Persistência de dados com React Native

Implementação de persistência de dados na plataforma React Native, com a utilização de diversas formas de armazenamento, incluindo dados planos em arquivos de texto, bases relacionais, bases NoSQL e modelo orientado a objetos.

Prof. Denis Gonçalves Cople

1. Itens iniciais

Propósito

Ao final dos estudos, você deverá estar apto a construir aplicativos com persistência de dados, por meio da plataforma React Native, utilizando diversas metodologias para armazenamento. Com o conhecimento adquirido, será possível definir o modelo de armazenagem de dados para os mais diversos aplicativos, desde sistemas simples, com base em AsyncStorage, SQLite ou Realm, até a utilização de NoSQL, via MongoDB.

Preparação

Antes de iniciar este conteúdo, é necessário configurar o ambiente, com a instalação do JDK, Android Studio, Visual Studio Code, MongoDB e Node.js, além da inclusão, via NPM ou YARN, dos ambientes react-native-cli e expo-cli. Também deve ser incluída, no Visual Studio Code, a extensão React Native Tools, e o dispositivo de testes deve ter o aplicativo Expo, obtido gratuitamente na loja da plataforma.

Objetivos

- Empregar AsyncStorage para a persistência local de dados.
- Descrever a persistência no modelo relacional, com base em SQLite.
- Empregar Realm para a persistência no modelo orientado a objetos.
- Descrever a persistência no modelo NoSQL, com base no MongoDB.

Introdução

Neste conteúdo, analisaremos as ferramentas para persistência de dados mais comuns no ambiente do React Native, a começar pelo armazenamento local, em que temos opções como AsyncStorage, banco relacional SQLite e banco orientado a objetos Realm.

No módulo final, utilizaremos uma abordagem NoSQL, com base em MongoDB, para o armazenamento remoto. Veremos como configurar a base de dados no MongoDB, criar um servidor minimalista, com a biblioteca Express, e utilizar a biblioteca axios na definição de um aplicativo cliente do tipo React Native.

Serialização e persistência

Sistemas de informação podem ser descritos como ferramentais tecnológicos capazes de trazer significado para um conjunto de dados, o que nos leva à necessidade natural de acumular os dados que serão interpretados. Em se tratando do **armazenamento de dados**, temos dois conceitos muito importantes, que são a **serialização** e a **persistência**.

Quando as estruturas de dados, como objetos, estão alocadas na memória, as regiões utilizadas são distribuídas de forma esparsa, muitas vezes inter-relacionadas, tornando inviável a transmissão ou a armazenagem dos dados representados. A solução, nesses casos, está na serialização e na persistência dos dados. Veja, a seguir, uma breve definição desses conceitos:

Serialização

Definição: É um modo de expressar os dados de forma contígua, podendo ser uma sequência de bytes ou algum padrão de texto conhecido, como XML (eXtended Markup Language) ou JSON (Java Script Object Notation).**Objetivo**: Permitir a transmissão em rede ou a armazenagem em disco dos dados serializados.

Persistência

Definição: Uma estrutura dotada de persistência ou persistente é aquela capaz de guardar seu estado corrente, normalmente em meio não volátil, como as unidades de disco.**Objetivo**: Recuperar posteriormente o seu estado corrente.

Ao trabalhar com linguagens como **Java** e **C++**, é comum ocorrer a serialização binária, na qual os bytes esparsos são representados em uma sequência contínua, mas ambientes baseados em JavaScript normalmente trabalham com representações textuais. No caso do **React Native**, é comum representar objetos no formato **JSON**.

Podemos observar, a seguir, a definição da classe **Produto**, na sintaxe **TypeScript**, que deve ser salva em um arquivo com o nome "**Produto.ts**".

```typescript
export class Produto{
     codigo: number;
     nome: string;
     quantidade: number;
     constructor(codigo: number, nome: string, quantidade: number){
         this.codigo = codigo;
         this.nome = nome;
         this.quantidade = quantidade;
     }
 }
```

Essa classe poderia ser utilizada na programação em **JavaScript** ou **TypeScript**, com a alocação efetuada pelo operador **new**, levando à ocupação de áreas de memória não contíguas. Para que possamos transmitir as informações representadas por cada um dos objetos, ou armazená-las em arquivo, devemos utilizar um formato apropriado em modo texto, no caso **JSON**.

A transformação de um objeto da classe Produto para texto **JSON**, bem como o processo contrário, podem ser observados no fragmento de código seguinte.

```
typescript

import { Produto } from './Produto';

 let prod1: Produto = new Produto(1,'Teclado',50);
 let prod1Str: string = JSON.stringify(prod1);
 let prod2: Produto = JSON.parse(prod1Str);
```

No código de exemplo, vemos a utilização da classe **JSON**, que faz parte do núcleo básico da plataforma **Node.js**, com a serialização do objeto prod1 pelo método **stringify**, bem como a recuperação do seu estado em um segundo objeto, processo que também é conhecido como de-serialização, por meio do método **parse**. Teremos, ao longo deste estudo, uma ampla utilização dos dois métodos descritos.

Ao criarmos servidores para Web baseados no Node.js, é muito simples gravar e ler as informações contidas em arquivos, utilizando a biblioteca **fs** (File System)

```
typescript

const fs = require('fs');

 let prod1 = null;
 try{
     let prodStr = fs.readFileSync('prod1.json');
     prod1 = JSON.parse(prodStr);
 }catch(err){
     prod1 = {codigo: 1, nome: 'Teclado', quantidade: 50};
     let prodStr = JSON.stringify(prod1);
     fs.writeFileSync('prod1.json',prodStr);
 }
```

Pelo método **readFileSync** é possível efetuar a leitura de um arquivo diretamente para a variável de texto, com subsequente transformação em objeto por meio de **parser**, enquanto a gravação ocorre por intermédio de **writeFileSync**. Infelizmente a gravação de forma síncrona não é totalmente compatível com o funcionamento dos dispositivos móveis, pois existe grande preocupação com a possibilidade de travamento da interface gráfica, e a biblioteca fs não é um padrão para o **React Native**, devendo ser utilizado o modelo assíncrono, com base em **AsyncStorage**, como será visto no decorrer deste conteúdo.

Parser

Parser é um analisador sintático que determina a estrutura lógica de um texto, de acordo com uma sintaxe de linguagem.

Formatos interoperáveis

Computadores transformam dados em informações, mas cada plataforma lidará com os modelos de interpretação e as estruturas de armazenagem de forma diferenciada. Por conta disso, não é raro observar diversos sistemas herméticos trabalhando a mesma informação de outros, o que significa um grande retrabalho que poderia ser amenizado.

Com a evolução da tecnologia, os repositórios de dados se tornaram cada vez maiores, e a replicação inevitável, pois as plataformas inicialmente não foram projetadas para que ocorresse o compartilhamento desses dados, algo que se tornou evidente com o surgimento e a popularização das redes de computadores.

Comentário

Muitos projetos apresentavam características semelhantes, mas, como não existiam interfaces comuns, ocorria um grande retrabalho na construção de novos produtos e bases de dados.

A partir dessa percepção, o compartilhamento de informações e serviços se tornou uma preocupação para os desenvolvedores, inicialmente com a padronização de formatos de arquivo e protocolos de rede, delineando as bases do que hoje conhecemos como **interoperabilidade**.

Interoperabilidade

Interoperabilidade é a habilidade que algumas plataformas oferecem para se comunicar por meios padronizados, viabilizando sistemas heterogêneos.

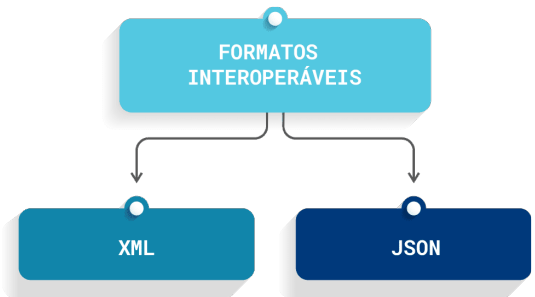
Bases de dados relacionais, como **Oracle** e **MySQL**, são exemplos clássicos de produtos que permitem a interoperabilidade, uma vez que o sistema de gerenciamento de dados pode ser acessado por diversas plataformas, como Java e C#, desde que elas respeitem os protocolos de rede definidos para acesso ao banco. Em vez de nos preocuparmos em tratar os dados armazenados em formatos proprietários, utilizamos uma linguagem de consulta padronizada, denominada **SQL** (Structured Query Language), permitindo que as instruções sejam mantidas, mesmo com a mudança de fornecedor.

Comentário

Embora o uso de uma sintaxe de consulta padronizada facilite a interoperabilidade no nível do compartilhamento de dados, ainda temos sistemas que não são capazes de se comunicar diretamente.

Por meio do compartilhamento de serviços, segundo protocolos e formatos padronizados, chegamos a um nível muito maior de interoperabilidade, algo que foi viabilizado com o surgimento de **Web Services**.

Na comunicação **B2B** (Business to Business), temos o uso de XML, que apresenta maior formalismo, através dos Web Services do tipo **SOAP** (Simple Object Access Protocol), enquanto o uso de **REST** (Representational State Transfer), que apoia as operações de consulta e cadastro nos métodos do HTTP, define um segundo tipo de Web Service, com o nome **RESTful**, o qual adota uma grande variedade de formatos para os dados, sendo definido o **JSON** como padrão, e é voltado para o **B2C** (Business to Consumer).



A criação de um Web Service pode ser feita em qualquer linguagem, mas toda a comunicação é efetuada com o uso de **XML** ou **JSON**, formatos amplamente utilizados no mercado, que permitem representar qualquer tipo de informação em modo texto, com neutralidade de linguagem, motivo pelo qual são classificados como **formatos interoperáveis**.

Como já pudemos observar, a plataforma do React Native oferece grande facilidade para transformar objetos das linguagens JavaScript e TypeScript no formato de texto JSON, assim como para processar os textos na recuperação dos objetos.

Atenção

A serialização para o formato JSON é necessária para a comunicação com Web Services do tipo RESTful, sendo também adequada para armazenamento em disco, mas todas as operações de entrada e saída, tanto na rede quanto em arquivos, exigirão operações assíncronas.

Armazenamento com AsyncStorage

Uma forma simples de persistência assíncrona é pelo **AsyncStorage**, que permite guardar e recuperar pares do tipo **chave-valo**r, ou seja, um identificador e um dado associado, da mesma forma que uma chave primária identificando um registro em uma tabela. Para utilizar objetos como valores, eles devem estar serializados no formato **JSON**, com a utilização dos métodos descritos anteriormente.

O quadro apresentado a seguir descreve os principais métodos de AsyncStorage.

Método	Parâmetros	Callback	Funcionalidade
getItem	Chave	(error, result) ⇒ void	Obtém um item da coleção a partir da chave fornecida.
setItem	Chave Valor	(error) ⇒ void	Armazena um novo item da coleção a partir da chave.
removeItem	Chave	(error) ⇒ void	Remove um item a partir da chave fornecida.
mergeItem	Chave Valor	(error) ⇒ void	Atualiza um item da coleção localizado pela chave.
getAllKeys	(Nenhum)	(error, keys[ ]) ⇒ void	Obtém todas as chaves que estão na coleção.
multiGet	Chave[ ]	(erros[ ], result[ ]) ⇒ void	Obtém múltiplos itens.

Tabela: Alguns dos métodos do AsyncStorage.

Para utilizar os métodos de AsyncStorage, será necessário adotar elementos da sintaxe voltados para a sincronização em ambiente assíncrono, tais como:

async

Na definição de funções chamadoras.

await

Na espera pela execução do método.

> Outra possibilidade é a utilização do operador then, seguindo uma abordagem funcional.

Por exemplo, podemos armazenar a configuração de um tema ou assunto para um aplicativo, por meio de código em sintaxe TypeScript, como pode ser observado a seguir.

```
typescript

const salvarTema = async (valorTema: string) => {
     try {
         await AsyncStorage.setItem('tema', valorTema);
     } catch (e) {}
 }
```

Do mesmo modo, é possível recuperar a configuração armazenada, em uma execução posterior do aplicativo, como pode ser visto na listagem seguinte. Note que o retorno de uma função assíncrona deve ser um elemento **Promise** para o tipo desejado.

```
typescript

const recuperarTema = async (): Promise => {
     try {
         return await AsyncStorage.getItem('tema');
     } catch (e) {}
     }
```

Podemos observar, na listagem seguinte, uma biblioteca de persistência para a classe **Produto**, obedecendo à sintaxe do TypeScript. Ela deverá ser salva em um arquivo com o nome "**GestorDados.ts**".

```typescript
import AsyncStorage
        from '@react-native-async-storage/async-storage';
 import { Produto } from './Produto';

 const salvarProduto = async (key: string, value: any) => {
   try {
     const jsonValue = JSON.stringify(value);
     await AsyncStorage.setItem(key, jsonValue);
   } catch (e) {}
 }
 const removerProduto = async (key: string) => {
   try {
     await AsyncStorage.removeItem(key);
   } catch(e) {}
 }
 const obterProdutosJSON = async () => {
   try {
     let keys: Array = []
     keys = await AsyncStorage.getAllKeys();
     return await AsyncStorage.multiGet(keys);
   } catch(e) { return []; }
 }
 const obterProdutos = async () => {
   try {
     let objetos = [];
     let objJSON = await obterProdutosJSON();
     if(objJSON!=null && objJSON.length>0)
       objJSON.forEach(element => {
         let produto: Produto = JSON.parse(element[1]);
         objetos.push(produto);
       });
     return objetos;
   } catch(e) { return []; }
 }

 class GestorDados{
   public async remover(chave: number){
     removerProduto(chave.toString());
   }
   public async adicionar(produto: Produto){
     salvarProduto(produto.codigo.toString(),produto);
   }
   public async obterTodos(): Promise>{
     let lista: Array = await obterProdutos();
     return lista;
   }
 }

 export default GestorDados;
```

Após as importações necessárias, vemos a criação da função assíncrona **salvarProduto**, tendo como parâmetros a **chave**, do tipo string, e o **objeto** que será persistido. No corpo da função, temos a transformação do objeto para texto JSON, seguido da armazenagem em disco com o uso de **setItem**, em que ocorre o bloqueio de execução por meio de **await**.

