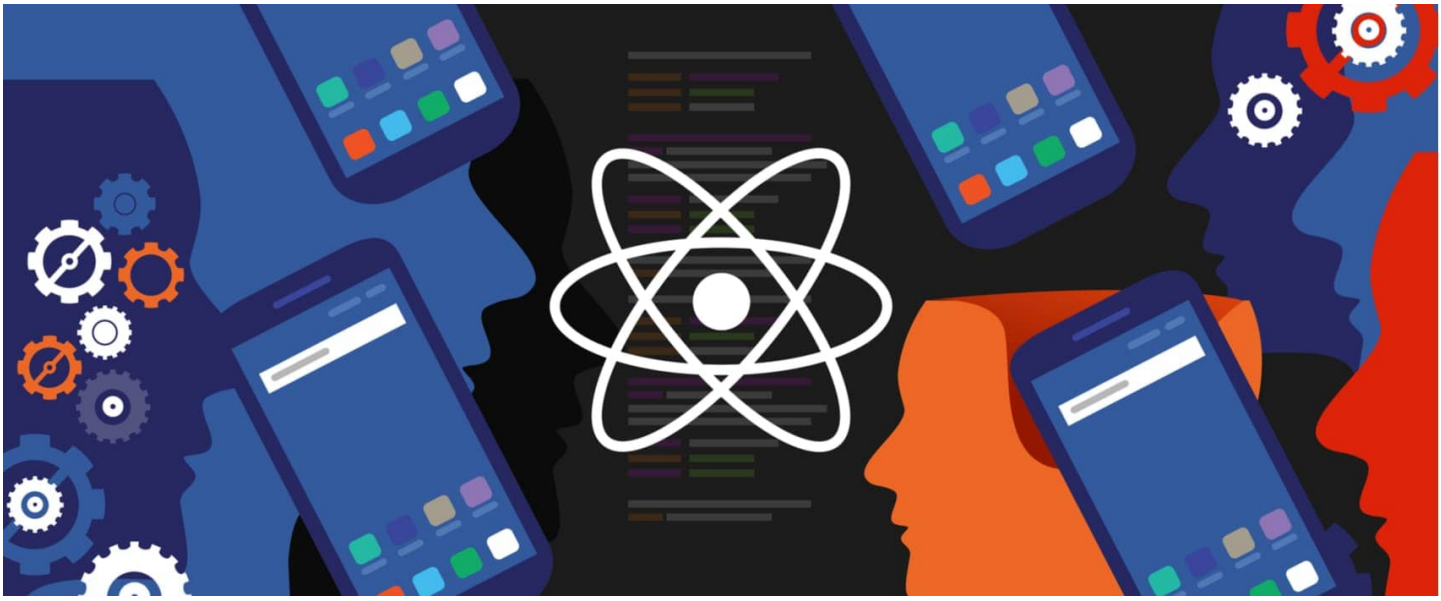


Tópicos avançados em React Native

Prof. Denis Cople

false

Descrição

Utilização das arquiteturas MVC, Flux e Redux nos aplicativos React Native, implementação de criptografia para armazenamento de dados e guarda de senhas, garantia de qualidade via testes unitários e análise de performance, além do processo de publicação do aplicativo.

Propósito

Com o conhecimento adquirido, o aluno será capaz de desenvolver aplicativos, na plataforma React Native, alinhados às melhores técnicas adotadas pelo mercado.

Preparação

Antes de iniciar este conteúdo, é necessário configurar o ambiente, com a instalação do JDK, Android Studio, Visual Studio Code e NodeJS, 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 utilizado para testes deve ter o aplicativo Expo, obtido gratuitamente na loja padrão da plataforma.

Objetivos

Módulo 1

Arquitetura MVC com React Native

Analisar a arquitetura MVC com React Native.

Módulo 2

Arquiteturas Flux e Redux com React Native

Empregar arquiteturas Flux e Redux com React Native.

Módulo 3

Criptografia com React Native

Aplicar criptografia na persistência e no controle de acesso com React Native.

