05

Monkey Patch: grandes poderes trazem grandes responsabilidades

Transcrição

O design da `ConnectionFactory` está quase pronto. Eu não posso permitir que o desenvolvedor feche a conexão, como especificamos em uma das regras:

> D) Toda conexão possui o método close, mas o programador não pode chamá-lo, porque a conexão é a mesma para a aplicação inteira.

Se a conexão é a mesma na aplicação inteira, caso o desenvolvedor chame o `connection.close()`, ela será fechada em todas as partes solicitadas.

```
> ConnectionFactory.getConnection().then(connection => connection.close()));
```

A única forma aceitável para que o desenvolvedor faça isso será da seguinte maneira:

```
> ConnectionFactory.closeConnection();
```

O desenvolvedor não poderá obter uma conexão e a partir desta, fechar, porque assim pode ocorrer um problema geral na aplicação. Para isto, vamos utilizar o **Monkey Patch**, que consiste forçarmos a modificação de uma API. No caso, nós iremos alterar o método `close()`.

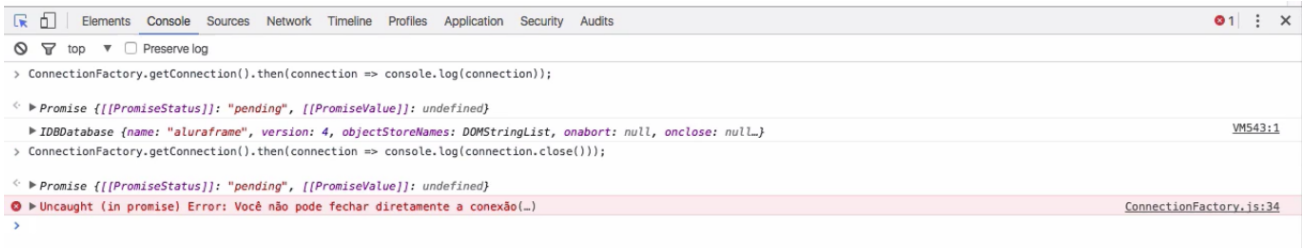
```
openRequest.onsuccess = e => {

    if(!connection) {
        connection = e.target.result;
        connection.close = function() {
              throw new Error('Você não pode fechar diretamente a conexão');
        };
    }
    resolve(connection);
}
```

Se o desenvolvedor tentar chamar o `connection.close`, ele receberá essa mensagem de erro. Vamos ver se o código está funcionando. Primeiramente, executaremos o `console.log` da `connection`:

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

E depois, testaremos o `connection.close()`:

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

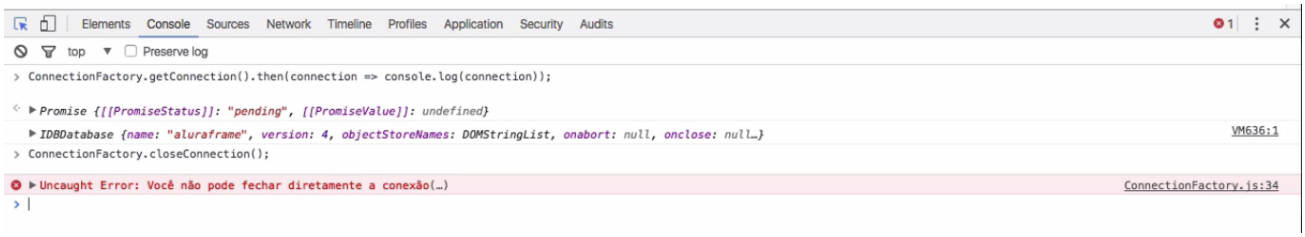


Veremos um mensagem de erro. Funcionou! Nós fizemos um *Monkey Patch* do método `close()`, substituindo-o por um novo código que ignora o anterior e impede que o desenvolvedor feche a nossa conexão. Em seguida, adicionaremos o método `closeConnection()` para testar se existe uma `connection`. Caso exista, ele chamará o `close` e informará que `connection` é igual a `null`:

```
static closeConnection(){

    if(connection){
        connection.close();
        connection = null;

    }
}
```

Se alguém chamar o `getConnection` novamente, verá que a conexão é nula e obterá uma nova. No navegador, pediremos que ele faça a conexão e, depois, chamaremos o `closeConnection`:

```
> ConnectionFactory.closeConnection();
```



Mas não conseguimos fechar a conexão. Isto está ocorrendo porque nós destruímos método `close()` no `onsuccess`, logo, não conseguiremos chamá-lo no `closeConnection`. Como podemos resolver o problema? Começaremos adicionando uma variável `close` abaixo da `connection`:

```
var ConnectionFactory = (function() {

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

    var connection = null;
    var close = null;

    return  class ConnectionFactory {

        constructor() {

            throw new Error("ConnectionFactory não pode ser instanciada");
```

```
        }
//...
```

E adicionaremos o `close` no `onsuccess`:

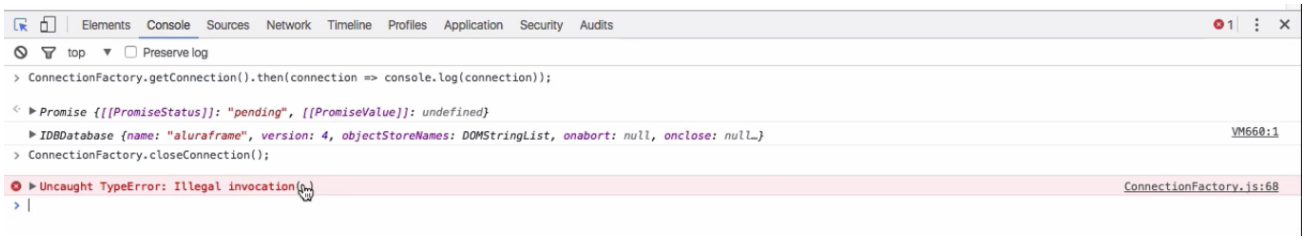
```
openRequest.onsuccess = e => {

    if(!connection) {
        connection = e.target.result;
        close = connection.close;
        connection.close = function() {
              throw new Error('Você não pode fechar diretamente a conexão');
        };
    }
    resolve(connection);
}
```