Os mesmos procedimentos são utilizados nos métodos **removerProduto**, em que temos a chamada para **removertem**, a partir da chave fornecida, e **obterProdutosJSON**, que inicia com a recuperação das chaves mediante **getAllKeys**, seguido da obtenção de todos os produtos, no formato **JSON**, com a chamada para

multiGet. Já na função **obterProdutos**, temos a de-serialização dos valores recuperados no formato JSON, finalizando com o retorno de um vetor com todos os produtos armazenados em disco.

Prosseguindo com a análise do código, definimos uma classe com o nome **GestorDados**, apresentando os métodos **remover**, **adicionar** e **obterTodos**, e efetuando as chamadas necessárias para as funções criadas anteriormente.

Comentário

Observe a necessidade da utilização de Promise no retorno de obterTodos, uma vez que teremos um vetor de produtos fornecido de forma assíncrona.Ao final, a classe é exportada, e nossa camada de persistência está completa, permitindo a definição de uma interface gráfica adequada para sua utilização.

AsyncStorage

O modelo de persistência local, utilizando o componente AsyncStorage é apresentado no vídeo a seguir.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Cadastro baseado em arquivos

Vamos criar um aplicativo cadastral simples, baseado na navegação padrão entre telas, utilizando as classes criadas anteriormente.

Para configurar nosso projeto, devemos executar os comandos listados a seguir, com base no ambiente **expo**, template **blank**.

```
python

expo init CadastroLocal
 cd CadastroLocal
 expo install @react-native-async-storage/async-storage
 npm install @react-navigation/native @react-navigation/stack
 expo install react-native-gesture-handler react-native-reanimated react-native-screens
react-native-safe-area-context @react-native-community/masked-view
```

Com o projeto configurado, vamos criar o diretório **dados**, em que colocaremos as classes **Produto** e **GestorDados**, além de criar, com a listagem seguinte, o arquivo "**CommonStyles.js**", onde teremos a concentração de todos os estilos utilizados no aplicativo.

```javascript
 import { StyleSheet } from 'react-native';

export const styles = StyleSheet.create ({container: {flex: 1, backgroundColor: '#fff',
marginTop: 10, width: '100%'},scrollContainer: {width: '90%'},

itemsContainer: {marginTop: 10, padding: 20, borderTopLeftRadius: 10,
borderTopRightRadius: 10, alignItems: 'stretch', backgroundColor: '#fff'},

inputContainer: {flex: 1, marginTop: 30, marginLeft: '5%', width: '90%', padding: 20,
alignItems: 'stretch', backgroundColor: '#fff'},

buttonsContainer: {flexDirection: 'row-reverse', alignItems: 'flex-end',
borderBottomWidth: 1, borderBottomColor: '#CCC', paddingBottom: 10, marginTop: 10},

input: {marginTop: 10, height: 60, backgroundColor: '#fff', borderRadius: 10,
paddingHorizontal: 24, fontSize: 16, alignItems: 'stretch'},

button: {marginTop: 10, marginBottom: 10, height: 60, backgroundColor: 'blue',
borderRadius: 10, paddingHorizontal: 24, fontSize: 16, alignItems: 'center',
justifyContent: 'center', elevation: 20, shadowOpacity: 20, shadowColor: '#ccc'},

buttonText: {color: '#fff', fontWeight: 'bold'},

buttonTextBig: {color: '#fff', fontWeight: 'bold', fontSize: 24},

textItem: {fontSize: 20},

deleteButton: {marginLeft: 10, height: 40, width: 40, backgroundColor: 'red',
borderRadius: 10, padding: 10, fontSize: 12, elevation: 10, shadowOpacity: 10,
shadowColor: '#ccc', alignItems: 'center'},

});
```

A utilização de um gestor de folhas de estilo em um arquivo único, além de minimizar a quantidade de código necessária nas telas do aplicativo, permite efetuar mudanças no design de forma global, atuando como um tema. Terminada a definição dos estilos, já estamos aptos para a criação de nossa interface visual.

Começaremos a definição das telas com a criação do arquivo "**ProdutoForm.js**", que será utilizado para a inclusão de um produto. O código do novo arquivo é apresentado na listagem seguinte.

```typescript
import React, {useState} from 'react';
                         import { Text, View, TextInput, TouchableOpacity }
                                from 'react-native';
                         import { Produto } from './dados/Produto';
                         import GestorDados from './dados/GestorDados';
                         import { styles } from './CommonStyles';

                         export default function ProdutoForm( { navigation } ) {
                           const gestor = new GestorDados();
                           const[codigo,setCodigo] = useState('');
                           const[nome,setNome] = useState('');
                           const[quantidade,setQuantidade] = useState('');
                           const salvar = () => {
                             prodAux =
                                 new Produto(parseInt(codigo),nome,parseInt(quantidade));
                             gestor.adicionar(prodAux).then(
                                 navigation.navigate('ListaProd'));
                           }

                           return (




                                   Salvar


                           );
                         }
```

Seguindo a definição padrão de componentes para React Native, teremos o retorno de uma **View** composta de três componentes **TextInput**, relacionadas aos atributos código, nome e quantidade, além de um botão do tipo **TouchableOpacity** com o texto Salvar.

A alteração dos valores nos campos de texto acionará os métodos de escrita sobre os atributos referenciados, e o clique sobre o botão inicia a função **salvar**. Com a execução da função, ocorre a geração de uma instância de **Produto**, com os dados dos atributos, e a armazenagem do novo produto, invocando **adicionar** no objeto **gestor**, seguido da **navegação** para a tela principal (**ListaProd**), a qual apresentará um botão para iniciar o cadastro e uma lista com os produtos existentes.

Atenção

O uso do operador then permite que a navegação ocorra apenas quando a inclusão do produto for efetivada, no retorno método assíncrono.

Como utilizaremos uma lista, inicialmente criaremos o arquivo "**ProdutoItem.js**", o qual será codificado de acordo com a listagem apresentada a seguir.

```
typescript

import React from 'react';
 import {Text, View, TouchableOpacity} from 'react-native';
 import { styles } from './CommonStyles';

 export default function ProdutoItem(props){
     return (


         {props.produto.codigo} - {props.produto.nome}

         Quantidade: {props.produto.quantidade}


             X



     );
 }
```

No código, é possível observar o componente que será utilizado para a visualização de um produto na lista. Temos dois elementos do tipo **Text** para exibir os atributos de um **produto**, fornecido pelas propriedades (**props**) da tag, além de um botão que irá acionar a função **onDelete**, também fornecida pelas propriedades, para excluir o produto na ocorrência de um clique.

Agora que o item de lista está pronto, vamos criar o arquivo **"ProdutoLista.js"**, de acordo com a listagem seguinte, definindo nossa tela principal.

```
typescript

import React, {useState, useEffect} from 'react';
 import { Text, View, TouchableOpacity, FlatList }
     from 'react-native';
 import GestorDados from './dados/GestorDados';
 import ProdutoItem from './ProdutoItem'
 import { styles } from './CommonStyles';
 import { useIsFocused } from '@react-navigation/native';

 export default function ProdutoLista( { navigation } ) {
 const gestor = new GestorDados();
 const [produtos, setProdutos] = useState([]);

 const isFocused = useIsFocused();

 useEffect(() => {
     gestor.obterTodos().then(objs => setProdutos(objs));
 }, [isFocused]);

 const myKeyExtractor = item => {
     return item.codigo.toString();
 };

 function excluirProduto(codigo){
     gestor.remover(codigo).then(
     gestor.obterTodos().then(objs => setProdutos(objs))
     );
 }

 return (

     navigation.navigate('NovoProd')}>
         Novo Produto


         excluirProduto(item.codigo)}
         produto={item} />
         }
     />

 );
 }
```

Tudo que temos na tela principal é um botão do tipo **TouchableOpacity**, para a abertura da tela de cadastro e um componente do tipo **FlatList**, para a lista de produtos. O clique no botão faz apenas uma chamada simples para **navigation**, tendo como destino a tela de cadastro de produtos (**NovoProd**).

A lista é alimentada a partir de um vetor observável, com o nome de produtos, em que cada item possui a propriedade **onDelete** associada a uma chamada para **excluirProduto**, tendo como parâmetro o código do item corrente, além da passagem do item em si, por meio de uma propriedade denominada **produto**. Observe a necessidade de uma função para extração da chave correta a partir do item, com o nome **myKeyExtractor**, a qual retorna o código do produto no formato texto.

O carregamento da lista ocorre nos momentos em que a tela ganha o foco, algo que é garantido pelo método **useEffect**, com o fluxo de carga no primeiro parâmetro e o atributo **isFocused** no segundo.

Comentário

O atributo que acompanha o foco é definido a partir do método useIsFocused, oferecido pela biblioteca de navegação padrão do React Native, automatizando a atualização do valor durante a execução do aplicativo.

Note que, no preenchimento da lista, temos uma chamada para **obterTodos**, de forma assíncrona, o que exige o operador **then** para esperar a conclusão da consulta, sendo o retorno associado, em seguida, ao vetor de produtos que alimenta a **FlatList**. No método **excluirProduto**, temos ampla utilização do operador **then**, pois inicialmente aguardamos a conclusão da chamada para remover, depois invocamos o método **obterTodos**, e por fim efetuamos o carregamento da **FlatList** com o valor retornado, ao final da execução, segundo o mesmo processo adotado no controle de foco.

Finalmente, alteramos o arquivo principal do aplicativo (**App.js**), definindo o sistema de navegação, conforme a listagem que se segue.

```
typescript

import { NavigationContainer } from '@react-navigation/native';
 import { createStackNavigator } from '@react-navigation/stack';
 import React from 'react';
 import ProdutoForm from './ProdutoForm';
 import ProdutoLista from './ProdutoLista'

 const Stack = createStackNavigator();

 export default function App() {
     return (









     );
 }
```

Temos uma navegação constituída de duas telas, em que:

1

A **primeira** é **ListaProd**, baseada no componente **ProdutoLista**.

2

A **segunda** é referenciada como **NovoProd**, utilizando o componente **ProdutoForm**.

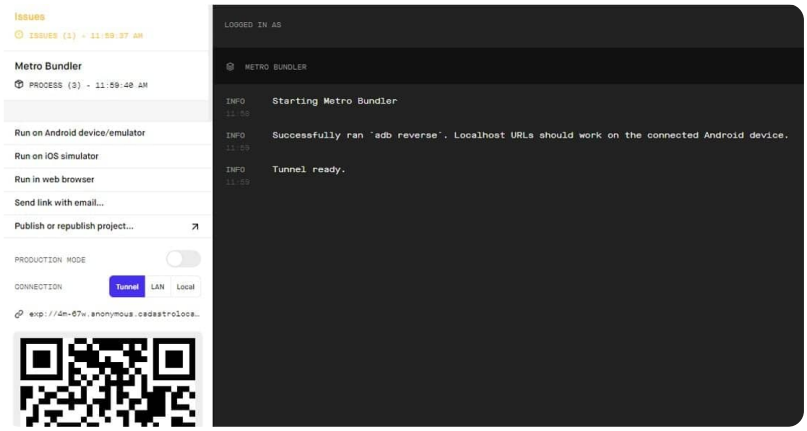
A navegação é do tipo pilha (**Stack**), sendo configurados os títulos das janelas por meio de **options**, no atributo **title**.

Com tudo resolvido, podemos iniciar o ambiente **expo**, com a execução do aplicativo, com o comando seguinte, que deve ser invocado a partir do diretório do projeto.

```
typescript

expo start
```

O resultado da invocação do comando será a abertura do navegador Web padrão, em que teremos a página administrativa do servidor **Expo**. Vamos trabalhar com a opção **Tunnel**, em que é configurado o **tunelamento**, permitindo uma execução bastante simplificada, desde que o servidor e o aplicativo cliente estejam na mesma rede. Veja a imagem a seguir.