Módulo 4

Publicação de aplicativos com React Native

Empregar testes e análise de performance na publicação de aplicativos com React Native.

Introdução

Neste conteúdo, abordaremos elementos essenciais para a construção de aplicativos comerciais robustos, que estejam alinhados aos requisitos e processos de produção adotados pelas mais modernas empresas de tecnologia. Iniciaremos nossos estudos abordando a utilização de arquiteturas robustas, como MVC, baseada em camadas, Flux e Redux, as duas últimas direcionadas para o fluxo de dados.

Após a adoção das arquiteturas, veremos como integrar elementos criptográficos em nossos aplicativos, garantindo o sigilo e o controle de acesso, além de definir testes unitários, com base em Jest, e verificar o funcionamento de ferramentas para análise de performance. Com nossos aplicativos prontos, dentro de padrões de qualidade devidamente aferidos, veremos o processo de publicação nas lojas mais conhecidas.

1 - Arquitetura MVC com React Native

Ao final deste módulo, você será capaz de analisar a arquitetura MVC com React Native.

Padrões de desenvolvimento

Os **padrões de projeto (design patterns)** surgiram com o objetivo de oferecer um catálogo de soluções para problemas observados de forma recorrente, na área de desenvolvimento de software.

Todo padrão de desenvolvimento deve apresentar nome, descrição do problema a ser resolvido, descrição da solução e possíveis consequências de sua adoção. De forma geral, a solução apresentada pelo padrão é descrita não apenas de forma textual, mas com a utilização de diagramas da **UML** (*Unified Modeling Language*).

Existem muitos conjuntos de padrões, mas o GoF (Gang of Four), definido pelos autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, determinou uma família com 23 padrões, dividida em três grupos:

Grupo	Caracteristicas	Padrões de Desenvolvimento
Padrões criacionais	Utilizados na definição de métodos flexíveis para criação de objetos.	Abstract Factory, Builder, Factory Method, Prototype e Singleton.
Padrões estruturais	Para a definição de estruturas compostas flexíveis e eficientes.	Adapter, Bridge, Composite, Decorator, Facade, Flyweight e Proxy.
Padrões comportamentais	Voltados para algoritmos e atribuição de responsabilidades.	Chain of Reponsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor e Template Method.

Quadro: Organização dos padrões GoF.
Elaborado por Denis Cople, adaptado por Gérsica Telles e Heloise Godinho.

- **SINGLETON**: Objetiva definir apenas uma instância para determinada classe, algo útil para controladores de acesso e gerenciadores de conexões com bancos de dados. A implementação de um componente Singleton exige apenas a definição de uma instância global privada, um método estático público para recuperação da instância e um construtor privado.
- **DAO (Data Access Object)**: Com larga aceitação no mercado, é um padrão referente à concentração das operações de acesso a banco de dados, o que evita a multiplicação de comandos SQL ao longo de todo o código. A utilização do DAO diminui as dificuldades relacionadas à manutenção, ao mesmo tempo em que aumenta o reuso da persistência de dados para os sistemas.
- **Facade**: Simplifica a utilização de subsistemas complexos, como nas chamadas efetuadas de forma sucessiva para um grupo de componentes do tipo DAO, na implementação de um processo de negócios mais amplo. Inclusive, em se tratando de componentes de negócios, é comum a criação de pools de objetos, tendo como base o padrão Flyweight, que são acessados remotamente, segundo o padrão Proxy.

Saiba mais

As interfaces de nossos aplicativos, criados no ambiente do **React Native**, fazem ampla utilização do padrão **Composite**, em que um componente maior é criado a partir de outros menores, de forma hierárquica, e do **Observer**, com a atualização dos valores presentes na tela de forma automática, a partir da configuração efetuada via useState.