Nós informamos que a variável `close` receberá a função `close` - que será sobrescrita a seguir. Chamaremos o método `close()` em `closeConnection()`:

```
static closeConnection() {

    if(connection) {
        close();
        connection = null;
    }
}
```

A variável `close` guardou a função antes que esta fosse modificada. Vamos ver se funcionou no navegador, chamando o `closeConnection`:

```
ConnectionFactory.closeConnection();
```



Recebemos uma mensagem de erro. Como o `close` pertence ao objeto `connection`, isto significa que o `this` do `close` será o `connection`. Mas quando pegamos o método e colocamos na variável `close`, ele perdeu o link com o `connection` e ao ser chamado, não conseguirá operar. Vimos anteriormente, que temos formas de contornarmos esta limitação. Quando estivermos copiando o `close` no `onsuccess`, faremos já associado com o `connection` utilizando o `bind`.

```
openRequest.onsuccess = e => {

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

```
            close = connection.close.bind(connection);
            connection.close = function() {
                throw new Error('Você não pode fechar diretamente a conexão');
            };
        }
        resolve(connection);
    }
```

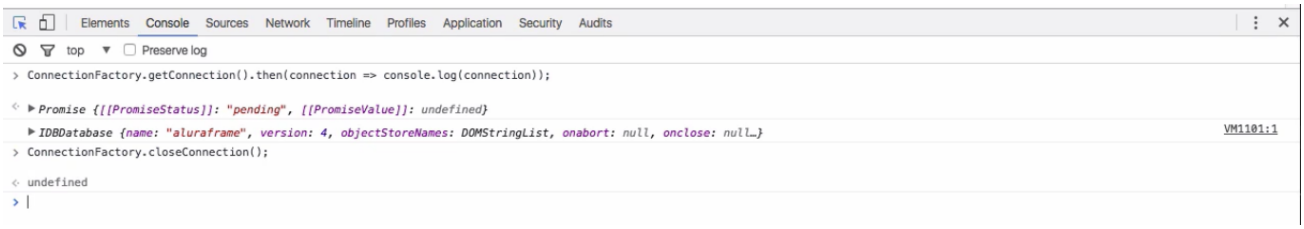
Agora, o `this` será a própria `connection`. Nós já fizemos algo parecido anteriormente nos cursos de JavaScript. Mas se não fizermos o `Bind`, será que temos outra opção? Sim, mas será um pouco mais trabalhosa. Removeremos o `Bind` do `onsuccess`:

```
    openRequest.onsuccess = e => {

        if(!connection) {
            connection = e.target.result;
            close = connection.close;
            connection.close = function() {
                throw new Error('Você não pode fechar diretamente a conexão');
            };
        }
        resolve(connection);
    }
```

E iremos chamar o método `close()`, usando o `Reflect.apply`. Teremos como contexto o `connection` e não receberá parâmetros.

```
    static closeConnection() {

        if(connection) {
            Reflect.apply(close, connection, [])
            connection = null;
        }
    }
```

Vamos testar no navegador.

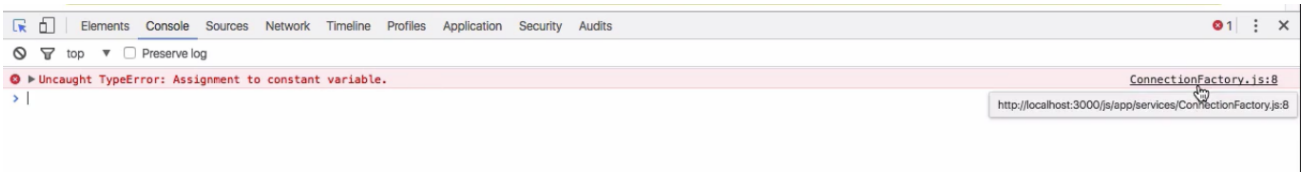


Vimos que funciona a adoção do `Reflect.apply`. No entanto, continuaremos a usar o `Bind`, por ser uma solução mais simples.

Existe uma última melhoria que podemos fazer: observe que faz sentido reatribuirmos os valores das variáveis `connection` e `close`, pois dependendo da situação seus valores podem ser `null`. Contudo, não faz sentido em tempo de execução reatribuirmos os valores das variáveis `dbName`, `version` e `stores`. Podemos declará-las como constantes trocando `var` por `const`:

```
var ConnectionFactory = (function() {

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

    var connection = null;
    var close = null;
```

Com uma constante, não podemos reatribuir seu valor. Por exemplo, se tentarmos alterar o valor de `dbName` para `calopsita` , veja o que acontecerá ser recarregarmos o código:



Receberemos uma mensagem de erro... Não podemos reatribuir.

Isto não significa que os campos sejam imutáveis. Você poderá se aprofundar no assunto nos exercícios (https://cursos.alura.com.br/course/javascript-es6-orientacao-a-objetos-parte-3/task/19782). Mas sempre que tivermos uma variável cujo valor não queremos que seja reatribuido, usaremos o `const` . Pratique o conteúdo fazendo os exercícios e mais adiante, implementaremos um novo padrão de projeto.