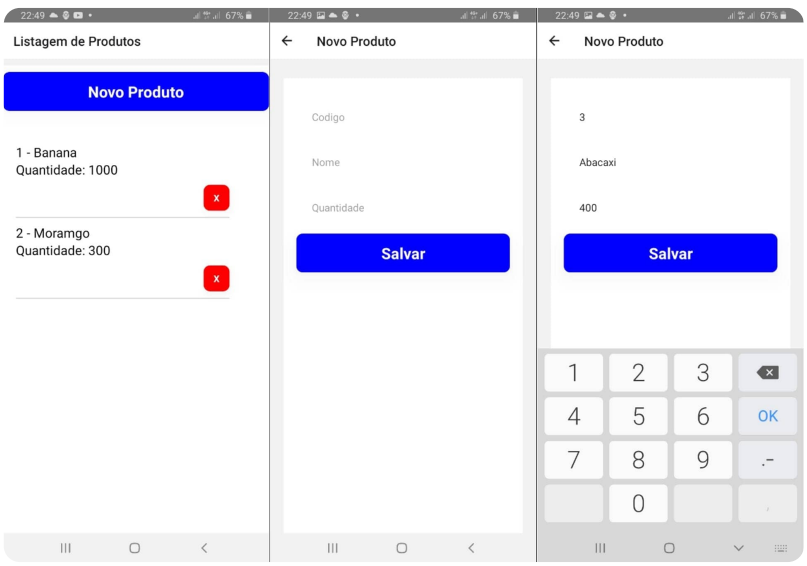


Execução do Expo em modo de tunelamento.

Atenção

No celular, devemos executar o aplicativo Expo, baixado gratuitamente na loja padrão da plataforma, e ler o QR Code a partir do aplicativo. Desde que o dispositivo móvel e o computador estejam na mesma rede, o aplicativo será transferido pelo Metro, iniciando a execução em sequência.

As telas de listagem e inclusão de produtos, durante a execução de nosso aplicativo, podem ser observadas a seguir.



Telas do aplicativo de exemplo.

Verificando o aprendizado

Questão 1

Os sistemas informatizados evoluíram muito no decorrer do tempo, e os modelos de sistemas simples e isolados foram substituídos gradativamente por complexos sistemas heterogêneos conectados, exigindo ferramentas de interoperabilidade, como a criação de Web Services. Em termos de B2B, temos uma grande utilização de Web Services do tipo SOAP, que trafegam dados no formato

A JSON

B XML

C YAML

D CSV

E RTF

A alternativa B está correta.

Atualmente, temos dois tipos principais de Web Services, que são o SOAP, voltado para B2B, utilizando texto no formato XML, e o REST, mais adotado no B2C, com tráfego de dados preferencialmente no formato JSON.

Questão 2

Utilizar acesso assíncrono aos recursos de entrada e saída, como arquivos ou rede, é mais que uma boa prática nos dispositivos móveis, tratando de uma necessidade para a correta operação do sistema. Para definir uma função assíncrona, devemos utilizar qual palavra reservada?

A Await

B Promise

C Async

D Result

E Then

A alternativa C está correta.

Uma função é configurada como assíncrona por meio da palavra reservada async. Para funções assíncronas, temos o retorno como um Promise tipado, e o valor retornado, do tipo especificado, é obtido por meio de await ou then.

Modelo relacional

Bases de dados relacionais seguem um modelo estrutural baseado na álgebra relacional e no cálculo relacional, áreas da matemática voltadas para a manipulação de conjuntos, apresentando ótimos resultados na implementação de consultas.

> Podemos dizer que um banco relacional é um repositório de dados capaz de manter relacionamentos consistentes entre os elementos armazenados.

Considerando a **álgebra relacional**, ela alia a teoria clássica de conjuntos, trazendo as operações básicas que podem ser feitas sobre eles, como união, interseção e subtração, além de operadores específicos para tipos de dados compostos, como projeção, seleção e junção. Os novos operadores são necessários para lidar com a representação padrão utilizada para os elementos do conjunto, conhecida como **tupla**, que pode ser definida como uma sequência finita de valores.

Podemos observar um pequeno exemplo, com dois conjuntos e uma operação de junção natural, nos quadros a seguir.

Alunos	
Alunos	Curso
Ana Claudia	Engenharia
Luiz Carlos	Medicina
Manoel de Oliveira	Engenharia

Quadro: Exemplo de conjuntos, suas tuplas, e operação de junção natural.

Cursos	
Curso	Coordenador
Engenharia	Rafael Santos
Medicina	Felipe Nascimento

Quadro: Exemplo de conjuntos, suas tuplas, e operação de junção natural.

Alunos | x | Cursos		
Aluno	Curso	Coordenador
Ana Claudia	Engenharia	Rafael Santos
Luiz Carlos	Medicina	Felipe Nascimento
Manoel de Oliveira	Engenharia	Rafael Santos

Quadro: Exemplo de conjuntos, suas tuplas, e operação de junção natural.

Seguindo procedimentos similares, no **cálculo relacional** temos a consulta aos dados baseada em expressões formais, além de incluir alguns termos novos, como "**Existe**" e "**Para Todo**". Para exemplificar, poderíamos

efetuar a consulta expressa pela sentença "obter todos os coordenadores de cursos em que existe ao menos um aluno inscrito".

```typescript
( (Curso) | (Aluno |X| Curso), Coordenador)
```

A partir do conhecimento acerca dos elementos matemáticos que fundamentam os bancos de dados relacionais, podemos compreender melhor as ferramentas de consulta e manipulação de dados, com base na sintaxe **SQL**. Embora eles não sejam o único tipo de banco de dados existente, já que temos opções multidimensionais e NoSQL, entre outras, são a escolha mais comum em termos cadastrais, pois permitem uma estruturação de dados muito organizada, com consultas ágeis, fundamentadas em procedimentos matemáticos já consolidados e amadurecidos ao longo de décadas.

Mesmo apresentando grande eficiência, a representação **matricial** dos dados, com cada **linha**, ou **registro**, representando uma **tupla**, e cada coluna representando um **campo**, não é a mais adequada para um ambiente de programação orientado a objetos, sendo necessário um procedimento de conversão chamado de **mapeamento objeto-relacional**, em que os **conjuntos de registros são expressos como coleções de objetos**, conforme será demonstrado posteriormente neste módulo.

Linguagem SQL

Para trabalharmos com bancos de dados relacionais, é necessário conhecer ao menos comandos básicos da sintaxe SQL, os quais podem ser divididos em três áreas:

- **DDL** (Data Definition Language).
- **DML** (Data Manipulation Language).
- **Seleção** ou **Consulta**.

Os comandos **DDL** são responsáveis pela criação das estruturas que receberão os dados e manterão os relacionamentos de forma consistente, tendo como elementos principais as tabelas e índices. Basicamente, utilizamos os comandos **CREATE, ALTER** e **DROP** como no exemplo seguinte, para a criação de uma tabela.

```sql
CREATE TABLE IF NOT EXISTS PRODUTO(
     CODIGO INTEGER PRIMARY KEY,
     NOME VARCHAR(20),
     QUANTIDADE INTEGER);
```

Aqui temos a criação da tabela chamada **PRODUTO**, com as colunas **CODIGO**, elemento numérico que funciona como identificador do registro, ou chave primária, **NOME**, do tipo texto, e **QUANTIDADE**, que também aceitará valores numéricos. Após a definição da estrutura de armazenamento, podemos utilizar os comandos **DML** para efetuar as operações de inclusão, alteração e exclusão de registros, o que é feito, respectivamente, por meio das instruções **INSERT, UPDATE** e **DELETE**.

Poderíamos criar um registro na tabela PRODUTO com o comando apresentado a seguir.

```
sql

INSERT INTO PRODUTO (CODIGO, NOME, QUANTIDADE)
 VALUES (1, 'Morango', 200);
```

Além das instruções utilizadas para definição e manipulação de dados, as consultas efetuadas com o comando **SELECT** talvez tenham o papel mais relevante em termos de SQL. Esse comando se divide, inicialmente, em duas partes principais, que são a **projeção** (campos) e a **restrição** (condições).

Podemos observar, a seguir, um exemplo de comando de seleção.

```
sql

SELECT NOME FROM PRODUTO
 WHERE CODIGO BETWEEN 1 AND 5;
```

No exemplo apresentado, temos uma consulta que retorna valor do campo **NOME** para cada registro presente na tabela **PRODUTO** (projeção), mas apenas para aqueles em que o campo **CODIGO** apresenta valores entre 1 e 5 (restrição). O comando **SELECT** é muito amplo e aceita elementos para ordenação, agrupamento, combinação e operações de conjunto, entre diversas outras possibilidades.

Alguns dos operadores aceitos pela SQL são apresentados no quadro seguinte.

Operador	Utilização
IN	Retorna os registros em que o valor de um determinado campo está presente em um conjunto de valores.
NOT IN	Retorna os registros em que o valor de um determinado campo não pode ser encontrado em um conjunto de valores.
LIKE	O valor do campo deve estar de acordo com um padrão, sendo tipicamente utilizado em situações do tipo "começado com".
EXISTS	Verifica uma condição de existência relacionada ao campo.
NOT EXISTS	Verifica uma condição de inexistência relacionada ao campo.
BETWEEN	Verifica se o valor se encontra entre dois limites.
ALL	Retorna o valor caso todos os elementos do conjunto satisfaçam à condição.
ANY	Retorna o valor caso algum elemento do conjunto satisfaça à condição.

Quadro: Principais operadores do SQL.

Além dos operadores apresentados, podemos utilizar a comparação tradicional, como os símbolos de menor que, igualdade e maior que. Também são permitidas combinações lógicas com o uso de **AND**, **OR** e **NOT**.

É possível agrupar campos com **GROUP BY**, e utilizar operações de sumarização, como **MAX**, **MIN** e **COUNT**, além da possibilidade de aplicar restrições aos grupos formados com o uso de **HAVING**. Não menos importante, podemos ordenar os resultados por quaisquer campos, de forma ascendente ou descendente, com o uso de **ORDER BY**.

Uma consulta SQL mais complexa é apresentada na listagem seguinte.

```
sql

SELECT CODIGO, NOME FROM PRODUTO
 WHERE QUANTIDADE IN (
     SELECT MAX(QUANTIDADE) FROM PRODUTO)
 AND NOME LIKE 'A%';
```

Segundo o que está expresso no exemplo, teremos como retorno o nome e o código dos produtos da base cuja quantidade seja equivalente ao máximo valor apresentado em toda a tabela e cujos nomes são iniciados com a letra "A".

Banco de dados SQLite

Quando falamos do **SQLite**, estamos tratando de um banco de dados relacional que não necessita de um servidor, como ocorre no Oracle, MySQL e outras ferramentas do tipo, permitindo **armazenamento local** de informações, como em um sistema de arquivos comum.

> O SQLite é uma ferramenta minimalista, de código aberto, gratuita e autossuficiente, muito utilizada em sistemas móveis, e que faz parte da arquitetura básica do Android.

SQLite.

Devido à sua estrutura leve, o SQLite oferece comportamento muito fluido, além de não precisar de nenhum tipo de configuração. Oferecido como uma biblioteca, tem versões para diversas linguagens de programação, o que facilita a padronização para atividades de persistência em sistemas com grande heterogeneidade e, como é baseado em arquivos simples, pode ser copiado de forma direta de um sistema para outro.

Pode-se dizer que a **segurança** não é uma preocupação do SQLite, pois não podem ser definidos usuários para acesso ao banco, ficando aberto para qualquer um que tenha acesso aos arquivos. Essa fragilidade é contornada, de forma geral, por meio da gerência de permissões do próprio sistema operacional, como ocorre na Sandbox do Android, apoiada no modelo permissivo do SE Linux.

[1] Construído com linguagem ANSI-C, o SQLite surgiu no ano 2000 e, apesar de exigir uma estrutura mínima, ocupando menos de 600 Kbytes, permite números grandiosos em termos de armazenamento, como tamanho de linha de até 1 Gigabyte e tamanho de banco máximo de 281 Terabyte, o que está muito acima das possibilidades físicas dos dispositivos móveis atuais.

[2] Com a viabilidade de grande espaço para armazenamento e acesso aos dados em velocidade que pode superar o uso direto do disco, além de um modelo físico que permite a cópia direta, o SQLite tem se tornado uma boa opção para o armazenamento local de recursos em jogos digitais.

[3]Algo importante a ser mencionado é que as transações oferecidas pelo banco SQLite possuem as propriedades ACID (atomicidade, consistência, isolamento e durabilidade), impedindo que os dados sejam corrompidos, mesmo com quedas de energia ou travamentos.

