04

O padrão de projeto Module Pattern

Transcrição

Uma solução para garantirmos que teremos a mesma conexão sempre que o método `getConnection()` for chamado é declarar a variável `connection` fora da classe `ConnectionFactory`:

```
var stores = ['negociacoes'];
var version = 4;
var dbName = 'aluraframe';

var connection = null;
```

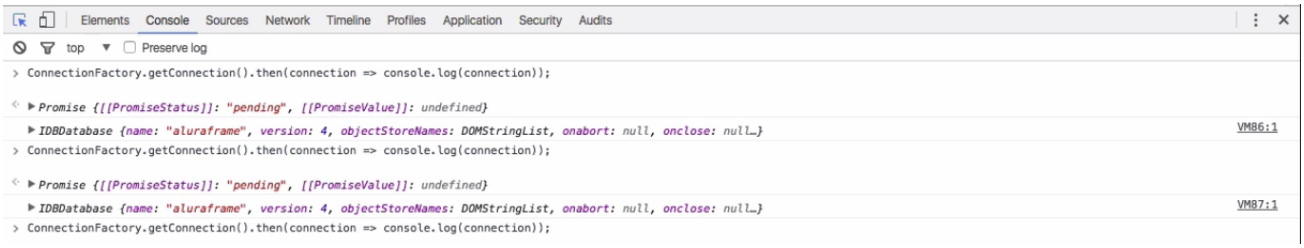
Depois, a variável será incluída no `onsuccess`:

```
openRequest.onsuccess = e => {

                if(!connection) connection = e.target.result;

                // recebe conexão já existente ou uma que acabou de ser criada

                resolve(connection);
 };
```

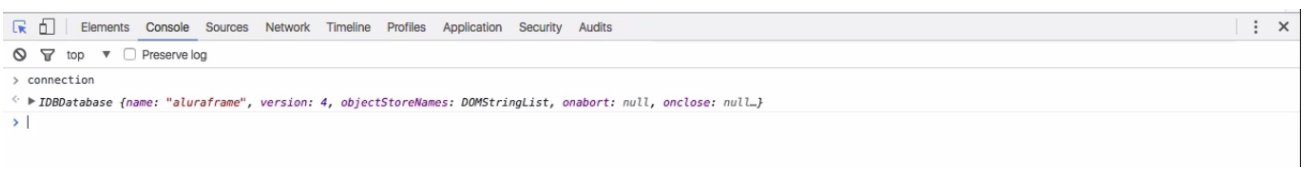
Se chamarmos o método `getConnection()` pela primeira vez, o valor de `connection` será nulo. Mas se chamarmos novamente, o `if` não será executado e o retorno será a mesma conexão. De volta a página no navegador, no Console, invocaremos o `getConnection()`:

```
ConnectionFactory.getConnection().then(connection => console.log(connection));
```

Se repetirmos a ação diversas vezes e o retorno será sempre o mesmo:



Resolvemos a questão. Mas ainda temos o problema de que temos quatro variáveis no escopo global. Se quisermos acessar pelo Console a variável `connection`, teremos que fazer isso:



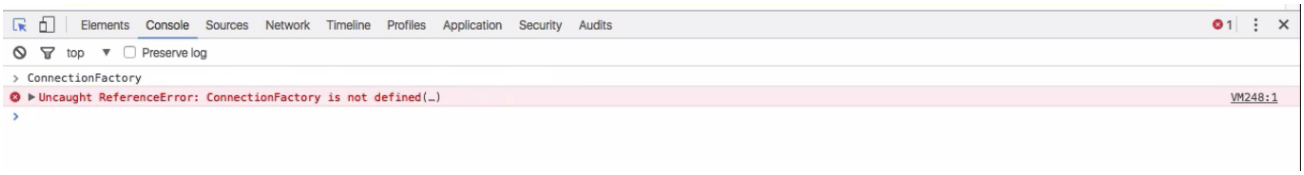
No entanto, não queríamos que isto ocorresse - e também podemos acessar as outras variáveis. Resolveremos o problema, aplicando um padrão de projeto JavaScript chamado *Module Pattern*. Um módulo é uma unidade código confinada e que ninguém tem acesso ao conteúdo dentro dele. Uma maneira de criarmos um escopo privado no JavaScript é colocando o código em uma função. Criaremos a `function tmp()`, e moveremos para dentro as variáveis juntamente com a definição da classe `ConnectionFactory`.

```
function tmp() {

    var stores = ['negociacoes'];
    var version = 4;
    var dbName = 'aluraframe';

    var connection = null;

    class ConnectionFactory {

        constructor() {

            throw new Error('Não é possível criar instâncias de ConnectionFactory');
        }

//...
```

