08

Construindo armadilhas para métodos

Transcrição

Vamos procurar a solução "cangaceira" para resolvermos o nosso problema. Lembra que anteriormente falamos que quando chamamos uma função, o JavaScript fará um getter e após a leitura, será enviada um `apply`. Teremos que substituir o `set` para o `get`:

```
<script>

  let lista = new Proxy(new ListaNegociacoes(), {
    get: function(target, prop, receiver) {

    }
  });
    lista.adiciona(new Negociacao(new Date(), 1, 100));
</script
```

Observe que o `get` não receberá o `value`. A questão é: quando o getter for executado, queremos perguntar se ele está na lista de métodos que queremos interceptar. Para isto, adicionaremos um `if` para o `get`.

```
get: function(target, prop, receiver) {
    if(['adiciona', 'esvazia'].includes(prop) && type(target[prop]) == typeof(Function)) {

    }
}
```

Criaremos uma condição que testará se o método incluído é o `adiciona()` ou o `esvazia()`, que tem ou não `prop`s (uma novidade do ES6 é podermos fazer este tipo de pergunta para o array) e se é uma função. Para testarmos esta última parte, usamos o `typeof[]`, que recebeu a propriedade do `target`. Se isso é uma função ou método, o `typeof` será o parâmetro. Vamos verificar se isso é o `typeof` de `Function`.

Se viermos no Console e digitarmos `Function`, veremos que existe uma função que existe `function` em JavaScript:

```
Function
function Function() { [native code] }
```

Se `adiciona()` e `esvazia()` tem a propriedade e é uma função, faremos algo a respeito. Caso contrário, é a leitura de um `get` padrão.

```
<script>

  let lista = new Proxy(new ListaNegociacoes(), {
    get: function(target, prop, receiver) {
        if(['adiciona', 'esvazia'].includes(prop) && type(target[prop]) == typeof(Function)

        }
```

```
            return Reflect.get(target, prop, receiver);
        }
        });
          lista.adiciona(new Negociacao(new Date(), 1, 100));
    </script>
```

Se está ou não na nossa lista, teremos o retorno do valor, considerando que estamos fazendo um `get`. Adicionaremos o `lista._negociacoes`:

```
lista.adiciona(new Negociacao(new Date(), 1, 100))
lista._negociacoes;
```

Com isso, já faremos um get. E o que iremos devolver? Se a propriedade está na lista (de `adiciona()` ou `esvazia()`) e é uma função, em vez de retornarmos um valor direto, retornaremos uma função.

Tem que ser `function` para ter o this dinâmico. Não pode ser *arrow function* que possui escopo léxico.

A função não pode ser arrow function, porque ela deve ter um contexto dinâmico, e dentro, substituiremos o método por outro que tem a armadilha - porém, a substituição será feita no Proxy. No entanto, este não nos permite colocar uma armadilha para método, encontraremos um forma de que ao cairmos no método substituiremos por outro do Proxy. Falamos do `return`.

```
<script>

  let lista = new Proxy(new ListaNegociacoes(), {

    get: function(target, prop, receiver) {
        if(['adiciona', 'esvazia'].includes(prop) && typeof(target[prop]) == typeof(Functio
        return function() {
console.log(`a propriedade "${prop}" foi interceptada`);
            }
        }
        return Reflect.get(target, prop, receiver);
    });
      lista.adiciona(new Negociacao(new Date(), 1, 100));
    lista._negociacoes
</script>
```

Agora, teremos que fazer o método receber os parâmetros que ele está recebendo aqui. Como estamos substituindo e retornando a função, substituiremos o `adiciona()`, quando este método for chamado estaremos chamando na verdade, o `console.log`. No entanto, precisamos fazer com que o `adiciona()` receba o parâmetro original dele:

```
lista.adiciona(new Negociacao(new Date(), 1, 100));
```

Como a função no `if`, substituirá o método `adiciona()`, existe um objeto implícito chamado `arguments` que dá acesso a todos os parâmetros passados para a função. A seguir, usaremos o `Reflect.apply()`, e chamaremos uma função.

