precisamos configurar nosso loader. Seu papel será carregar `boot.js`. Como `boot.js` depende `NegociacaoController.js`, o loader entende que deve baixar esse script e quando ver que este módulo depende de outros sairá baixando e resolverá todas as dependências. A vantagem disso é que não precisaremos importar mais scripts em nossa página! Apenas precisaremos importar o loader e indicar para ele que `boot.js` será o primeiro a ser carregado.

Vamos baixar nosso loader, o Systemjs usando o Terminal e dentro da pasta `aluraframe/client`:

```
npm install systemjs@0.19.31 --save
```

Com o script baixado, vamos importá-lo como único script em `index.html`:

```
    <!-- código anterior omitido -->
    <script src="node_modules/systemjs/dist/system.js"></script>
</body>
</html>
```

Agora, vamos indicar para o loader que `boot.js` será o primeiro módulo a ser carregado:

```
    <!-- código anterior omitido -->
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script>
        System.defaultJSExtensions = true; // permite omitir a extensão .js dos imports
        System.import('js/app/boot').catch(function(err){
            console.error(err);
        });
    </script>
</body>
</html>
```

Indicamos em `System.import` que `boot.js` será o primeiro módulo a ser carregado. Observe que não precisamos mais nos preocupar com a ordem de carregamentos de scripts, o loader vai resolver tudo para nós.

Vamos parar o babel e rodá-lo novamente para que compile nossos módulos. Em seguida, carregaremos nosso projeto que deve continuar funcionando. Se você tentar imprimir no console o nome de uma de nossas classes, inclusive a instância de `negociacaoController`, o resultado será `undefined`. Lembre-se que agora não trabalhamos mais com variáveis globais e não há mais o risco de colisão. Além disso, não precisamos mais nos preocupar com a ordem de carregamentos dos scripts.