

JavaScript Avançado III: ES6, orientação a objetos e padrões de projetos

43%

Abrir caderno

AULA 03

Padronizando acesso aos dados com o pattern DAO

ATIVIDADES

01

O padrão de projeto DAO

13min

02

Combinando padrões de projeto

09min

03

Exibindo todas as negociações

11min

04

Removendo todas as negociações

07min

05

O padrão DAO

06

Método que devolve uma promise

07

Combinando ConnectionFactory e Neg...

08

Consolidando seu conhecimento

09

Para saber mais: IndexedDB e transaç...

10

Para saber mais: bibliotecas que encaps...

AULAS

03. Padronizando acesso aos dados com o pa

OUTROS LINKS

Fórum

Trocar Curso

T

Thais Poentes

14.2k xp

a

Se você já trabalhou com algum banco de dados relacional deve ter reparado que em nenhum momento chamamos métodos como `commit` ou `rollback` para consolidar a transação ou abortá-la. Por mais que isso possa lhe causar certo espanto, o IndexedDB trabalha um pouquinho diferente.

### Transações do IndexedDB são auto committed

É por meio de uma transação que temos acesso a uma store e dela podemos realizar operações como a inclusão de um objeto. Quando essa operação é realizada com sucesso, ou seja, quando o evento `onsuccess` é chamado a transação é fechada, ou seja, as transações do IndexedDB são *auto committed*. É por isso que cada método do nosso `NegociacaoDao` solicita uma transação toda vez que é chamado.

### Podemos cancelar uma transação através do método `abort`

Ótimo, já sabemos quando uma transação é efetivada e que este é um processo automático, no entanto nem sempre queremos efetivá-la, ou seja, queremos abortá-la. Fazendo uma alusão aos bancos de dados relacionais, queremos ser capazes de realizar um `rollback`.

Para cancelarmos (rollback) uma transação podemos chamar o método `abort`:

```
ConnectionFactory.  
  .getConnection()  
  .then(connection => {  
  
    let transaction = connection.transaction(  
  
    let store = transaction.objectStore(  
  
    let negociacao = new Negociacao(new Date(),  
  
    let request = store.add(negociacao);  
  
    // ##### VAI CANCELAR A TRANSAÇÃO. O e  
    transaction.abort();  
  
    request.onsuccess = e => {  
  
      console.log('Negociação incluída  
    };  
  
    request.onerror = e => {  
  
      console.log('Não foi possível incl  
    };  
  
  })
```

Ao executar o código a seguinte mensagem de erro será exibida no console:

```
DOMException: The transaction was aborted, so the  
Não foi possível incluir a negociação
```

### Trate o cancelamento de uma transação no evento `onabort` de transaction

Contudo, podemos tratar os erros de uma transação abortada no evento `onabort` da transação, ao invés de lidarmos com ele em `onerror`.

```
ConnectionFactory.  
  .getConnection()  
  .then(connection => {  
  
    let transaction = connection.transaction(  
  
    let store = transaction.objectStore(  
  
    let negociacao = new Negociacao(new Date(),  
  
    let request = store.add(negociacao);  
  
    // ##### VAI CANCELAR A TRANSAÇÃO. O e  
  
    transaction.abort();  
    transaction.onabort = e => {  
      console.log(e);  
      console.log('Transação abortada');  
    };  
  
    request.onsuccess = e => {  
  
      console.log('Negociação incluída  
    };  
  
    request.onerror = e => {  
  
      console.log('Não foi possível incl  
    };  
  
  })
```

Apesar do que aprendemos aqui não ser útil dentro do cenário da aplicação Aluraframe, informações extras como essa são sempre bem-vindas!