04

Ah se meu código funcionasse no Firefox...

Desde o início do treinamento solicitei que vocês usassem o Google Chrome por ele suportar vários recursos do ES6. Contudo, o Firefox não fica atrás. Então, porque eu deixei esse navegador de fora? Simplesmente por eles não suportarem o input do tipo `Date`. Só por isso, Flávio? Sim!

Se eu tivesse começado o treinamento sem o auxílio do input do tipo `Date` muitos alunos focariam na validação do campo e perderiam o foco do treinamento. Contudo, com uma pequena alteração podemos fazer com que nosso código funcione no Firefox.

ATENÇÃO: o código foi testado no Firefox 45.0.2! Quer saber se versões mais antigas suportam os recursos que utilizamos? Acesse o módulo I deste curso, e veja a dica no exercício obrigatório do capítulo (https://cursos.alura.com.br/course/javascript-es6-orientacao-a-objetos-parte-1/task/16495).

Alterando a interface

O primeiro passo é deixarmos de usar o input do tipo date. Vamos usar um input padrão, do tipo texto e usar um placeholder para solicitar que o usuário digite seu código no formato `dd/mm/aaaa`:

```
<!-- aluraframe/client/index.html -->
<div class="form-group">
    <label for="data">Data</label>
    <!-- alterou para input text e ainda esta com um placeholder -->
    <input type="text" placeholder="dd/mm/aaaa" id="data" class="form-control" required autofoc
</div>
```

Alterando nosso helper de data

Precisamos alterar a classe `DateHelper`. Hoje ela espera receber um string no formato `aaaa-mm-dd`. Esse formato só era assim porque usávamos o input date, agora, vamos fazer com que o DateHelper funcione da seguinte maneira:

1 - validar uma data no formato dd/mm/yyyy

2 - extraia do formato dd/mm/yyyy um array com o ano, mes e dia.

Alterando a classe `DateHelper`:

```
// aluraframe/client/js/app/helpers/DateHelper.js

class DateHelper {

    // código anterior omitido

    static textoParaData(texto) {

        // mudamos a validação para aceitar o novo formato!
        if(!/\d{2}\/\d{2}\/\d{4}/.test(texto))
```

```
            throw new Error('Deve estar no formato dd/mm/aaaa');

        // veja que usamos no split '/' no lugar de '-'. Usamos `reverse` também para ficar ano,
        return new Date(...texto.split('/').reverse().map((item, indice) => item - indice % 2))
    }
}
```

Quando alteramos nossa expressão regular, trocamos `-` por `/`, contudo, como esse é um caractere especial, precisamos usar `\/`. O nosso processo de desmembrar a string continua o mesmo, mas como temos uma data no formato `dd/mm/aaaa`, precisamos realizar um split usando `/` como separador e aplicar um `.reverse()`! A inversão dos itens do array é importante, porque a função `map` espera encontrar um array com ano, mês e dia e não dia, mês e ano.

Realize um teste ainda no Chrome. Digite por enquanto apenas datas válidas. Assim que você verificar que esta funcionando, digite seu nome no campo que captura a data e clique em incluir. O que acontecerá? Nada! Abrindo o console vemos a mensagem:

```
DateHelper.js:16 Uncaught Error: Deve estar no formato dd/mm/aaaa
```

Existe try e catch em JavaScript?

O usuário nem fica sabendo que sua data é inválida! Que tal exibirmos essa mensagem para o usuário para que ele saiba o que esta acontecendo? Lembre-se que no método `DateHelper.textoParaData` lançamos um erro com a instrução `throw`. Essa instrução indica que houve um erro e que o método onde ele ocorreu não vai tratá-lo, mas sim lançá-lo para quem chamou o método. Sendo assim, quem chamou `DateHelper.textoParaData`? O método `_criaNegociacao` de `NegociacaoController`.

Quando temos uma área que pode resultar em um erro, envolvemos essa área com a instrução `try`. É como se aquela área fosse um campo minado e que em algum momento pode ocorrer um erro e precisamos estar preparados para lidar com ele. No bloco `try` quando ocorre um lançamento com `throw`, podemos capturar o erro lançado no bloco `catch`. Vejamos:

```
class NegociacaoController {

    // código anterior omitido

    adiciona(event) {

        event.preventDefault();

        try {
            this._listaNegociacoes.adiciona(this._criaNegociacao());
            this._mensagem.texto = 'Negociação adicionada com sucesso';
            this._limpaFormulario();
        } catch(erro) {
            this._mensagem.texto = erro;
        }
    }
```

Veja que a chamada de `this._criaNegociacao()` está dentro do bloco `try`, mas não ela apenas, mas o código que exibe a mensagem de sucesso e o de limpeza do formulário. A ideia é a seguinte: quando um erro é lançado, como

`this_criaNegociacao()` não o trata, o erro sobe na pilha. Daí, o interpretador JavaScript perguntará se quem chamou `this._criaNegociacao` está preparado para tratá-lo. E sim, está!

Se um erro acontecer, nosso código será direcionado para a cláusula `catch`, e nela temos acesso ao erro lançado pela instrução `throws` lá do nosso `DateHelper`. Resumindo: se um erro acontecer, não exibiremos a mensagem de sucesso e não limparemos o formulário e exibiremos a mensagem de erro para o usuário. Se não houver erro, as três instruções bloco `try` serão executadas, ou seja, a negociação será adicionada, a mensagem de sucesso exibida e o formulário limpo.

Teste mais uma vez no Chrome. Funcionando? Agora abra o Firefox e verifique que tudo funciona.