JavaScript Avançado II: 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;
    - Programar assincronamente com promises;
    - Utilizar padrões de projeto;
    - Implementar uma solução de data binding;

Aulas:

1. **Como saber quando o modelo mudou?**
    - **Reflect.apply:** O primeiro parâmetro é o método ou função que deseja-se invocar, o segundo parâmetro é o contexto que o método ou função adotará, ou seja, o valor que será assumido pelo this. Por fim, o último parâmetro é um array que contém todos os parâmetros que o método passado como primeiro parâmetro receberá;
    - **Padrão de projeto Observer:** sempre que queremos notificar partes do sistema interessadas quando um evento importante for disparado em nosso sistema;
        - Vimos como a chamada da atualização da View no código, quando o modelo for alterado. Seguindo o exemplo do listaNegociacoes, em que os métodos adiciona() e esvazia() eram chamados, será esse o momento no qual vamos disparar a atualização da View. Optamos por colocar "armadilhas", funções passadas para o construtor da classe que são chamadas sempre que os métodos adiciona() ou esvazia() forem usados. Estas funções recebem um código que acessa a View do Controller e executará o método update().
        - Mas fracassou, porque a function() era enviada para o construtor do modelo que é a armadilha. Quando isso acontecia, o this era dinâmico, ou seja, não pertencia ao Controller, e sim, ao model. Então, tentamos acessar no this a nossa View. Vimos como fazer isto por meio da API de reflexão do JavaScript, Reflection API, usando **reflect.apply()**. Ela recebe o nome do método, o contexto em que queremos executar o método, além dos parâmetros que serão passados para o método para corrigir o this no momento da execução da função. Mas vimos que este processo não era necessário. Em vez disso, usamos uma arrow function.
        - As arrow functions possuem um escopo léxico, enquanto as funções padrões têm um escopo dinâmico. Isto significa que, se temos uma função em JavaScript que varia de acordo com o contexto no qual é chamada, o this léxico de uma arrow function manterá o mesmo this em todas as chamadas da função.

2. **Existe modelo mentiroso? O padrão de projeto Proxy!**
    - O padrão de projeto Proxy nada mais é do que um objeto "falso", "mentiroso", que envolve e encapsula o objeto real que queremos interagir. É como se fosse uma interface, entre o objeto real e o resto do código. Conseguimos assim controlar o acesso aos seus atributos e métodos. Nele também podemos pendurar códigos que não cabem de estar alocados nos nossos modelos, mas que necessitam ser executados no caso de uma alteração ou atualização do mesmo;
    -
```
let pessoaProxy = new Proxy(pessoa, {
    get(target, prop, receiver) {
        //...
    }
```
Como segundo argumento de um proxy, passamos um handler, que é um objeto JavaScript que contém as armadilhas (traps) do nosso Proxy. Neste objeto, podemos criar uma propriedade get e passar para ela uma função com 3 parâmetros:
        - O **target** é o objeto real que é encapsulado pela proxy. É este objeto que não queremos "sujar" com armadilhas ou qualquer código que não diga respeito ao modelo;
        - O **prop** é a propriedade em si, que está sendo lida naquele momento;
        - O **receiver** é a referência ao próprio proxy. É na configuração do handler do Proxy que colocamos armadilhas;

      Dentro do get após as modificações necessário: **return Reflect.get(target, prop, receiver)**, pois ela que efetivamente realiza a operação no objeto real;
    - Para interceptar o método grita? A má notícia é que toda proxy criada, por padrão, não está preparada para interceptar métodos (getters e setters são exceções). Essa limitação ocorre porque sempre que um método de um objeto (que não deixa de ser uma propriedade que armazena uma função) é chamado, primeiro é realizado uma operação de leitura (get, do nosso handler da proxy) e depois os parâmetros são passados através de **Reflect.apply**. O problema é que, como o método é interceptado pelo get do handler passado para a proxy, não temos acesso aos seus parâmetros.
      Solução:

```
let pessoa = new Proxy(new Pessoa('Barney'), {

    get(target, prop, receiver) {
        if(prop == 'grita' && typeof(target[prop]) == typeof(Function)) {
        // essa função retornada irá substituir o método 'grita' no proxy!!!
        // Ou seja, estamos usando o handler do proxy para modificar o próprio proxy
            return function() {
                console.log(`Método chamado: ${prop}`);
                // Quando usarmos Reflect.apply, Reflect.get e Reflect.set precisamos
                // retornar o resultado da operação com return
                return Reflect.apply(target[prop], target, arguments);
                // "arguments" é uma variável implícita que dá acesso à todos os
                // parâmetros recebidos pelo método/função
            }
        }
        // só executa se não for função
        return Reflect.get(target, prop, receiver);
    }
})
pessoa.grita('Olá');
```

3. **E se alguém criasse nossas proxies? O Padrão de Projeto Factory:**
    - **REST**:
```
constructor(tipo, ...itens) {
    //lógica
}
```
utiliza-se antes do último parâmetro, e assim tudo que passarmos de extra será colocado dentro de um array;
    - **O Padrão de projeto Factory:**
        - Ele é utilizado quando precisamos facilitar a criação de um objeto;
        - É ideal quando queremos criar objetos similares, com apenas seus detalhes diferentes, que podemos passar nos argumentos da Factory;
        - É bom para abstrair a criação de um objeto complexo, já que o programador que utilizar a Factory não precisa necessariamente saber como é feita esta operação;
        - O padrão de projeto Factory ocorre quando temos uma classe que nos ajuda a criar um objeto complexo, ou seja, ela esconde de nós os detalhes de criação desse objeto. É por isso que uma classe Factory possui apenas um método. Faz sentido, porque se tivéssemos que chamar mais de um para criar um objeto a responsabilidade de sua criação cairia em nosso colo.

4. **Importando negociações**

5. **Combatendo Callback HELL com Promises:**
    -
```
let promise = new Promise((resolve, reject) => {
    // aqui executamos algo demorado

    if (/* tudo deu certo */) {
        resolve("Funcionou!");
    }
    else {
        reject("Deu erro ... ");
    }
})
.then(mensagem => console.log(mensagem))
.catch(erro => console.log(erro));
```
    - O construtor de Promise recebe uma função como parâmetro. É essa função passada como parâmetro que será chamada internamente pela Promise, quando for criada. Como é a própria Promise que chama essa função, ela passa sempre dois parâmetros para ela nesta ordem: a função na qual passamos o valor de sucesso(resolve) e a função que passamos o valor de fracasso(reject). Elas representam o resultado futuro de uma ação, que pode ser de sucesso ou fracasso. Elas visam tornar códigos assíncronos mais legíveis e fáceis de manter, evitando o Callback Hell;
    - Usamos os métodos **then** e **catch** para capturar o resolve e o reject oriundo da promisse e dentro deles é necessário passar uma function ou Arrow function com o comando desejado;

6. **Considerações finais e exercícios bônus**