Podemos observar, a seguir, um exemplo de código JavaScript para acesso ao SQLite.

```
javascript

import { openDatabase } from 'react-native-sqlite-storage';

 var db = openDatabase({ name: 'LojaDatabase.db' });

 db.transaction(function(txn) {
     txn.executeSql('SELECT * FROM PRODUTO',[],
         function(tx,res) {
             console.log('registros:', res.rows.length);
         } ,
         function(tx,err) {
             console.log('erro:', err.message);
         }
     );
 });
```

No exemplo, temos a abertura do banco em um arquivo com o nome "**LojaDatabase.db**", pela função **openDatabase**. Com o banco criado, iniciamos uma transação, com o uso do método **transaction**, e utilizamos a transação para executar um comando SQL de consulta ao banco de dados.

A consulta é iniciada pelo método **executeSQL**, utilizando como parâmetros uma **instrução** em SQL, valores de **parâmetros** para a instrução, uma função **callback** para tratamento dos dados recebidos, e outra para o tratamento de erros de execução. No caso, efetuamos uma consulta na tabela **PRODUTO**, sem parâmetros, sendo exibido o total de registros retornados na primeira função de callback.

De forma geral, temos a necessidade da primeira callback apenas para o tratamento de dados recebidos, podendo ser omitida em instruções dos tipos **DDL** e **DML**.

Exemplo

Para a criação de tabelas, teríamos instruções CREATE TABLE, enquanto a inserção de um registro exigiria um comando INSERT, mas nos dois casos não temos a callback.

Ao final do código, podemos observar a segunda função de callback, que será utilizada para o tratamento de erros na execução do comando SQL. Nela, temos apenas a emissão da mensagem de erro, decorrente da execução, para o console.

Assim como outros bancos de dados, no SQLite temos algumas tabelas de **catálogo**, com dados estruturais, ou metadados, de todo o banco. Com o acesso ao catálogo, será possível definir comandos dinamicamente, consultar dados administrativos e verificar a existência ou o estado atual de qualquer estrutura do banco de dados.

O quadro apresentado a seguir descreve as tabelas de catálogo e seus campos.

Tabela	Campo	Descrição
sqlite_master	**type** **name** **tbl name** **rootpage** **sql**	Tipo de objeto, podendo ser table, index, trigger ou view. Nome do objeto. Nome da tabela à qual o objeto está associado. Número da página raiz. Instrução SQL para a criação do objeto.
sqlite_sequence	**name** **sql**	Nome da tabela associada ao campo de autoincremento. Último valor utilizado para o campo de autoincremento.
sqlite_stat1	**tbl** **idx** **stat**	Nome da tabela que foi analisada. Nome do índice que foi analisado. Informação retornada pelo otimizador de consultas.

Quadro: Catálogo do banco de dados SQLite.

Por meio de **sqlite_master**, obtemos os dados de qualquer estrutura do banco, incluindo nome, tipo e SQL de criação. Um campo interessante é **rootpage**, que identifica a árvore do tipo **b-tree** responsável pela organização dos dados da estrutura.

Atenção

Cada tabela ou índice do SQLite utiliza armazenagem em uma árvore do tipo b-tree, e o campo rootpage identifica qual b-tree foi utilizada para a estrutura.

Os campos do tipo autoincremento irão usar a tabela **sqlite_sequence** para guardar o último valor utilizado, constituindo um elemento essencial para o funcionamento do banco de dados.

Já o acesso aos dados de **sqlite_stat1** irá fornecer todos os resultados obtidos pelo otimizador de consultas, permitindo identificar fragilidades na definição de índices específicos e a eficiência da utilização em meio a comandos de seleção.

Como um exemplo simples, podemos obter os nomes de todas as tabelas do banco de dados por meio do comando apresentado a seguir.

```
sql

SELECT name FROM sqlite_master
     WHERE type='table';
```

Cadastro com SQLite

Para implementar nosso cadastro com uso de SQLite, iremos reaproveitar boa parte do código definido no primeiro módulo, fazendo apenas as adaptações necessárias.

> Nosso principal foco de mudanças será a classe GestorDados, uma vez que ocorrerá a mudança do modelo de armazenamento, mas a interface será mantida.

Inicialmente, vamos configurar nosso projeto, que executará no modelo nativo, com os comandos apresentados a seguir.

```
typescript

react-native init CadastroSQLite
 cd CadastroSQLite
 npm install react-native-sqlite-storage @types/react-native-sqlite-storage @types/react
--save
 npm install @react-navigation/native @react-navigation/stack
 npm install react-native-reanimated react-native-gesture-handler react-native-screens
react-native-safe-area-context @react-native-community/masked-view
```

Vamos aproveitar boa parte do exemplo anterior, com persistência baseada na gravação assíncrona em arquivos. Iniciaremos copiando, para o novo projeto, os arquivos **App.js**, **CommonStyles.js**, **ProdutoForm.js**, **ProdutoItem.js** e **ProdutoLista.js**, todos no diretório raiz, além de **Produto.ts**, no diretório **dados**.

Para definir a conexão com o banco de dados, vamos criar o arquivo **DatabaseInstance**, do tipo JavaScript, no diretório **dados**.

```
javascript

import { openDatabase } from 'react-native-sqlite-storage';

 var db = openDatabase({ name: 'LojaDatabase.db' });

 export default db;
```

1

No arquivo **DatabaseInstance.js**, teremos apenas o uso de openDatabase, definindo o nome do arquivo que será utilizado pelo **SQLite**, para obtenção da conexão na variável **db**, com a exportação ocorrendo ao final.

2

A conexão obtida será utilizada por todos os arquivos que precisarem executar comandos sobre o banco, como **GestorDados.ts**, que será responsável por concentrar as operações efetuadas sobre a tabela de produtos.

O arquivo **GestorDados**, do tipo TypeScript, deverá ser criado no diretório **dados**.

```typescript
import db from './DatabaseInstance';
 import { Produto } from './Produto';

 const sqlCreate =
        'CREATE TABLE IF NOT EXISTS PRODUTO('+
        'CODIGO INTEGER PRIMARY KEY, '+
        ' NOME VARCHAR(20), QUANTIDADE INTEGER)';
 const sqlInsert =
        'INSERT INTO PRODUTO ( CODIGO, NOME, QUANTIDADE )'+
        ' VALUES (?,?,?)';
 const sqlDelete =
        'DELETE FROM PRODUTO WHERE CODIGO=?';
 const sqlSelect =
        'SELECT * FROM PRODUTO';

 class GestorDados {
     private async criarBanco(){
     db.transaction(txn => txn.executeSql(sqlCreate,[]));
     }
     public async remover(chave: string){
     db.transaction(txn =>
             txn.executeSql(sqlDelete,[parseInt(chave)]));
     }
     public async adicionar(produto: Produto){
     db.transaction(txn => txn.executeSql(sqlInsert,
             [produto.codigo,produto.nome,produto.quantidade]));
     }
     public async obterTodos(
             useRetorno: (produtos: Array) => void){
     let objetos = [];
     db.transaction((txn) => txn.executeSql(sqlSelect,[],
         (txn, results) => {
         for (let i = 0; i < results.rows.length; ++i) {
             let linha = results.rows.item(i)
             let produto: Produto = new Produto(
                 linha.CODIGO, linha.NOME, linha.QUANTIDADE);
             objetos.push(produto);
         }
         useRetorno(objetos);
         if(objetos.length<1)
             this.criarBanco();
         }));
     }
 }

 export default GestorDados;
```

Analisando a listagem, podemos observar que a conexão **db** é importada, assim como a classe **Produto**, e temos a definição dos comandos **SQL** em algumas variáveis de texto nas linhas seguintes. Temos as variáveis:

sqlCreate

Para a criação da tabela.

sqlSelect

Para a seleção de todos os produtos.

sqlInsert

Para a inserção de um registro.

sqlDelete

Para a remoção do registro.

Os comandos SQL definidos são utilizados pela classe **GestorDados**, implementada logo após esses comandos. Essa classe é semelhante à que foi utilizada no primeiro exemplo, mas agora conta com a persistência sendo efetuada no SQLite.

Comentário

Além dos métodos públicos para consulta, remoção e inserção que irão complementar as funcionalidades dos demais arquivos, temos na classe GestorDados o método privado criarBanco, que executará o comando definido em sqlCreate, gerando a tabela.

Podemos observar a presença de algumas **interrogações** (?) nos comandos para inserção e remoção de registros, representando **parâmetros** que devem ser preenchidos a partir da chamada para execução. Veja a seguir a descrição dos valores dos parâmetros de cada método.

remover

No método **remover**, temos o preenchimento do parâmetro com o valor da **chave**.

adicionar

No método **adicionar**, os parâmetros são preenchidos com os valores de um vetor que contém os atributos de **produto**.

obterTodos

O método mais complexo é **obterTodos**, que deverá receber uma função, com o nome **useRetorno**, na qual teremos como parâmetro um vetor de produtos.

Na verdade, o que estamos realizando é a definição de uma **callback**, ou seja, uma função que deverá ser executada ao final de determinado procedimento, algo muito útil quando trabalhamos com processamento **assíncrono**.

No corpo do método, temos a criação de um vetor de produtos vazio, o qual deverá ser preenchido a partir da consulta efetuada na linha seguinte. Com o retorno da consulta em **results**, temos um loop que percorre todas as linhas, e para cada linha adicionamos um objeto do tipo **Produto**, com os dados da linha, ao vetor. Por fim, a

função **callback** é invocada, com a passagem do vetor preenchido e, se não tivermos nenhum produto no vetor, ocorre a chamada para **criarBanco**.

Atenção

Todos os comandos SQL são invocados pelo método executeSql, sempre dentro de transações, definidas por meio de transaction.

O arquivo **ProdutoLista.js** precisará ser modificado, para se adequar ao padrão utilizado nas consultas baseadas em SQLite.

A seguir, é apresentado o início do código para a classe **ProdutoLista**, contemplando as modificações citadas.

```
typescript

export default function ProdutoLista( { navigation } ) {
     const gestor = new GestorDados();
     const [produtos, setProdutos] = useState([]);

     const isFocused = useIsFocused();

     useEffect(() => {
         gestor.obterTodos((objs)=>setProdutos(objs));
     }, [isFocused]);

     const myKeyExtractor = item => {
         return item.codigo.toString();
     };

     function excluirProduto(codigo){
         gestor.remover(codigo).then(
             gestor.obterTodos((objs) => setProdutos(objs))
         );
     }
```

O que temos na prática é a modificação da chamada para o método **obterTodos**, no qual agora, em vez de adotar o operador **then**, fornecemos uma função que utilizará o vetor obtido em uma chamada para **setProdutos**. Devemos alterar a chamada para **useEffect** e o corpo do método **excluirProduto**, mantendo todo o restante do código original.

> Note que, como uma boa prática, evitamos trabalhar diretamente com o resultado da consulta aos dados, em que é adotado o modelo relacional. Em vez de trabalharmos com o resultado diretamente, utilizamos uma coleção de objetos do tipo Produto, na qual cada um reflete as informações de um registro.

Atenção

A conversão de tabelas e respectivos registros em coleções de entidades é uma técnica conhecida como mapeamento objeto-relacional.

Feitas as inserções e as alterações necessárias nos arquivos do projeto, podemos executar o aplicativo e verificar que, mesmo com a persistência modificada, a interface original do primeiro exemplo é mantida, trazendo todas as funcionalidades anteriores. Temos a impressão de estar utilizando o mesmo aplicativo, mas devemos ter em mente que os dados passaram a ser armazenados em uma base de dados do tipo SQLite.

Armazenagem com SQLite

No vídeo a seguir, apresentamos uma classe de persistência, com base em SQLite, contemplando as operações de inclusão, alteração, exclusão e consulta aos dados do banco.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

O uso de comandos SQL dispersos, em meio ao código do aplicativo, diminui o reuso e aumenta a dificuldade de manutenção. Com a utilização de um gestor de dados, temos a concentração dos comandos SQL em uma única classe, em que existem métodos para o retorno de entidades, como obterTodos, que estaria relacionado ao comando

A INSERT.

B CREATE.

C DELETE.

D UPDATE.

E SELECT.

A alternativa E está correta.

Os gestores de dados seguem o padrão em que precisamos minimamente dos métodos obter Todos, incluir e excluir, os quais estarão relacionados, respectivamente, aos comandos SELECT, INSERT e DELETE. Com base nesses métodos, temos a possibilidade de listar os registros, acrescentar um novo registro, ou remover um registro da base de dados.

Questão 2

O banco de dados SQLite é adotado como padrão de armazenagem no Android, e sua utilização, no ambiente do React Native, é muito simples. Para efetuar a inserção de um registro na base, qual método deve ser executado, quando utilizamos React Native?

A executeSql

B executeInsert

C transaction

D openDatabase

E executeDelete

A alternativa A está correta.

Para efetuar a inclusão de um registro no banco de dados, devemos utilizar o método executeSql, com a passagem de um comando SQL do tipo INSERT. De modo geral, o método precisa estar em uma transação, iniciada por meio de transaction, a partir de uma conexão com o banco já estabelecida, com base no método openDatabase.