Para sistemas mais complexos, é conveniente organizar os processos em componentes do tipo Command, com escolha do fluxo correto pelo padrão Strategy. Também é comum observar, em frameworks, a adoção de modelos genéricos para a geração de objetos, com base no padrão Abstract Factory, ou a definição de processos genéricos que devem ser adaptados ao contexto de utilização, via Template Method.

Enquanto alguns padrões só demonstram sua utilidade em contextos realmente amplos, outros são adotados nas situações mais comuns do desenvolvimento de um software, como, por exemplo, o padrão Iterator, que define uma metodologia para acessar todos os elementos de uma coleção de forma sequencial, sendo a base do método forEach.

Padrões arquiteturais

Os padrões de desenvolvimento permitem o reuso do conhecimento, trazendo soluções padronizadas para problemas já conhecidos no mercado. Embora eles solucionem boa parte dos problemas internos do desenvolvimento de um sistema, devemos observar também a necessidade de uma estratégia arquitetural para satisfazer a determinados **domínios**, e daí surge a ideia por trás dos **padrões arquiteturais**.

Uma arquitetura muito comum no mercado corporativo é a de **Broker**, utilizada para objetos distribuídos, como CORBA e EJB, definindo a presença de stubs e skeletons, descritores de serviço, protocolo de comunicação, e demais elementos para distribuição do processamento. O uso de um padrão arquitetural sugere a utilização de diversos padrões de desenvolvimento associados ao mesmo, como o **Proxy e Flyweight**, que são adotados, respectivamente, na comunicação e gestão do pool de objetos do Broker.

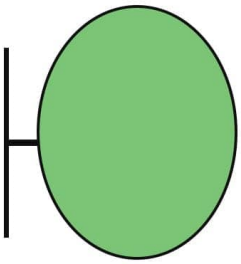
A classificação de algumas arquiteturas pode ser observada no quadro seguinte.

Modelo	Estilo Arquitetural
Sistemas Distribuídos	Broker
Mud to Structure	Camadas, Pipes/Filters e Blackboard
Sistemas Interativos	MVC e PAC
Sistemas Adaptáveis	Microkernel e Reflexiva

Quadro: Arquiteturas comuns no desenvolvimento de software.
Elaborado por Denis Cople, adaptado por Heloise Godinho.

Entre os diversos padrões arquiteturais existentes, o **Model-View-Controller**, ou **MVC**, acabou se tornando uma referência para o mercado de desenvolvimento, no que se refere às aplicações cadastrais com uso de bancos de dados. O padrão promove a divisão do sistema em três camadas, englobando a persistência de dados (Model), processos de negócios (Controller), e interação com o usuário (View).

Veja a seguir a arquitetura MVC e simbologia utilizada:

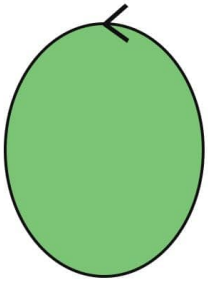


View

Telas e interfaces do sistema.

Fornece as informações obtidas a partir do controle.

Envia as solicitações do usuário ao controle.

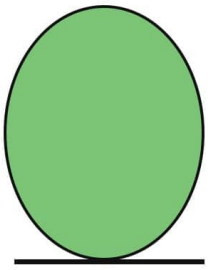


Controller

Componentes que definem o comportamento do sistema.

Mapeia as opções para consultas e alterações.

Define as regras de negócio do sistema.



Model

Elementos voltados para a persistência de dados.

Encapsula o estado geral do sistema.

Trabalha com padrão DAO e mapeamento objeto-relacional.

Na primeira versão do MVC, a camada Model utilizava entidades que gravavam seus próprios estados no banco, segundo o padrão de desenvolvimento Active Record. Os dados eram transitados no padrão VO (Value Object), que trazia os mesmos dados das entidades, mas sem métodos de persistência, impedindo a manipulação do banco pela View.

Já na segunda versão do MVC, a camada Model sofreu mudanças, e as entidades passaram a deter apenas os atributos referentes aos campos da tabela, e a persistência foi delegada para classes no padrão DAO, permitindo que a própria entidade seja transitada entre as camadas.

