03

Refatorando nosso código com import e export

Transcrição

Temos um caso atípico com a classe `ConnectionFactory`. Nela, criamos o Model Pattern para exportá-la no escopo global. Mas escondemos algumas variáveis:

```
var ConnectionFactory = (function () {

      const stores = ['negociacoes'];
      const version = 4;
      const dbName = 'aluraframe';

      var connection = null;

      var close = null;

      return class ConnectionFactory {

          constructor() {

              throw new Error('Não é possível criar instâncias de ConnectionFactory');
          }
//...
```

Como todo arquivo JS é um módulo, o conteúdo não será mais acessível. Apenas o que é exportável será acessível para outros módulos. Em vez do `return`, vamos usar o `export` para a classe:

```
const stores = ['negociacoes'];
const version = 4;
const dbName = 'aluraframe';

let connection = null;

let close = null;

export class ConnectionFactory {

    constructor() {

        throw new Error('Não é possível criar instâncias de ConnectionFactory');
    }
//...
```

Observe que também substituímos o `var` pelo `let`, porque as variáveis não vão mais cair no escopo global. Em seguida, faremos alterações no arquivo `NegociacaoDao`:

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

export class NegociacaoDao {
```

```
    constructor(connection) {

        this._connection = connection;
        this._store - 'negociacoes';
    }
//...
```

Adicionaremos o `export` e `import` em `Bind.js` :

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

export class Bind {

    constructor(model, view, ...props) {

        let proxy = ProxyFactory.create(model, props, model =>
            view.update(model));

            view.update(model);
    }
}
```

Voltaremos em `NegociacaoService` para fazer uma modificação que deixamos escapar. Adicionaremos mais duas importações:

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

export class NegociacaoService {

      constructor(){

          this._http = new HttpService()
      }
 //...
```

Depois, iremos inserir o `export` em `ListaNegociacoes.js` :

```
export class ListaNegociacoes {

    constructor() {

        this._negociacoes = [];
 }
//...
```

Faremos o mesmo em `Mensagem.js` :

```
export class Mensagem {

    constructor(texto) {

        this._texto = texto || '';
    }
//...
```

Temos que exportar também em `Negociacao.js` :

```
export class Negociacao {

    constructor(data, quantidade, valor) {

      this._data = new Date(data.getTime());
      this._quantidade = quantidade;
      this._valor = valor;
      Object.freeze(this);
    }
//...
```

As próximas alterações serão em `NegociacaoController` :

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

class NegociacaoController {

    constructor() {

       let $ = document.querySelector.bind(document);
//...
```

Observe que a Controller tem várias dependências.

Nós fizemos diversas modificações nos arquivos, no entanto, se executarmos o código, ele ainda não funcionará. Usamos a sintaxe do ES6 (de importação e exportação de módulos), mas entenderemos por que isso ainda não é o suficiente.