05

Melhorando a legibilidade e manutenção do nosso código

Transcrição

Será que podemos melhorar o nosso código ainda mais? Com certeza! Observe que em todas as partes em que é preciso interagir com o IndexedDB, tivemos que criar uma conexão e passa-la para o construtor do DAO. Então realizarmos uma série de operações com o resultado. Veja que repetimos isso em vários trechos de código:

```
ConnectionFactory
    .getConnection()
    .then(connection => new NegociacaoDao(connection))
```

O desenvolvedor precisa ficar atento a muitos detalhes para trabalhar com o código, por exemplo, vamos analisar o `NegociacaoController.js` , que adiciona uma negociação no banco local:

```
// js/app/controllers/NegociacaoController.js

// código anterior omitido

    adiciona(event) {

        event.preventDefault();

        ConnectionFactory
            .getConnection()
            .then(conexao => {

                let negociacao = this._criaNegociacao();

                new NegociacaoDao(conexao)
                    .adiciona(negociacao)
                    .then(() => {
                        this._listaNegociacoes.adiciona(negociacao);
                        this._mensagem.texto = 'Negociação adicionada com sucesso';
                        this._limpaFormulario();
                    });

        })
        .catch(erro => this._mensagem.texto = erro);

    }

//...
```

Queremos algo assim:

```
    adiciona(event) {

        event.preventDefault();
```

```
        let negociacao = this._criaNegociacao();

        new NegociacaoService.cadastra(negociacao)
            .then(mensagem => {
                this._listaNegociacoes.adiciona(negociacao);
                this._mensagem.texto = mensagem;
                this._limpaFormulario();
            }).catch(erro => this._mensagem.texto = erro);
    }
```

O exemplo de código anterior esconde a complexidade de se lidar com a conexão, inclusive com a criação do DAO. Preparado para melhorar nosso código?

Vamos editar `aluframe/client/js/app/services/NegociacaoService.js` e adicionar o método `cadastra()`, que receberá uma negociação:

```
class NegociacaoService {

    constructor() {

        this._http = new HttpService();
    }
    // código anterior omitido

    cadastra(negociacao) {
    }

}
```

Agora, vamos mover o código que está no método `adiciona()` de `NegociacaoController`, para dentro do método `cadastra()` de `NegociacaoService`, exceto a instrução `event.preventDefault()`:

```
class NegociacaoService {

    constructor() {
        this._http = new HttpService();
    }

    // código anterior omitido

    cadastra(negociacao) {

         ConnectionFactory
            .getConnection()
            .then(conexao => {

                // VEJA QUE ISSO NÃO FAZ MAIS SENTIDO!
                let negociacao = this._criaNegociacao();

                new NegociacaoDao(conexao)
                    .adiciona(negociacao)
                    .then(() => {
                        this._listaNegociacoes.adiciona(negociacao);
                        this._mensagem.texto = 'Negociação adicionada com sucesso';
```

```
                        this._limpaFormulario();
                    });
            })
            .catch(erro => this._mensagem.texto = erro);
    }
}
```

Observe que o método `cadastra` recebe como parâmetro a negociação que será incluída, é por isso que podemos remover a instrução que constrói uma negociação usando `this._criaNegociacao()` e as instruções que interagem com propriedades de `NegociacaoController`. Podemos até remover alguns blocos da nossa *promise*, para tornar o código mais enxuto:

```
class NegociacaoService {
    constructor() {
        this._http = new HttpService();
    }
    // código anterior omitido

    cadastra(negociacao) {
         return ConnectionFactory
            .getConnection()
            .then(conexao => new NegociacaoDao(conexao))
            .then(dao => dao.adiciona(negociacao))
            .then(() => 'Negociação cadastrada com sucesso')
            .catch(erro => {
                throw new Error("Não foi possível adicionar a negociação")
            });
    }
}
```

Vamos analisar e relembrar o que aprendemos nos cursos anteriores. Realizamos uma série de operações encadeadas a partir de `.getConnection()`. Como este método retorna uma *promise*, temos acesso ao seu resultado, a conexão, na próxima chamada de `then`. Não queremos trabalhar com uma conexão, mas com um DAO, por isso, fizemos:

```
.then(conexao => new NegociacaoDao(conexao))
```

Como não usamos bloco, o retorno é automático e será uma instância de `NegociacaoDao`, que teremos acesso na próxima chamada ao método `then`. Agora que temos o DAO, podemos adicionar uma negociação. Como o método `adiciona()` do nosso DAO também retorna uma *promise*, quando a operação for concluída, o código da próxima chamada do `then` será executado.

Como `dao.adiciona` não retorna nenhum valor, ele simplesmente insere no banco, passamos nossa próxima chamada do `then`, ficando `then(() =>)`. Veja que retornamos a string **"Negociação cadastrada com sucesso"**. Esse valor será disponibilizado para quem chamar método `cadastra()`, quando for executado com sucesso. Em `catch`, logamos o erro e lançamos uma exceção que fará o `catch()` de quem está usando o `cadastra` ser executado.

Agora, vamos alterar o método `adiciona` de `NegociacaoController` para que faça uso do novo método de `NegociacaoService`:

```
// aluraframe/client/js/app/controller/NegociacaoController.js

    adiciona(event) {

        event.preventDefault();

        let negociacao = this._criaNegociacao();

        new NegociacaoService()
            .cadastra(negociacao)
            .then(mensagem => {
                this._listaNegociacoes.adiciona(negociacao);
                this._mensagem.texto = mensagem;
                this._limpaFormulario();
            }).catch(erro => this._mensagem.texto = erro);
    }
```

Veja que agora o desenvolvedor não precisa conhecer os detalhes de criação de um DAO para adicionar uma nova negociação. Ele pode, simplesmente, chamar um método.