> **A regra primordial da arquitetura MVC é a de que apenas a camada Model deve permitir o acesso ao banco de dados, e apenas a Controller pode invocar os componentes DAO oferecidos pela Model.**

Como a camada View se comunica apenas com a Controller, nós garantimos o correto isolamento do banco de dados, além de generalizar os processos de negócios, viabilizando a utilização de múltiplas interfaces para o mesmo sistema.

Podemos observar, na figura seguinte, uma representação da arquitetura MVC, com a utilização de componentes comuns do React Native.

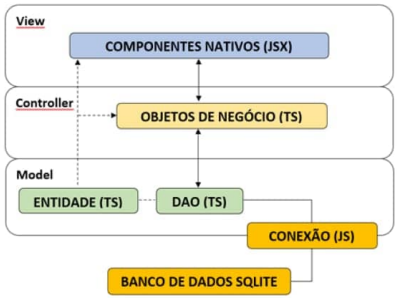


Diagrama da arquitetura MVC no React Native.

Padrão DAO no React Native

Ao trabalharmos com um banco de dados relacional, como o SQLite, devemos efetuar o mapeamento objeto-relacional, em que os registros resultantes da consulta são transformados em coleções de entidades. A transformação deve ocorrer na classe DAO, permitindo que todo o resto do sistema trabalhe apenas com o modelo baseado em objetos.

Como os processos básicos de consulta e manipulação de dados são muito repetitivos, é possível adotar um modelo genérico, em que temos uma classe de entidade T e uma classe de chave primária K associada. Para tal, definiremos uma classe DAO genérica, no arquivo GenericDAO.ts, de acordo com a listagem apresentada a seguir.

Javascript

```
1  import db from './DatabaseInstance';
2
3   export abstract class GenericDAO<T,K>{
4     protected abstract getCreateSQL(): string;
5     protected abstract getTableName(): string;
6     protected abstract getInsertSQL(): string;
7     protected abstract getInsertParams(entidade: T): any[];
8     protected abstract getUpdateSQL(): string;
9     protected abstract getUpdateParams(entidade: T): any[];
10    protected abstract getDeleteSQL(): string;
11    protected abstract getSelectAllSQL(): string;
```

```
12      protected abstract getSelectOneSQL(): string;
13      protected abstract getEntidade(linha: any): T;
14
```

De modo geral, os métodos **incluir e alterar** utilizam uma transação, a partir da qual é executado um comando SQL, tendo como parâmetros os dados da entidade fornecida como parâmetro, segundo alguma ordem, em um vetor. Os fatores dinâmicos de ambos os processos devem ser fornecidos pelos descendentes, com um método **getter** para o comando SQL, e outro para obtenção do vetor a partir da entidade. Com relação ao método **excluir**, embora trabalhe da mesma forma que os anteriores, exige apenas o fornecimento de um comando SQL, já que o vetor deve ser formado pela própria chave.

> **No método obterTodos teremos, como parâmetro, uma callback com o recebimento de um vetor de entidades, sendo executado o comando SQL de consulta que será fornecido pelo descendente, via getter apropriado, e percorrido todo o conjunto resultante, com a adição de uma entidade ao vetor para a cada linha.**

O vetor preenchido será fornecido na chamada para a callback, e no processo, como um todo, precisaremos de outro método getter, para efetuar o mapeamento objeto-relacional, recebendo uma linha da consulta, do tipo any, e retornando um objeto do tipo da entidade.

Temos um comportamento muito similar ao anterior no método obter, mas com o uso de uma chave como parâmetro, além da callback com o recebimento de uma instância simples da entidade. Assim como na exclusão, o método obter estará preparado apenas para chaves que não sejam compostas, e será apoiado nos getters para o comando SQL de consulta pela chave e para o mapeamento objeto-relacional.

Finalmente, o construtor deve testar a existência da tabela, com o nome fornecido por getTableName, e caso não exista, deve ser criada por meio do comando SQL retornado via getCreateSQL, ambos getters que devem ser implementados pelos descendentes.