Banco de dados orientado a objetos

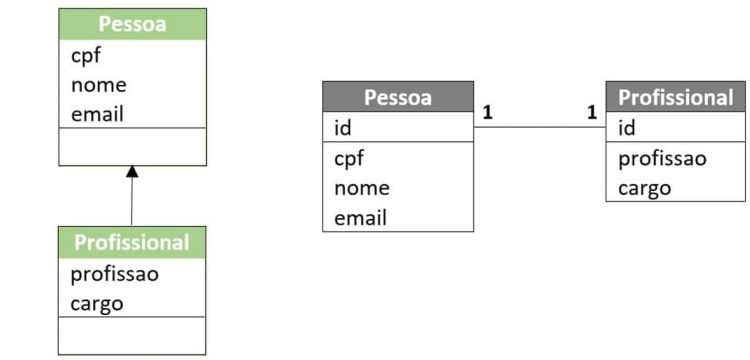
Atualmente, o desenvolvimento de sistemas por meio de linguagens orientadas a objetos é um padrão de mercado bem aceito pelas mais diversas empresas e consultorias, mas os bancos de dados mostram uma predominância do modelo relacional. Tal cenário traz uma boa justificativa para a adoção de técnicas de **mapeamento objeto-relacional**, mas a possibilidade de utilizar uma base de dados orientada a objetos simplificaria muito o processo de transferência dos dados para as estruturas da linguagem.

Um **OODBMS**, ou **sistema de banco de dados orientado a objetos**, permite representar estruturas complexas de uma forma muito mais eficaz que os bancos relacionais. Enquanto temos uma representação focada na arquitetura de armazenamento para os relacionais, nos bancos que utilizam objetos tiramos proveito dos diversos conceitos oferecidos pela metodologia orientada a objetos, incluindo princípios como **herança** e **polimorfismo**.

> Com os princípios da orientação a objetos, um OODBMS consegue utilizar modelos próximos do mundo real.

Exemplo

Seria possível definir uma classe Profissional a partir de Pessoa, aproveitando todas as características da classe original e mantendo a semântica, já que um profissional é uma pessoa com alguns atributos a mais, enquanto no modelo relacional teríamos duas tabelas, em que Pessoa seria uma entidade forte e Profissional a entidade fraca, dentro de um relacionamento do tipo um para um, algo que demonstra uma preocupação maior com o armazenamento dos dados que com a representatividade da forma em si.



Classes na orientação a objetos e tabelas no modelo relacional.

Utilizando ponteiros no lugar de junções, o modelo orientado a objetos permite uma execução bem mais rápida que no banco relacional, já que o acesso aos dados é baseado no simples encadeamento de ponteiros. Além da agilidade no acesso, a habilidade de lidar com grandes massas de dados é uma característica importante de um OODBMS, como no uso de **Objectivity/DB** pelo Stanford Linear Accelerator Center (SLAC), que, em 2015, já acumulava cerca de 900 terabytes de dados, como resultado dos experimentos com aceleração de partículas.

Comentário

Enquanto apresenta diversas vantagens, um OODBMS também traz algumas desvantagens, como a inexistência de um modelo universal de dados e a falta de uma linguagem de consulta padronizada. Essas são características que estão presentes no modelo relacional, por meio das tabelas, com tipos de campos bem definidos, e da sintaxe oferecida pela linguagem SQL.Outras desvantagens, frente ao modelo relacional, são a impossibilidade de executar consultas aninhadas e a ausência de princípios matemáticos robustos, como a álgebra relacional e o cálculo relacional.

História dos bancos de dados orientados a objeto

O termo **banco de dados orientado a objetos** surgiu em torno de 1985, representando o resultado de pesquisas efetuadas por empresas como Bell Labs, Texas Instruments e Hewllet-Packard, entre várias outras. Surgiram muitos produtos no mercado a partir daí, como GemStone, Versant, Matisse e Objectivity/DB.

Em 1991, foi criado um consórcio de fornecedores de **OODBMS**, iniciado por Rick Cattell, da Sun Microsystems, com a sigla **ODMG** (*Object Database Management Group*), que tinha como objetivo a definição de padrões. Esses padrões incluíram uma linguagem para a definição de objetos **ODL** (*Object Definition Language*), um padrão para intercâmbio de informações **OIF** (*Object Interchange Format*) e uma sintaxe declarativa para consulta aos objetos **OQL** (*Object Query Language*).

A versão 3.0 da especificação foi publicada no ano 2000, e a maior parte dos bancos de dados orientados a objetos e das ferramentas para mapeamento objeto-relacional buscaram a conformidade com os padrões.

Foram definidos conectores para várias linguagens, incluindo C++, Smalltalk e Java, mas, em 2001, o **ODMG** *Java Language Binding* foi submetido ao Java Community Process como base para a especificação **JDO** (*Java Data Objects*). As empresas participantes do ODMG decidiram, então, concentrar seus esforços na especificação JDO, terminando o consórcio em 2001 e interrompendo a busca por um padrão global de mercado.

Comentário

Ao longo do tempo, surgiram diversas opções de OODBMS, como pode ser observado no ranking de bancos de dados do mercado (DB-engines) sugerido na seção Explore +.Para nossos exemplos, utilizaremos o Realm como opção de banco de dados orientado a objetos, pois o seu acesso a partir do React Native é bastante simplificado. Embora também seja reconhecido como um banco de dados NoSQL baseado em documentos, possui características próprias do modelo orientado a objetos.

Banco de dados Realm

O banco **Realm** é uma alternativa de código aberto ao armazenamento com SQLite, com uma abordagem mais amigável para o desenvolvedor, uma vez que trabalha dentro do padrão orientado a objetos. É uma boa opção para as plataformas móveis, com versões para as linguagens Swift, Objective-C, Java, Kotlin, C# e JavaScript.

Sua utilização é muito intuitiva, seguindo a codificação padrão da linguagem, o que torna a curva de aprendizagem muito menor, além de não necessitar de processos voltados para o mapeamento objeto-

relacional, diminuindo a quantidade total de código. Todos os dados são observados como objetos e coleções, sendo necessário apenas instanciar um gestor para o acesso ao banco.

Entre as características mais relevantes do Realm, destacam-se:

- Ser uma **plataforma leve**, totalmente funcional e independente.
- Ter um **baixo consumo** de memória.
- Utilizar **pouco espaço** em disco.

Isso faz com que se torne uma plataforma adequada para trabalhar offline, motivo pelo qual foi adotado em diversas plataformas móveis, o que não impede seu uso em sistemas desktop ou cliente-servidor.

A listagem seguinte, que deverá ser salva no arquivo "**DatabaseInstance.js**", apresenta o código JavaScript para instanciar um banco de dados Realm no React Native.

```
typescript

import Realm from 'realm';

 var db = new Realm({
     path: 'ProdutosDB.realm',
     schema: [
         {
             name: 'Produto',
             primaryKey: 'codigo',
             properties: {
                 codigo: 'int', nome: 'string', quantidade: 'int',
         },
         },
     ],
     });

 export default db;
```

O uso de um banco de dados exige a definição de um esquema estrutural, com as classes que serão utilizadas, definindo seus atributos, com respectivos tipos, e assinalamento do campo de identificação, ou chave primária. No exemplo, definimos apenas a estrutura para **Produto**, trazendo os campos **código**, **nome** e **quantidade**, junto a seus respectivos tipos, além da seleção de código para a chave primária.

Após a importação da classe necessária, a partir da biblioteca **realm**, criamos a instância do gestor, com o nome **db**, fornecendo o nome do arquivo em **path** e a estrutura do banco por meio de **schema**. Para o esquema de banco, temos um vetor de classes para os metadados, em que cada classe apresenta o nome (**name**), chave primária (**primaryKey**) e as propriedades (**properties**), com seus respectivos tipos.

Com a exportação do gestor, ele pode ser utilizado em qualquer código JavaScript ou TypeScript, a partir da importação do arquivo.

```
javascript

for(let obj of db.objects('Produto')
                 .filtered('codigo > $0', 100)){
     console.log(obj.codigo+'::'+obj.nome);
 }
```

No exemplo, temos uma consulta aos produtos da base, invocando **objects**, com uma filtragem subsequente para todos os códigos maiores que 100, por meio de **filtered**, e para acessar os atributos de cada objeto,

temos a tipagem para Produto. O retorno será do tipo **Results< Produto>**, permitindo que seja utilizado um loop, no qual percorremos todos os resultados e efetuamos a impressão do **código** e **nome** de cada produto.

As operações de inclusão, alteração e exclusão, equivalentes aos comandos DML do modelo relacional, também seguem procedimentos simples, como pode ser observado na listagem apresentada a seguir.

```
typescript

let produto = new Produto(1,'Morango',200);
 db.write(() => db.create('Produto', produto));
```

Note que o processo é iniciado com a definição de um trecho transacional, para **escrita** dos dados no banco, pela chamada para **write**, que é aplicado ao conjunto de comandos no interior da função anônima; e por estar em uma **transação**, o bloco inteiro é confirmado ao final, ou cancelado quando um erro ocorre. Em nosso exemplo, temos apenas a inclusão do produto instanciado na linha anterior, invocando o comando **create**, que tem como parâmetros o nome da coleção e o objeto.

Os métodos da classe Realm podem ser observados no quadro seguinte.

Método	Utilização
open	Abre uma conexão com o banco, de forma equivalente ao construtor da classe.
close	Fecha a conexão com o banco de dados.
write	Define um bloco de escrita no banco.
create	Adiciona o objeto a uma coleção do banco.
delete	Remove o objeto de uma coleção do banco.
objects	Retorna a coleção de objetos a partir de seu nome no banco.
objectForPrimayKey	Retorna o objeto de uma coleção do banco a partir do valor de sua chave primária.
beginTransaction	Inicia uma transação no banco de dados.
commitTransaction	Confirma a transação, efetivando as modificações efetuadas.
cancelTransaction	Cancela a transação, desfazendo todas as alterações.

Quadro: Métodos da classe Realm.

> Algo que podemos observar é a ausência de um método para alteração, o que ocorre por não ser necessário.

Para alterar o valor de um determinado campo, basta iniciar o bloco de escrita, selecionar o objeto e alterar o valor do atributo desejado, causando a escrita direta no banco de dados.

```typescript
db.write(() => {
     let obj = db.objectForPrimaryKey('Produto',1002);
     obj.nome = 'XPTO';
     obj.quantidade = 1200;
 });
```

Na listagem de exemplo, temos a obtenção do produto a partir da chave primária, com valor 1002, seguido da alteração dos valores do nome e da quantidade. Como temos um bloco transacional de escrita, as modificações efetuadas são imediatamente refletidas no banco, ao final da execução do bloco.

Da mesma forma que temos o relacionamento entre tabelas no modelo relacional, as classes também se relacionam, mas no lugar de índices utilizamos coleções para definir esses relacionamentos, como pode ser observado a seguir, na definição de um esquema de classes mais complexo.

```typescript
const CarroSchema = {
     name: 'Carro',
     properties: {
         marca: 'string', modelo: 'string',
         milhas: {type: 'int', default: 0},
     }};
 const PessoaSchema = {
     name: 'Pessoa',
     properties: {
         nome: 'string', fotografia: 'data?'
         carros: 'Carro[]', // lista de carros
     }};
 var db = new Realm({
         path: 'EstacionamentoDB.realm',
         schema: [CarroSchema, PessoaSchema]
     });
```

Temos a definição de dois elementos de esquema no código, sendo:

1. O primeiro para a classe **Carro**, contendo os atributos **marca**, **modelo** e **milhas**.
2. O segundo para a classe **Pessoa**, com os atributos **nome**, **fotografia** e **carros**.

Observe que carros é uma **coleção** de objetos da classe Carro, além da definição de um valor **padrão** para o campo milhas via **default**, e a utilização de um campo **opcional**, com o nome fotografia, algo que é definido pelo uso da **interrogação** no tipo do atributo.

Comentário

A criação do banco engloba os dois esquemas de classe, o que significa que teremos duas coleções, uma para carros e outra para pessoas, sendo possível definir quais carros pertencem a uma determinada pessoa, a partir da coleção expressa no atributo.

Para efetuar a inclusão de um conjunto de objetos no banco de dados, basta instanciá-los em um bloco transacional de escrita.

```
typescript

db.write(() => {
     let john = db.create('Pessoa', {nome: 'John', carros: []});
     john.carros.push({marca: 'Honda',  modelo: 'Accord',
                         milhas: 1500});
     john.carros.push({marca: 'Toyota', modelo: 'Prius',
                         milhas: 2780});

     let joan = db.create('Pessoa', {nome: 'Joan', carros: []});
     joan.carros.push({marca: 'Skoda',  modelo: 'Octavia',
                         milhas: 1120});
     joan.carros.push({marca: 'Ford',   modelo: 'Fiesta',
                         milhas: 95});
     joan.carros.push({marca: 'VW',     modelo: 'Golf',
                         milhas: 1270});
 });
```

Podemos observar que, após a criação de uma pessoa no Realm, utilizamos o método push da coleção de carros, em cada pessoa criada, para incluir novos objetos do tipo Carro diretamente no banco de dados, já associados à coleção de carros daquela pessoa.

