JavaScript Avançado I: ES6, orientação a objetos e padrões de projetos

- Esse curso capacita a:
  - Unir o paradigma orientado a objetos ao funcional para resolver problemas;
  - Aplicar novos recursos do ECMASCRIPT 6;
  - Estruturar a aplicação no modelo MVC;
  - Utilizar padrões de projeto;

Aulas:

1. **Prólogo: regras, código e manutenção**

2. **Especificando uma Negociação:**
   - Em uma classe para deixar os atributos privados: **this._atributo** e adicionar depois de todos os atributos no construtor: **Object.freeze(this)** [Vai congelar um objeto depois de criado]**;**
   - Para fazer um **get,** lembra como em Python, igual uma property:
     get <nomeAtributo>() {}

3. **A ligação entre as ações do usuário e o modelo:**
   - **let $ = document.querySelector.bind(document);** [Simula "sintaxe" jQuery em um .js sem a sua utilização]
   - A view apresenta um modelo em uma tabela, em um formulário ou em parágrafos, e o padrão MVC permite que qualquer alteração na view não interfira com o modelo;
   - O controller é aquele que recebe as ações do usuário e que sabe interagir com o modelo. Como o modelo é independente da view, esta precisa ser renderizada para que reflita as alterações no modelo. Em suma, o controller é a ponte de ligação entre a view e o modelo;
   - **Spread operator (...<array>)**: Antes do array passado como parâmetro, cada item dele será passado para cada parâmetro recebido pela função, inclusive isso vale para o constructor de uma classe;
     - Criamos o primeira Controller da negociação: NegociacaoController. Vimos como associar uma ação do usuário, como a submissão do formulário, e chamar o método controller. Para criar um negociação do DOM, tivemos que criar os elementos do DOM, equivalente ao input da quantidade, da data e do valor para capturar os valores, sendo possível depois, instanciar uma negociação. Comentamos também como não é recomendável fazer isto todas as vezes que chamarmos o método adiciona(). Por isso, colocamos como propriedade da classe NegociacaoController o elemento do DOM.
     - Para evitarmos escrever repetidas vezes document.querySelector, nós usamos o "truque" de colocá-lo na variável **$**. Mas vimos que

neste processo, o querySelector perdia o contexto do document, e o this deixava de apontar para este. Para resolver o assunto, usamos a chamada para o método bind() e o $ - equivalente ao querySelector - fizesse uma referência para o document. Fizemos um sintaxe parecida com jQuery.

- Falamos também que não era suficiente capturarmos a data do formulário e passá-la como parâmetro para o construtor de Date, porque o input vinha no formato ano, mês e dia. Depois, tivemos que fazer algumas transformações e vimos que o Date aceita trabalhar com alguns parâmetros. Passamos um array para o Date, também passamos uma string com ano, mês e dia, cada item separado por uma vírgula.

4. **Lidar com data é trabalhoso? Chame um ajudante!**
    - **Template String**: let frase = `${nome} ${sobrenome} também sabe JavaScript`;
        - Quando trabalhamos com data, precisamos transformar o formato da string para um objeto Data, com dia-mês-ano. É importante evitar a repetição do código sempre que for preciso usar a data no sistema, para isto, isolamos o trecho referente às conversões numa classe: o DateHelper. Em vez de usarmos concatenações nesta, optamos por usar uma template string que é criada com uma crase. Quando usamos no início e no fim, podemos colocar expressões dentro da template string, sem precisar das concatenações.
        - Como a Negociacao está pronta, começamos a preparar a listagem de negociações para a exibição. Porém, nesta listagem não podemos incluir, remover ou alterar uma Negociacao - uma das regras de negócio. Então, nós criamos uma classe do modelo que recebeu o nome ListaNegociacoes. Depois, conseguimos adicionar e obter essas negociações, por meio do método adiciona() e do getter negociacoes. Porém, vimos que essa lista de negociações dentro do model não estava blindada. Qualquer um que chamasse o getter conseguiria apagar ou incluir a lista, por isso, lançamos mão da programação defensiva.
        - Caso o getter de ListaNegociacoes fosse chamado, o retorno seria um array original e independente de qualquer interferência de fora. Fizemos isto utilizando um array vazio, seguido da função concat(). Criamos também alguns métodos auxiliares "privados" na Controller, além de brincarmos um pouco com as expressões regulares para validarmos o texto passado para conversão de data.

5. **Temos o modelo, mas e a view?**
    - A função **reduce** recebe dois parâmetros: uma função e um valor inicial. Na função interna ao reduce, o primeiro parâmetro é o valor da última iteração, e o segundo parâmetro é o valor da iteração atual;

```
let numeros = [1,2,3,4];

let resultado = numeros.reduce((anterior, atual) => anterior + atual, 0);
```

    - A função **map** recebe uma função como parâmetro, e nessa função, utiliza-se um parâmetro que é o elemento de cada índice do array que deverá ser retornado em cada iteração;
        - Implementamos um mecanismo de View dentro da aplicação. Ou seja, nós temos uma tabela na qual exibimos os dados da negociação. Mas em vez das marcações estarem no arquivo HTML, estas foram colocadas em uma classe nova chamada NegociacaoView.js.
        - Como o código da tabela ficou no JavaScript, a View precisou encontrar alguma forma de se renderizar e aparecer no HTML. Por isso, nós criamos uma que leva o id e indicou o ponto no qual o arquivo da tabela será inserido. Para realizar tal ação, a View recebeu um modelo - com qual "tampamos" as lacunas do template.
        - Nós ainda criamos o método _template(), utilizando a template string. Vimos que podemos gerar expressões mais "rebuscadas" para montar tags <tr> dinamicamente. Usamos novamente a função map() para transformar o conteúdo de um array. Além disso, utilizamos a função join() para poder concatenar todos os itens do array que equivalem às tags <tr> da tabela.

6. **Generalizando a solução da nossa View:**
    - Vimos que o código das Views NegociacoesView e MensagemView tinham trechos em comum. Nós isolamos tais partes dentro de uma classe, juntamente com o construtor que recebeu o elemento e o método update(). Depois, fizemos com que as duas Views herdassem da classe View, assim, não repetimos o código em comum. Mas coube às classes filhas implementarem o método template().
    - Criamos ainda uma "armadilha" para evitar a possibilidade de que o desenvolvedor se esquecesse de incluir o método, incluindo uma mensagem de erro no Console. Lembrando que um método da classe filha sobrescreve métodos da classe pai.
    - Fizemos também um pequeno ajuste, retirando o prefixo _ do método template(), que anteriormente era privado. A alteração foi necessária porque os métodos template() de NegociacoesView eMensagemView precisavam sobrescrever o método em View.