01

Organização do código através de namespaces

Transcrição

Começando deste ponto? Você pode fazer o [download (https://s3.amazonaws.com/caelum-online-public/typescript/06-alurabank.zip)](https://s3.amazonaws.com/caelum-online-public/typescript/06-alurabank.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

Hoje a definição das nossas classes vivem no escopo global. Além disso, para que o desenvolvedor saiba quais são as classes disponíveis pelo autocomplete do TypeScript ele precisa saber pelo menos parte do nome. Contudo, TypeScript oferece o conceito de namespace. Podemos agrupar classes dentro de um mesmo namepace e acessá-las através dele.

No caso, vamos envolver todas as classes dentro de `alurabank/app/views` no namespace `Views`:

```
namespace Views {

    export abstract class View<T> {

        protected _elemento: JQuery;

        constructor(seletor: string) {

            this._elemento = $(seletor);
        }

        update(model: T) {

            this._elemento.html(this.template(model));
        }

        abstract template(model: T): string;

    }

}
```

Veja que além do namespace, é necessário adicionarmos a instrução `export` para que a classe esteja disponível.

Agora, vamos alterar `MensagemView` e `NegociacaoView` adicionando-os no mesmo namespace. Agora, para estendermos da classe `View` precisaremos usar a sintaxe `Views.View`:

```
namespace Views {

    export class MensagemView extends Views.View<string> {

        template(model: string): string {

            return `<p class="alert alert-info">${model}</p>`;
        }
```

```
    }
}


namespace Views {

    export class NegociacoesView extends Views.View<Negociacoes> {

        template(model: Negociacoes): string {

            return `
            <table class="table table-hover table-bordered">
                <thead>
                    <tr>
                        <th>DATA</th>
                        <th>QUANTIDADE</th>
                        <th>VALOR</th>
                        <th>VOLUME</th>
                    </tr>
                </thead>

                <tbody>
                    ${model.paraArray().map(negociacao =>
                        `
                            <tr>
                                <td>${negociacao.data.getDate()}/${negociacao.data.getMonth() +1
                                <td>${negociacao.quantidade}</td>
                                <td>${negociacao.valor}</td>
                                <td>${negociacao.volume}</td>
                            <tr>
                        `
                    ).join('')}
                </tbody>

                <tfoot>
                </tfoot>
            </table>
            `;
        }
    }

}
```

Agora, em `NegociacaoController`:

```
class NegociacaoController {

    private _inputData: JQuery;
    private _inputQuantidade: JQuery;
    private _inputValor: JQuery;
    private _negociacoes = new Negociacoes();
    private _negociacoesView = new Views.NegociacoesView('#negociacoesView');
    private _mensagemView = new Views.MensagemView('#mensagemView');
```

```
// código posterior omitido
```

Excelente, mas essa solução não resolve o problema de termos que nos preocupar com a ordem de importação dos scripts em `index.html`, um dos tendões de Aquiles do mundo JavaScript. É por isso que o TypeScript aceita também o sistema de módulos do ES2015. Será ele que utilizaremos, em vez do sistema de namespace.