> O controle transacional é uma funcionalidade que permite o isolamento de um conjunto de operações, garantindo a consistência na execução do processo completo.

De forma geral, ele ocorre da seguinte maneira:

- Uma transação é iniciada.
- As operações são executadas.
- O bloco é confirmado com o uso de **commit**, ou a reversão das operações, com **rollback**.

Com o uso de **transações**, temos a garantia de que o sistema irá desfazer as operações anteriores à ocorrência de um erro, como na inclusão dos itens de uma nota fiscal. Sem o uso de uma transação, caso ocorresse um erro na inclusão de algum item, seríamos obrigados a desfazer as inclusões que ocorreram antes do erro, de forma programática, mas com a transação será necessário apenas emitir um comando de rollback.

Se quisermos controlar a transação fora de um bloco de escrita, precisamos chamar os métodos de controle transacional da classe Realm.

```
typescript

db.beginTransaction();
 try {
     db.create('Pessoa', {nome: 'Arthur', carros: []});
     db.create('Pessoa', {nome: 'Rafael', carros: [
     {marca: 'Toyota', modelo: 'Prius', milhas: 1500}
     ]});
     db.commitTransaction();
 } catch (e) {
     db.cancelTransaction();
     throw e;
 }
```

No exemplo de código, temos o início da transação por meio do uso de **beginTransaction**, seguido de um bloco **try**, em que são incluídas duas pessoas e respectivos carros, com a confirmação da transação por meio de **commitTransaction**. Caso ocorra algum erro no processo, o fluxo é desviado para o bloco **catch**, no qual qualquer alteração pendente será desfeita com a utilização de **cancelTransaction**, equivalente a um comando rollback.

Cadastro baseado no Realm

Após compreender as funcionalidades do Realm, podemos utilizá-lo na persistência de produtos em mais um exemplo. Vamos iniciar com a criação e configuração do projeto, com base nos comandos apresentados a seguir.

```
typescript

react-native init CadastroRealm
 cd CadastroRealm
 npm install realm --save
 npm install @react-navigation/native @react-navigation/stack
 npm install react-native-reanimated react-native-gesture-handler react-native-screens
react-native-safe-area-context @react-native-community/masked-view
```

Tendo configurado o projeto, vamos copiar os arquivos que serão reutilizados de nosso primeiro exemplo, com o nome CadastroLocal. Os arquivos que precisamos são **App**, **CommonStyles**, **ProdutoForm**, **ProdutoItem** e **ProdutoLista**, todos do tipo JavaScript, além do arquivo **Produto**, do tipo TypeScript, posicionado no diretório **dados**.

Agora vamos adicionar o arquivo **DatabaseInstance.js** no diretório **dados**, para definir a conexão com o banco de dados, de acordo com a listagem seguinte.

```typescript
import Realm from 'realm';

 var db = new Realm({
     path: 'ProdutosDB.realm',
     schema: [
         {
         name: 'Produto',
         primaryKey: 'codigo',
         properties: {
             codigo: 'int',
             nome: 'string',
             quantidade: 'int'
         }
         }
     ]
     });

 export default db;
```

Tudo que precisamos é um objeto do tipo **Realm**, contendo o caminho para o banco de dados no atributo **path**, e o conjunto de entidades englobadas em **schema**. Para nosso exemplo, temos apenas a entidade Produto, com a definição dos campos código, nome e quantidade, por meio de properties, além da escolha da chave primária, no atributo primaryKey, como sendo o campo código.

No passo seguinte, devemos acrescentar o arquivo GestorDados.ts, no diretório dados, contendo o código apresentado a seguir.

```typescript
import db from './DatabaseInstance';
 import { Produto } from './Produto';

 class GestorDados {
     public async remover(chave: string){
     db.write(()=>
         db.delete(db.objects('Produto')
                     .filtered('codigo = $0', parseInt(chave)))
     );
     }
     public async adicionar(produto: Produto){
     db.write(() => db.create('Produto', produto));
     }
     public async obterTodos(): Promise>{
     let objetos = [];
     for(let obj of db.objects('Produto')){
         objetos.push(JSON.parse(JSON.stringify(obj)));
     }
     return objetos;
     }
 }

 export default GestorDados;
```

Aqui teremos a gerência das operações para consulta e manipulação dos produtos em nossa base de dados, por intermédio do objeto **db**, exportado anteriormente. Como visto nos exemplos anteriores, precisamos

definir os métodos **remover**, **adicionar** e **obterTodos**, todos eles assíncronos, de modo a completar a funcionalidade dos arquivos reutilizados.

O método **adicionar** é o mais simples, no qual temos apenas a utilização de **create**, dentro de uma transação de escrita, para o acréscimo do produto na coleção. Já para o método **remover**, inicialmente encontramos o produto a partir da chave, com o uso de **filtered**, e na sequência temos a remoção do objeto recuperado, com o **delete**, dentro de um fluxo que também ocorre em uma transação.

Para a consulta, não necessitamos de transação, e o que temos é a recuperação de todos os objetos da coleção, devendo ser observada a tipagem para **Produto**. Ao contrário do que pensaríamos inicialmente, não obtemos uma coleção de objetos Produto, mas sim um **Realm.Results** contendo objetos do tipo **Realm.Object**, trazendo os atributos da classe Produto, o que exige a conversão de todos os elementos para Produto, por meio de funções de manipulação **JSON**, antes de adicionar ao vetor de retorno do método.

Ao final, a classe **GestorDados** é exportada, completando a funcionalidade exigida pelos demais arquivos do projeto, e já podemos executar nosso aplicativo, agora, com uso de persistência baseada em **Realm**.

Armazenagem com Realm

No vídeo a seguir, apresentamos uma classe de persistência, com base em Realm, contemplando as operações de inclusão, alteração, exclusão e consulta aos dados do banco.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando a aprendizagem

Questão 1

Uma das vantagens dos bancos de dados orientados a objetos é a grande similaridade com as estruturas de dados das atuais linguagens de programação, tornando o processo de mapeamento objeto-relacional desnecessário. Qual o nome do consórcio, criado no ano de 1991 por Rick Cattell, voltado para os bancos de dados orientados a objetos?

A OQL

B ODMG

C OIF

D ODL

E OODMBS

A alternativa B está correta.

Em 1991, foi criado o consórcio ODMG (Object Database Management Group), que tinha como objetivo a definição de padrões para OODBMS (Servidores de Banco de Dados Orientados a Objetos), incluindo a linguagem para a definição de objetos (ODL), o padrão para intercâmbio de informações (OIF) e a sintaxe declarativa para consulta (OQL).

Questão 2

As transações são essenciais para a garantia da consistência nos bancos de dados em operações múltiplas, independentemente do modelo utilizado para armazenagem. No caso do Realm, uma transação pode ser iniciada pelo método

A Transaction.

B commitTransaction.

C beginTransaction.

D rollback.

E cancelTransaction

A alternativa C está correta.

Na utilização de um banco de dados Realm, podemos iniciar uma transação com o método **beginTransaction**. Com a transação iniciada, efetuamos as operações desejadas sobre o banco, confirmando o conjunto por meio de commitTransaction, ou desfazendo as alterações, com o uso de cancelTransaction.

Not Only SQL

Veja, a seguir, como o termo **NoSQL** se transformou em Not **Only SQL**:

Início

O termo **NoSQL** inicialmente era utilizado para definir bases de dados não relacionais, como uma negação às consultas via sintaxe SQL.

Evolução

O conceito evoluiu, incorporando maior heterogeneidade nos modelos de armazenamento.

Atual

Houve, então, a mudança de interpretação da sigla original, que passou a significar **Not Only SQL**, ou seja, bancos de dados que não utilizam apenas SQL.

Com o surgimento do Big Data, tivemos um aumento exponencial do uso de clusters no armazenamento, o que pode se tornar um problema quando precisamos nos preocupar com a manutenção de relacionamentos entre entidades. A abordagem NoSQL busca uma forma de armazenamento mais plana, facilitando a distribuição de dados e priorizando a escrita única e leitura múltipla, sem a preocupação com alterações.

> Quando utilizamos o armazenamento NoSQL, estamos comprometendo a consistência em favor da disponibilidade, da velocidade, da escalabilidade e da tolerância ao particionamento.

Boa parte das opções de mercado não oferece suporte às transações, com as alterações sendo feitas apenas de forma local e sendo repercutidas para os demais nós do cluster posteriormente, após um breve período, normalmente na ordem de milissegundos.

Os bancos de dados NoSQL não apresentam apenas uma estratégia arquitetural, mas todas perseguem os mesmos objetivos. Entre as opções existentes, podemos destacar as estratégias chave-valor, colunar, baseada em grafo e documental, além de algumas arquiteturas híbridas, que adotam combinações das anteriores.

Atenção

Enquanto no modelo relacional temos esquemas fixos para armazenagem, as estruturas de dados podem ser alteradas dinamicamente em bases NoSQL.

Uma arquitetura de armazenagem do tipo chave-valor é muito simples, contando com linhas em que temos um identificador, ou chave, associado a um valor, com a intepretação do valor sendo efetuada nos aplicativos. Mesmo com um modelo tão simples, podem ser incluídas funcionalidades avançadas, como as transações e o versionamento de registro, presentes no **Oracle Berkeley DB**.

Atenção

Como as bases NoSQL são voltadas para inclusão massiva, sem permitir alterações ou exclusões de valores, o versionamento do registro acaba se tornando uma necessidade para alguns tipos de aplicações.

Veja a seguir algumas observações sobre o modelo colunar:

1

O **modelo colunar** pode ser considerado como uma extensão da arquitetura chave-valor, pois o que temos é a divisão do valor em **setores**, ou **colunas**.

2

As colunas, por sua vez, são agrupadas em **famílias**, sendo possível acrescentar novas famílias e colunas de forma dinâmica, o que traz uma flexibilidade muito maior que a do modelo relacional.

Um banco de dados colunar amplamente utilizado no mercado é o **HBase**, que possibilita a armazenagem de dados no formato texto ou sequências binárias, com processamento distribuído, sendo oferecido como um produto de código aberto. Ele foi construído com a linguagem Java, sendo modelado a partir do **Google Big Table**, e oferece integração fácil com o ambiente Hadoop, viabilizando consultas por meio de operações **MapReduce**.

Podemos observar, a seguir, um exemplo de representação de dados no HBase.

	Família aluno		Família turma		Família imagens
	aluno:matrícula	**aluno:nome**	**turma:codigo**	**turma:professor**	**imagens:foto**
1	20181.1873223	Luiz Carlos	ENG2010001	José Oliveira	HzvMg4da12...
2	20181.2341581	Ana Lima	ECO2010012	Thiago Abreu	HzfxHb5kPr...
3	20192.0923127	Paulo Silva	ENG2010001	José Oliveira	HzAtyKmcv5...

Tabela: Fragmento de dados no HBase.

No fragmento de dados exposto, temos três famílias de colunas, uma com os dados do aluno, outra da turma, e finalmente uma foto do aluno.

Toda linha é definida por uma chave numérica e, no cruzamento entre a linha e a coluna, temos uma célula, com o valor do campo e um **timestamp** oculto, adicionado de forma automática, para viabilizar o controle de versão de forma transparente.

> Ao contrário do modelo relacional, em que há grande preocupação com a normalização, nas bases NoSQL serão verificados muitos dados duplicados, facilitando a distribuição do processamento.

Também é interessante observar que o campo foto, embora seja um atributo do aluno, está em uma família própria, algo que costuma ser adotado como uma técnica para agilizar as pesquisas, uma vez que a informação binária ocupa muito mais espaço que o texto comum e não aceita restrições na consulta.

Quando utilizamos a arquitetura documental, temos sempre uma chave associada a um documento, o qual pode ser uma notação de texto, como **XML**, **JSON** e **YAML**, ou algum formato binário, como DOC e PDF. Uma

característica dos documentos armazenados é trazer alguma estrutura de metadados associada, motivo pelo qual os dados são classificados como **semiestruturados**.

Exemplo

Um dos melhores exemplos de banco de dados no modelo documental é o MongoDB, uma opção de código aberto, multiplataforma e escrita em linguagem C++. Os valores armazenados no MongoDB são documentos no formato JSON, e as chaves podem ser geradas de forma automática ou especificadas de acordo com padrões específicos.

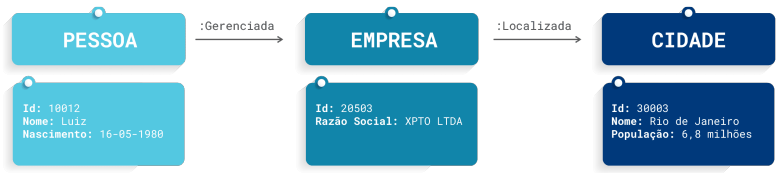
Nos bancos de dados baseados em grafos, temos os dados representados como **vértices** e as ligações entre eles definidas como **arestas**, seguindo a interpretação tradicional dos grafos utilizados na computação. Apesar de permitir expressar o relacionamento entre dois elementos de dados, as ligações efetuadas são muito mais flexíveis que as chaves estrangeiras adotadas no modelo relacional.

