01

Escopo global e carregamento de scripts = dor de cabeça

Transcrição

Começando deste ponto? Você pode fazer o DOWNLOAD (https://github.com/alura-cursos/javascript-avancado-iii/archive/aula6.zip) completo do projeto até aqui e continuar seus estudos.

Atenção: o projeto não possui a pasta `aluraframe/client/node_modules` e você precisará baixar as dependências abrindo o Terminal na pasta `aluraframe/client` para em seguida executar o comando **npm install**. Este comando lerá seu arquivo `package.json` e baixará todas dependências listadas nele.

Fizemos coisas fantásticas usando o JavaScript, mas não atacamos uma das grandes fraquezas da linguagem: o escopo global e o carregamento de script. Nos aprofundaremos sobre tais assuntos.

Dentro da pasta `app-es6`, criaremos uma outra pasta que receberá o nome `lib`. Nesta, criaremos um arquivo chamado `datex.js`, na qual terá uma biblioteca baixada da internet com alguns métodos de data que o nosso `DateHelper` não tem: `dateToString()` e `stringToDate()`.

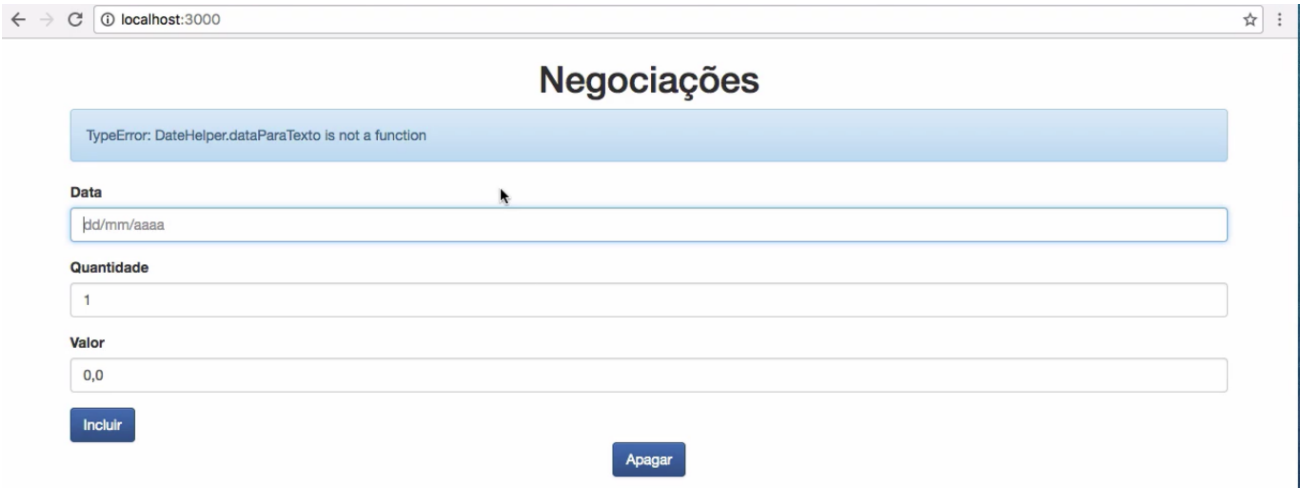
```
class DateHelper {

    dateToString(date) {
        /* faz algo */
    }

    stringToDate(string) {
        /* faz algo */
    }
}
```

Temos os métodos `dateToString()` e `stringToDate()`. Podemos adicionar ainda outras que nos auxiliaram a trabalhar com datas. Em seguida, importaremos este script como o último da lista `index.html`.

```
<div id="negociacoesView"></div>
<script src="js/app/polyfill/fetch.js"></script>
<script src="js/app/models/Negociacao.js"></script>
<script src="js/app/models/ListaNegociacoes.js"></script>
<script src="js/app/models/Mensagem.js"></script>
<script src="js/app/controllers/NegociacaoController.js"></script>
<script src="js/app/helpers/DateHelper.js"></script>
<script src="js/app/views/View.js"></script>
<script src="js/app/views/NegociacoesView.js"></script>
<script src="js/app/views/MensagemView.js"></script>
<script src="js/app/services/ProxyFactory.js"></script>
<script src="js/app/helpers/Bind.js"></script>
<script src="js/app/services/NegociacaoService.js"></script>
<script src="js/app/services/HttpService.js"></script>
<script src="js/app/services/ConnectionFactory.js"></script>
<script src="js/app/dao/NegociacaoDao.js"></script>
<script src="js/app/lib/datex.js"></script>
```

```
<script>
    var negociacaoController = new NegociacaoController();
</script>
```

O Babel está rodando e depois, compilará o arquivo, jogando-o dentro de `app`. Mas quando a página carregar no navegador, teremos uma mensagem de erro.



Por que tivemos um erro e a aplicação parou de funcionar? Na `lib` que baixamos, temos uma classe chamada `DateHelper` - e nós já tínhamos um arquivo com mesmo nome em `helpers`:

```
Class DateHelper {

    constructor() {

        throw new Error('Esta classe não pode ser instanciada');
    }

//...
```

Como esta definição de classe está no escopo global, quando o script `datex` é carregado, o `DateHelper` deste irá sobrescrever o anterior. Isto é das grandes limitações do JS - o mesmo ocorre com variáveis e funções que estão no escopo global. É impossível usar as duas classes `DateHelper` s simultaneamente. Outro problema é que quando importamos os scripts da nossa página, eles são colocados em determinada ordem. Lembre-se que o carregamento de View deve vir antes de `NegociacaoView` e `MensagemView`, porque `NegociacaoView` estende `View`. Então, se ocasionalmente carregarmos os scripts numa ordem contrária, teremos um erro de execução e a aplicação irá parar de funcionar. O desenvolvedor precisará ficar atendo sobre a ordem de dependência.

> A plataforma Node.js resolveu este problema adotando padrão CommonJS para criação de módulos, ainda há bibliotecas como `RequireJS` que usam o padrão AMD (Assincronous Module Definition). Contudo, o ES2015 especificou seu próprio sistema de módulos que resolve tanto o problema do escopo global quanto o de carregamento de scripts.

Nesta aula, nós usaremos o sistema de módulos do ECMAScript 2015, que nos ajudará a resolver os dois problemas.