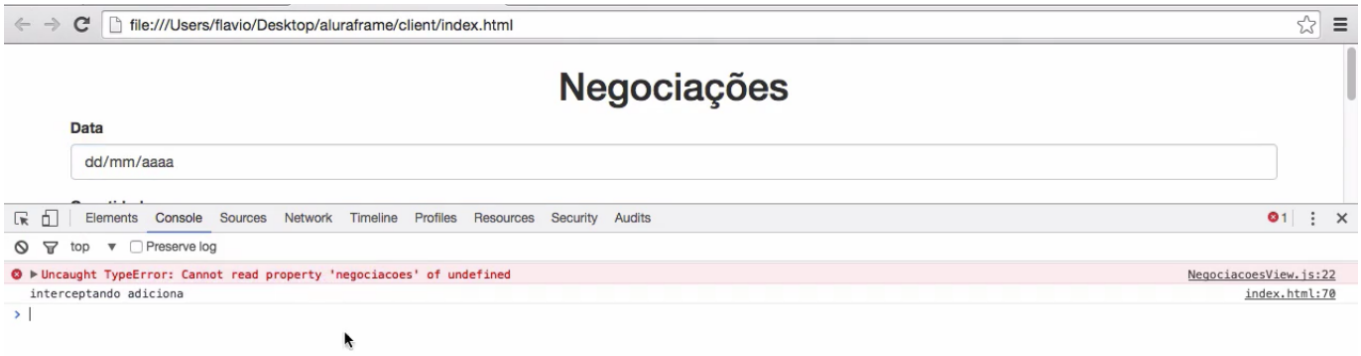
```
if(['adiciona', 'esvazia'].includes(prop) && typeof(target[prop]) == typeof(Function)) {
```

```
        return function() {

            console.log(`a propriedade "${prop}" foi interceptada`);
            Reflect.apply(target[prop], target, arguments);
    }
```

Vamos entender o que foi feito até aqui... Ao fazemos métodos e funções o Proxy sempre entende que é um `get`, quando fazemos o `lista.adiciona()`. No `get`, perguntaremos "você está na lista de itens que quero interceptar? E você é uma função?" Caso a resposta seja positiva, iremos substituir o `adiciona()` ou o `esvazia()` no Proxy. A substituição será feita por uma nova função. Então, ao ser chamada, a função imprimirá o conteúdo do `console.log`, porque a função lembrará do contexto de execução e quem é `prop`. Com o `Reflect.apply()`, faremos a função receber os parâmetros dela. O `arguments` é uma variável implícita que dá acesso a todos os parâmetros da função quando esta é chamada. Foi uma maneira de via `get` ter acesso aos parâmetros da função.

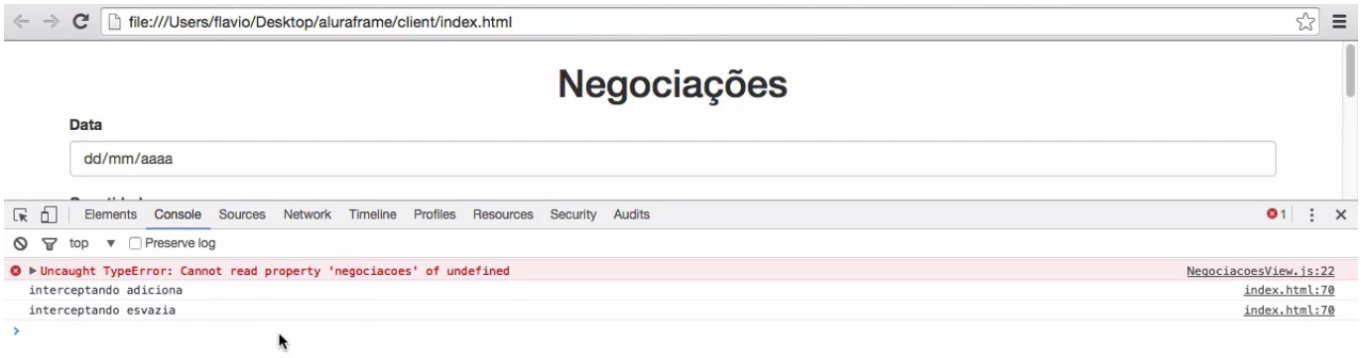
Ao recarregarmos a página no navegador, veremos a confirmação de que foi `interceptado adiciona`.



O mesmo ocorrerá se chamarmos o `lista.esvazia()`:

```
lista.adiciona(new Negociacao(new Date(), 1, 100));
lista.esvazia();
```

No Console, veremos o `interceptado esvazia`.



Desta forma, encontramos uma maneira de escolher qual método queremos interceptar e executar o código. Antes de continuarmos, faremos um pequeno ajuste no `get`:

```
get(target, prop, receiver) {
    if(['adiciona', 'esvazia'].includes(prop) && typeof(target[prop]) == typeof(Function))

        return function() {
```

```
            console.log(`a propriedade "${prop}" foi interceptada`);
            Reflect.apply(target[prop], target, arguments);
          }

      }
      return Reflect.get(target, prop, receiver);

  }
```

Agora, precisamos substituir o trecho de código no `Controller` . Vamos aplicar efetivamente no nosso sistema.