Os nós, ou vértices, sempre apresentam um identificador e um valor associado, como ocorre no modelo tradicional chave-valor, mas podemos definir arestas, com a ligação de dois identificadores, gerando toda uma rede de dados conectados.

Exemplo

Um bom exemplo de banco de dados baseado em grafos é o Neo4J, desenvolvido na linguagem Java, que tem código aberto e oferece suporte a transações.

Na imagem seguinte, é possível observar dados armazenados no modelo de grafo.



Dados no modelo de grafo.

Temos muitas opções de bancos de dados NoSQL no mercado, sempre amparados nos modelos aqui descritos, e alguns são enquadrados em mais de uma categoria, sendo chamados de **híbridos** ou **multimodelos**.

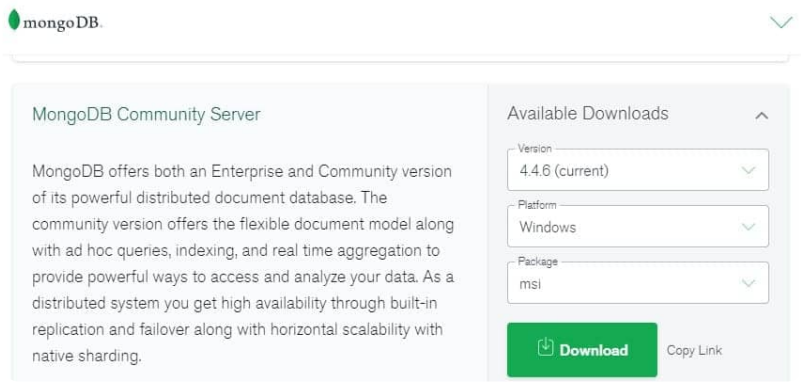
Dica

Uma lista dos principais bancos de dados NoSQL pode ser conferida no ranking (DB-ranking) sugerido na seção Explore +.

Banco de dados MongoDB

Trabalhando com um modelo orientado a documentos, **MongoDB** é baseado no formato **JSON**, permitindo a modelagem de dados complexos, com hierarquias expressas por meio de campos aninhados. O uso de uma notação natural do JavaScript facilita a **indexação** e a **busca**, sendo possível utilizar consultas por campo, faixa de valores ou até mesmo **expressões regulares**, além de oferecer recursos para **amostragem** aleatória.

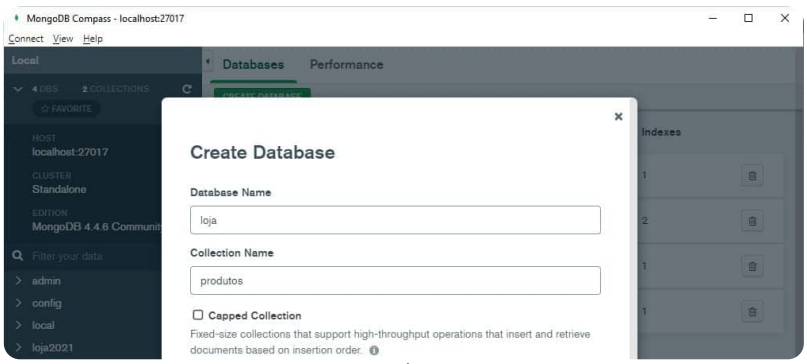
Para trabalharmos com o banco de dados MongoDB, devemos baixar e instalar a versão gratuita no endereço https://www.mongodb.com/ try/download/community. Escolha a versão adequada para seu sistema operacional e clique em download.



Download do MongoDB.

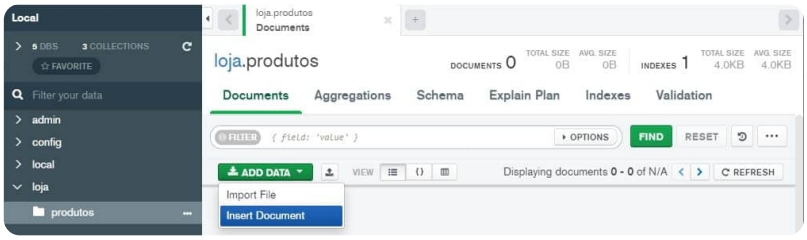
Ao final da instalação, teremos o banco de dados instalado como um **serviço**, além de um aplicativo para gerência com o nome **MongoDB Compass**. Considerando o peso do MongoDB, é interessante tirar o serviço do modo de inicialização automática e ativar apenas quando for necessário utilizá-lo.

Com o serviço ativo, vamos abrir o gerenciador e clicar diretamente na opção **Connect**, abrindo uma conexão com o servidor de forma local. Efetuada a conexão, vamos criar um banco de dados com o nome "**loja**", contendo uma coleção denominada "**produtos**", mas sem marcar a opção **Capped Collection**, pois isso impediria a remoção de registros.



Criação de banco no MongoDB por intermédio do Compass.

Agora já podemos inserir alguns registros, ou documentos, no formato JSON, navegando até o banco criado, na lateral esquerda, e escolhendo a coleção definida. Com a coleção aberta, escolhemos a opção **Add Data**, seguido de **Insert Document**, que resulta na abertura uma janela para a digitação do documento.



Inserção de registro por meio do Compass.

A **única restrição** acerca dos documentos criados no MongoDB é a necessidade de um campo identificador com o nome _id, funcionando como chave primária, que pode ser um número inteiro definido na inserção, ou um identificador gerado automaticamente, com tamanho de 12 bytes, sendo composto por um timestamp de 4 bytes, um número randômico de 5 bytes e um contador de 3 bytes.

```
typescript

{
     '_id': 1,
     'nome': 'Banana',
     'quantidade': 2000
 }
```

Após inserir alguns documentos, podemos efetuar consultas, digitando as expressões na barra de pesquisa, e clicando no botão **Find**. Temos diversos operadores disponíveis na sintaxe de consulta do MongoDB, como podemos observar no quadro seguinte.

Operador	Utilização
$gt	Retorna quando o campo apresenta valor maior que o especificado.
$lt	Retorna quando o campo apresenta valor menor que o especificado.
$in	Verifica se combina com qualquer dos valores do vetor.
$nin	Verifica se não ocorre combinação com qualquer dos valores do vetor.
$and	Combinação lógica de condições com uso de and.
$or	Combinação lógica de condições com uso de or.
$exists	Retorna os documentos que apresentam determinado campo.
$all	Verifica se todos os valores do vetor estão presentes.
$size	Verifica se o campo, do tipo vetor, tem o tamanho especificado.
$$regex	Seleciona os documentos a partir de uma expressão regular.

Quadro: Alguns operadores oferecidos pelo MongoDB.

Por exemplo, poderíamos recuperar todos os produtos com quantidade acima de 500 por meio do operador **$gt**.

```
typescript

{quantidade: {$gt: 500}}
```

Resultados melhores podem ser observados com uma gama maior de dados, sendo mais simples efetuar inserções em bloco por meio do utilitário **mongo**, um **shell** para execução de comandos, encontrado no diretório **bin** do MongoDB. Após iniciar o shell, podemos selecionar o banco de trabalho e executar uma inserção múltipla de documentos.

```
typescript

use loja;
 db.inventario.insertMany([
     {item: 'caneta', qtd: 25, cores: ['vermelho','azul']},
     {item: 'lapis', qtd: 50, cores: ['preto']},
     {item: 'papel', qtd: 100, cores: ['azul','branco','verde']},
     {item: 'pilot', qtd: 75, cores: ['azul','vermelho']},
     {item: 'post-it', qtd: 45, cores: ['amarelo','verde']}
 ]);
```

Com a execução dos comandos, teremos a inclusão de cinco produtos em uma coleção com o nome "**inventario**". As consultas podem ser efetuadas, em seguida, utilizando o comando **find** e os operadores do MongoDB.

```
typescript

> db.inventario.find({cores: {$all: ['vermelho','azul']}});
 {'_id': ObjectId('60de880ffe30ca11b392f98a'), 'item': 'caneta',
 'qtd': 25, 'cores': ['vermelho', 'azul']}
 {'_id': ObjectId('60de880ffe30ca11b392f98d'), 'item': 'pilot',
 'qtd': 75, 'cores': ['azul', 'vermelho']}
```

No exemplo anterior, temos o uso do operador **$all**, recuperando os documentos em que o campo **cores** contém todos os valores especificados no vetor, ou seja, **vermelho** e **azul**. Temos, como resultado da execução, o retorno dos itens **caneta** e **pilot**.

Com nosso banco configurado e alimentado, podemos iniciar a criação de um aplicativo para acesso aos dados. Por se tratar de uma tecnologia servidora, o acesso não poderá ser feito diretamente a partir do dispositivo móvel, exigindo um servidor Web adequado para intermediar as operações sobre o MongoDB.

Servidor Node.js com Express

Ao contrário dos exemplos que vimos até aqui, no caso do Mongo DB, temos uma base que deve estar em um servidor, exigindo uma interface de acesso via rede. Utilizaremos um servidor **minimalista**, que pode ser criado facilmente com o **Node.js**, com base na biblioteca **Express**.

```
typescript

npm install -g express-generator
 express -e Loja2021
 cd Loja2021
 npm install
 npm install mongodb
```

Iniciaremos a programação de nosso servidor com a persistência dos dados, criando o arquivo **GestorDB.js**, no diretório **dados**. Devemos efetuar a conexão com o MongoDB e definir as operações necessárias para **incluir**, **excluir** e **selecionar** documentos, a partir da coleção **produtos**.

```
typescript

const mongoClient = require('mongodb').MongoClient;

 mongoClient.connect('mongodb://localhost',
                     {useUnifiedTopology: true})
             .then(conn => global.conn = conn.db('loja'))
             .catch(err => console.log(err))

 function findAll() {
     return global.conn.collection('produtos').find().toArray();
 }

 function insert(produto) {
     return global.conn.collection('produtos')
                       .insertOne(produto);
 }

 function deleteOne(codigo) {
     return global.conn.collection('produtos')
                       .deleteOne({ _id: codigo });
 }

 module.exports = { findAll, insert, deleteOne }
```

Inicialmente, obtemos uma referência ao **MongoClient**, a partir da biblioteca **mongodb**, e a utilizamos para conectar com o banco de forma local. Com base no operador **then**, que irá esperar o término da execução assíncrona, definimos uma conexão com o nome **global.conn**, selecionando o banco **loja**.

Veja a seguir o comportamento das funções **findAll**, **insert** e do método **deleteOne**.

findAll

A função **findAll** acessa a coleção com o nome "**produtos**", a partir da conexão, e retorna os documentos na forma de um vetor. Como não foram definidos parâmetros na busca efetuada pelo método find, todos os documentos da coleção serão recuperados.

insert

A função **insert** (que tem como parâmetro um documento) apenas aciona o método **insertOne** com a passagem do documento.

deleteOne

O método **deleteOne** invoca o método de mesmo nome na coleção, passando o parâmetro **_id** com o valor do código.

Export

O módulo exporta suas funções para que possam ser invocadas a partir de outros arquivos do tipo **JavaScript** ou **TypeScript**.

Nosso gestor de persistência deve ficar disponível para todo o sistema, o que é feito com o acréscimo de uma linha no arquivo **www**, do tipo JavaScript, dentro do diretório **bin**, com a definição da referência **global.db**. Todo o restante do código original do arquivo deverá ser mantido inalterado.

```
javascript

#!/usr/bin/env node

 /**
     * Module dependencies.
     */

 global.db = require('../dados/GestorDB');
```

Agora vamos alterar o arquivo **index.js**, raiz do servidor Web, dentro do diretório **routes**, em que temos a definição de algumas rotas de exemplo, as quais serão substituídas por novas rotas. Seguiremos o padrão REST, utilizando os métodos do protocolo HTTP para gerenciar as operações de consulta e manipulação de dados, como pode ser observado no quadro seguinte.

Método HTTP	Utilização
GET	Consultas, com o retorno de entidades e coleções no formato JSON.
POST	Inserção de entidade, a partir dos dados no formato JSON.
PUT	Edição da entidade, com base na chave e nos dados em formato JSON.
DELETE	Remoção da entidade, com base na chave primária.

Quadro: Alguns operadores oferecidos pelo MongoDB.

Segundo o padrão descrito:

1

Um endereço do tipo **http://servidor/produto** retorna todos os produtos.

2

Enquanto **http://servidor/produto/101** retorna apenas o produto cuja chave primária apresenta o valor 101.

Em ambos os casos, os dados referentes aos produtos devem ser fornecidos no formato JSON.

Para nosso exemplo, consideramos a entidade como raiz do site, retornando todos os produtos com acesso ao endereço de base, além de acrescentarmos as rotas new, no modo post, e remove, complementada com o código, no modo delete.