Logo depois do `createStores()`, chamaremos a `tmp()`:

```
//...
  static _createStores(connection) {

      stores.forEach(store => {

          if(connection.objectStoreNames.contains(store)) connection.deleteObjectStore(store);
          connection.createObjectStore(store, { autoIncrement: true });
      });

  }
}
tmp();
```

Agora, conseguimos que ninguém tenha acesso ao conteúdo de `tmp()`, mas isto inclui o `ConnectionFactory`:



Se todo o código está dentro de `tmp()`, vamos adicionar um `return` na declaração da classe:

```
function tmp() {

    var stores = ['negociacoes'];
    var version = 4;
    var dbName = 'aluraframe';
```

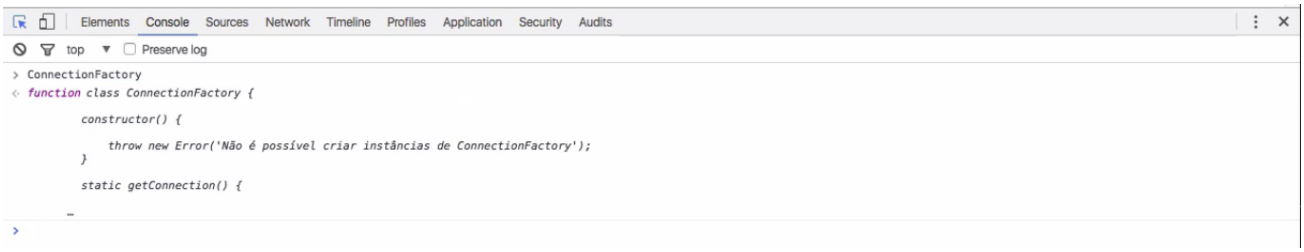
```
    var connection = null;

    return class ConnectionFactory {

        constructor() {

            throw new Error('Não é possível criar instâncias de ConnectionFactory');
        }

//...
```

Após darmos o `return` , colocaremos o `tmp()` do fim, dentro da variável `ConnectionFactory` .

```
var ConnectionFactory = tmp();
```

Usamos o mesmo nome da classe para podermos acessá-la. Vamos invocar o `ConnectionFactory` no Console:

```
> ConnectionFactory
```



Podemos utilizar a `ConnectionFactory` para obter uma conexão. Porém, o nosso código não ficou bom, primeiro declaramos a função e, depois, iremos chamá-la e dar o retorno. Para resolver a questão, não deixaremos mais que a função `tmp()` seja chamada, retirando o nome do código - e tornando a função **anônima**.

```
function () {

    var stores = ['negociacoes'];
    var version = 4;
    var dbName = 'aluraframe';

    var connection = null;

    return class ConnectionFactory {

        constructor() {

            throw new Error('Não é possível criar instâncias de ConnectionFactory');
        }

//...
```

Mas não podemos declarar funções anônimas desta forma - ainda que assim, a função `tmp()` não poderá ser chamada. Para resolvermos isto, envolveremos nossa função anônima por um parênteses. E para invocá-la, adicionaremos `()` no fim:

```
(function () {

    let stores = ['negociacoes'];
    let version = 4;
    let dbName = 'aluraframe';
    let connection = null;

    return class ConnectionFactory {


        constructor() {

            throw new Error('Não é possível criar instâncias de ConnectinFactory');
        }

        static getConnection() {

       }

        static _createStores() {

        }
    }
}) ();
```

Nós estamos criando uma função autoinvocada. Simultaneamente, ela será carregada e executada. Estamos enganando o compilador do JavaScript, porque o conteúdo do parênteses é uma função anônima... Em seguida, criaremos a variável `ConnectionFactory` no escopo global, mas o restante do código não estará.

```
var ConnectionFactory = (function () {

    var stores = ['negociacoes'];
    var version = 4;
    var dbName = 'aluraframe';

    var connection = null;

    return class ConnectionFactory {

        constructor() {

              throw new Error('Não é possível criar instâncias de ConnectionFactory');
        }

//...
```

Agora, se fizermos um teste e chamarmos a `ConnectionFactory` no Console, veremos que todo o escopo da classe será mostrada.

Aplicamos o *Module Pattern*, com o qual transformamos todo o `script` em um módulo - o código está todo confinado. E depois, definimos qual parte queremos exportar para o mundo externo usando o `return`. A `ConnectionFactory` é acessada, mas todo o restante não. Com isto, resolvemos o problema de utilizarmos uma única conexão.