O novo código do arquivo index.js pode ser observado na listagem seguinte.

```typescript
var express = require('express');
 var router = express.Router();

 router.get('/', async (req, res, next) => {
     try {
     const docs = await global.db.findAll();
     let valores = docs.map((item) => {return
                             {codigo:item._id, nome:item.nome,
                             quantidade:item.quantidade}});
     res.send(valores);
     } catch (err) {
     res.send({resultado:'Erro ao Listar', mensagem: err});
     }
 })

 router.post('/new', async (req, res, next) => {
     const _id = parseInt(req.body.codigo);
     const nome = req.body.nome;
     const quantidade = parseInt(req.body.quantidade);
     try {
     const result = await global.db.insert(
                                   {id, nome, quantidade});
     res.send({resultado:'Inserido'});
     } catch (err) {
     res.send({resultado:'Erro ao Inserir', mensagem: err});
     }
 })

 router.delete('/remove/:codigo', async (req, res, next) => {
     const codigo = parseInt(req.params.codigo);
     try {
     const result = await global.db.deleteOne(codigo);
     res.send({resultado:'Removido'});
     } catch (err) {
     res.send({resultado:'Erro ao Remover', mensagem: err});
     }
 })

 module.exports = router;
```

Inicialmente, temos uma referência ao módulo **express**, e a obtenção de um objeto para roteamento com o nome **router**. Ele será responsável por interceptar as chamadas e direcionar as respostas, de acordo com o protocolo HTTP, em que **req** é uma referência para a requisição e **res** para a resposta.

Em relação às rotas, veja a seguir:

Primeira rota

A primeira rota trabalha no modo **GET**, tendo como endereço a raiz, em que começamos aguardando a execução assíncrona de **findAll**, e recebendo os documentos na variável **docs**. Os documentos já são recebidos no formato **JSON**, mas o banco utiliza **_id** como padrão para a chave primária, exigindo o mapeamento dos campos para o formato da classe **Produto**, por meio de **map**, antes de retornar os valores convertidos via **send**.

Segunda rota

Na rota seguinte, temos uma chamada no modo **POST** para o endereço **new**, em que será fornecido um objeto do tipo **Produto**, no formato **JSON**, por meio do corpo da requisição, sendo considerado o código do produto como valor de **_id**. Após a execução do método **insert**, com a passagem dos valores obtidos, é fornecida uma resposta positiva via **send**.

Última rota

Na última rota, temos um endereço parametrizado, a partir de **remove**, tendo o código como parâmetro, e utilizando o modo **DELETE**. O parâmetro é recuperado via **params** e convertido para o formato inteiro, sendo utilizado na chamada para **deleteOne**, com o retorno da resposta positiva após a execução da função.

Em todos os casos, são utilizados blocos protegidos, permitindo que a mensagem de erro seja retornada para o usuário quando ocorrem exceções. Agora só precisamos iniciar nosso servidor, que neste ponto já se encontra totalmente funcional.

```
typescript

npm start
```

Fazendo a chamada para **http://localhost:3000**, uma vez que a porta padronizada do **express** tem valor **3000**, teremos como retorno a coleção de produtos no formato JSON.

```
typescript

[{'codigo':1,'nome':'Banana','quantidade':2000},
 {'codigo':2,'nome':'Morango','quantidade':500}]
```

Express e MongoDB

No vídeo a seguir, abordamos a criação de um servidor minimalista, com base em Express, para acesso a uma base de dados MongoDB, contemplando operações de consulta e manipulação dos dados, segundo uma arquitetura REST.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Cadastro com MongoDB

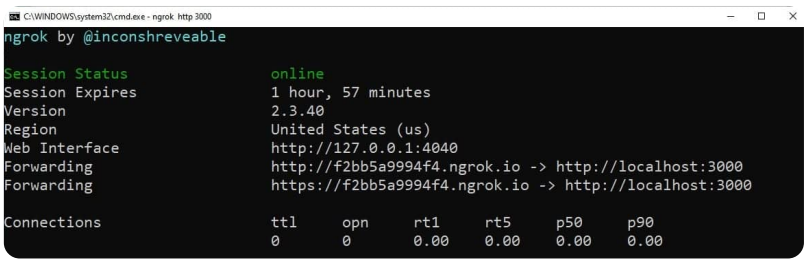
Com o servidor pronto, podemos definir nosso aplicativo cadastral, com acesso via rede, tendo como base o React Native. No entanto, não podemos utilizar o endereço localhost no aplicativo Android, pois estaríamos tentando acessar o próprio dispositivo, com base no sistema Linux. Atualmente, devido às restrições de segurança impostas nas atuais versões da plataforma, é aceito apenas o protocolo HTTPS.

Recomendação

Vamos baixar um aplicativo chamado ngrok, que faz um tunelamento de nosso servidor, fornecendo via HTTP e HTTPS simultaneamente dentro de uma mesma rede, o que irá viabilizar o acesso, a partir do Android, sem a necessidade de configurações complexas, em que teríamos a importação e o uso de certificados digitais no servidor.A utilização do ngrok é muito simples, bastando executar no console, com a passagem do protocolo HTTP e o número da porta, que no caso é 3000.

ngrok http 3000

Como resultado da execução do comando, teremos a definição de dois endereços, um para HTTP e outro para HTTPS, indicados no console, além do acompanhamento das chamadas que forem efetuadas. O nome do site é gerado de forma aleatória pelo **ngrok**, mas é possível especificar um nome por meio da opção **subdomain**, nos planos pagos.



Execução do aplicativo ngrok.

Para a execução apresentada, teremos o endereço **https://f2bb5a9994f4.ngrok.io**, mas você deverá utilizar o nome fornecido na execução em **seu ambiente**, lembrando que o nome mudará cada vez que o **ngrok** for executado novamente.

Já podemos iniciar a criação de nosso cliente, com os comandos apresentados a seguir.

```
typescript

react-native init CadastroMongo
 cd CadastroMongo
 npm install --save axios
 npm install --save @types/axios
 npm install @react-navigation/native @react-navigation/stack
 npm install react-native-reanimated react-native-gesture-handler react-native-screens
react-native-safe-area-context @react-native-community/masked-view
```

Com o projeto criado, copiaremos, a partir do projeto CadastroRealm, os arquivos **App, CommonStyles, ProdutoForm, ProdutoItem e ProdutoLista**, todos do tipo JavaScript, além do arquivo **Produto**, do tipo TypeScript, posicionado no diretório **dados**. Após a replicação da estrutura básica, vamos efetuar a conexão com nosso servidor e criar um gestor de dados apropriado para lidar com a comunicação remota.

Nosso próximo passo será a criação do arquivo **ApiService.ts**, no diretório **dados**, para implementar a conexão com o servidor, em que será utilizada a biblioteca **axios**, que permite definir clientes REST com grande facilidade.

> Lembre-se de utilizar o endereço HTTPS fornecido pelo aplicativo ngrok, quando executado em seu ambiente, no lugar do valor de exemplo adotado na listagem apresentada a seguir.

```
typescript

import axios, { AxiosInstance } from 'axios';

 const endereco = 'https://f2bb5a9994f4.ngrok.io'
 // use seu endereço ngrok

 const api:AxiosInstance = axios.create({
     baseURL: endereco,
 });

 export default api;
```

Como podemos observar, o acesso ao servidor com uso de **axios** é muito simples, pois temos apenas que utilizar o método **create**, com a passagem do endereço no atributo **baseURL**, gerando um objeto do tipo **AxiosInstance**. Ao final, exportamos o objeto, com o nome **api**, para que seja utilizado posteriormente no acesso ao nosso servidor.

Agora, criaremos o arquivo **GestorDados.ts**, no diretório **dados**, para intermediar todas as operações de consulta e manipulação de nossos produtos, utilizando o código apresentado na listagem seguinte.

```
typescript

import { Produto } from './Produto';
 import api from './ApiService';

 class GestorDados {
     public async remover(chave: string){
     await api.delete('/remove/'+chave);
     }
     public async adicionar(produto: Produto){
     await api.post('/new',produto);
     }
     public async obterTodos(): Promise>{
     let resposta = await api.get('/');
     return resposta.data;
     }
 }

 export default GestorDados;
```

No método **remover**, recebemos uma chave como parâmetro, e montamos uma URL por meio da concatenação de "**/remove/**" com a chave. Para um produto de código 120, o endereço resultante seria "/remove/120", sendo enviado para o servidor via método **delete** da **api**, em que a recepção da chamada irá excluir o produto no MongoDB.

Uma utilização levemente diferente do **axios** pode ser observada no método **adicionar**, que tem um produto como parâmetro. Agora, efetuamos uma chamada para o endereço "**/new**", pelo método **post**, tendo como pacote de dados o nosso produto, que será transformado para **JSON** no **corpo** da mensagem.

Com relação ao método **obterTodos**, ele faz uma chamada do tipo **get** para o endereço **raiz** do servidor, e retorna o campo **data** da resposta, que, como vimos na análise de nosso servidor, é um vetor de produtos no formato JSON.

Comentário

Como visto em outros exemplos, os métodos criados são assíncronos, e já que todas as assinaturas foram mantidas, nesse ponto completamos as funcionalidades necessárias para os arquivos copiados de CadastroRealm. Agora, com o MongoDB, o nosso servidor e o ngrok iniciados, podemos executar CadastroMongo, observando como funcionará da mesma forma que os outros exemplos, mas com a persistência ocorrendo no servidor.

Verificando o aprendizado

Questão 1

O banco de dados MongoDB trabalha com coleções de documentos no formato JSON e oferece muitas opções para a filtragem de uma consulta, com base no método find, por meio de operadores específicos. Qual dos operadores permite testar a existência de um determinado atributo em cada documento?

A $in

B $size

C $all

D $nin

E $exists

A alternativa E está correta.

Por meio do operador $exists, retornamos o conjunto de documentos que apresentam um atributo específico, como cor ou altura. Podemos observar um exemplo de utilização a seguir, com base no shell do MongoDB, retornando os produtos com o atributo cor.**> db.produtos.find({ cor: { $exists: true } })**

Questão 2

Para acesso ao MongoDB, por se tratar de uma tecnologia servidora, precisamos do intermédio de um servidor HTTPS, e uma opção de implementação é por meio do Express, em conjunto com ngrok. Ao utilizar Express, interceptamos as chamadas, de acordo com o método HTTP correto, a partir dos métodos da classe

A MongoClient.

B Transaction

C Parameter

D Router

E HTTPSession

A alternativa D está correta.

Por meio dos métodos oferecidos por Router, podemos interceptar as chamadas HTTP ou HTTPS dos tipos POST, GET, PUT e DELETE. Por exemplo, o método post, tendo como parâmetro "/produto", permitiria o tratamento de uma chamada do tipo POST para o endereço "http://localhost:3000/produto".

Considerações finais

Analisamos diversas formas de persistência para os aplicativos criados por meio do React Native, incluindo conteúdo do tipo chave-valor em arquivos planos, utilização de elementos estruturados em bases de dados locais, e o acesso a uma base do tipo NoSQL de forma remota.

Em termos de elementos estruturados, utilizamos o modelo relacional com o banco de dados SQLite, enquanto no modelo orientado a objetos foi utilizado Realm. Já com relação aos dados semiestruturados, foi possível observar o acesso a uma base remota MongoDB, do tipo NoSQL baseado em documentos.

Todos os conhecimentos adquiridos foram exemplificados mediante um cadastro muito simples, com o modo de persistência sendo modificado a cada módulo, o que permitiu demonstrar como uma interface pode ser amplamente reaproveitada, desde que haja um sistema de persistência encapsulado adequadamente.

Podcast

Ouça agora um resumo dos assuntos abordados ao longo dos módulos.

Conteúdo interativo

Acesse a versão digital para ouvir o áudio.

Explore +

- Leia o artigo **Building an offline first app with React Native and SQLite: 2020 refresh**, de Bruce Lefebvre, e saiba como criar o aplicativo offline first com React Native e SQLite.
- Leia a documentação oficial do **Realm Database para React Native**.
- Leia o artigo **Introdução ao MongoDB e conheça o framework**.
- Leia o artigo de Guilherme Alves, Express e MongoDB, **Conectando uma aplicação web ao MongoDB e saiba como criar uma API estilo REST com Express e MongoDB**;
- Leia o **DB-engines ranking**, a classificação dos principais bancos de dados do mercado (site db-engines.com), incluindo os relacionais, os orientados a objeto e os NoSQL.

Referências

BODUCH, A; DERKS, **R. React and React Native**. 3th ed. United Kingdom: Packt Publishing, 2020.

BROWN, E. **Programação Web com Node e Express**: beneficiando-se da Stack JavaScript. São Paulo: Novatec, 2020.

ESCUDELARIO, B.; PINHO, D. **React Native**: desenvolvimento de aplicativos móveis com React. São Paulo: Casa do Código, 2020.

SOUZA, M. **Desvendando o MongoDB**: do Mongo Shell ao Java Driver. Rio de Janeiro: Ciência Moderna, 2015.