Atenção

A classe GenericDAO está seguindo o padrão Template Method, em que os processos são definidos de forma genérica, com lacunas que devem ser preenchidas no nível dos descendentes, pela implementação dos métodos abstratos.

Com a criação de GenericDAO, definimos um arcabouço de persistência que será especializado para cada entidade, sendo necessário apenas compreender o objetivo de cada método getter, conforme o quadro apresentada a seguir, diminuindo muito o esforço de programação.

Método Getter	Funcionalidade
getCreateSQL	Retorna o comando SQL para criação da tabela.
getTableName	Retorna o nome da tabela.
getInsertSQL	Retorna o SQL parametrizado para inserção de registro.
getInsertParams	Retorna o vetor de parâmetros para o comando de inserção. Parâmetro de entrada: entidade do tipo T.
getUpdateSQL	Retorna o SQL parametrizado para alteração de registro.
getUpdateParams	Retorna o vetor de parâmetros para o comando de alteração. Parâmetro de entrada: entidade do tipo T.
getDeleteSQL	Retorna o SQL parametrizado para exclusão a partir da chave.
getSelectAllSQL	Retorna o comando SQL para consulta global.
getSelectOneSQL	Retorna o SQL parametrizado para consulta a partir da chave.
getEntidade	Retorna uma entidade a partir dos dados do registro. Parâmetro de entrada: linha (any).

Quadro: Funcionalidade dos getters abstratos de GenericDAO.
Elaborado por: Denis Cople.

Para deixar o arcabouço funcional, precisamos de uma variável db para conexão com o banco SQLite, a qual será definida no arquivo DatabaseInstance.js, contendo a listagem seguinte.

Javascript

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

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

 export default db;
```

No domínio de nosso exemplo, adotaremos uma classe de entidade Aluno, contendo a matrícula, o nome e a data de registro, que deverá ser codificada no arquivo Aluno.ts.

Javascript

```
  export class Aluno{
    matricula: string;
    nome: string;
    registro: Date;
    constructor(matricula: string,nome: string,registro: Date){
        this.matricula = matricula;
        this.nome = nome;
        this.registro = registro;
    }
}
```

Finalmente, definiremos o componente DAO concreto, no arquivo AlunoDAO.js, que terá os getters configurados para utilização de entidades do tipo Aluno.

Considerando a matrícula como chave primária, começamos definindo a classe AlunoDAO a partir da herança de GenericDAO, parametrizada para Aluno e string, ou seja, os tipos da entidade e da chave, respectivamente.

Javascript

```
import {Aluno} from './Aluno';
 import {GenericDAO} from './GenericDAO';

 export default class AlunoDAO extends GenericDAO<Aluno,string>{
   protected getCreateSQL(): string {
     return ' CREATE TABLE ALUNO (MATRICULA VARCHAR(10) '+
            ' PRIMARY KEY,NOME VARCHAR(20),REGISTRO INTEGER)';
   }
   protected getTableName(): string { return 'ALUNO'; }
   protected getInsertSQL(): string {
     return 'INSERT INTO ALUNO VALUES ( ?, ?, ? )';
   }
   protected getInsertParams(entidade: Aluno): any[] {
     let criacao = new Date();
```

> **Os comandos SQL, parametrizados ou não, fornecido pelos getters, estão associados à tabela ALUNO, cujo nome deve ser retornado em getTableName. Devemos lembrar que os parâmetros são definidos por meio de pontos de interrogação, com o preenchimento ocorrendo na execução do comando, a partir de um vetor de valores.**

Para configurar os parâmetros de alteração, temos apenas os atributos necessários da entidade, na ordem correta, dentro de um vetor. Já para os parâmetros da inserção, instanciamos um objeto com a data corrente, e montamos o vetor com a matrícula e o nome do aluno, além do valor da data em termos de milissegundos, obtido via getTime.

Finalmente, no método getEntidade temos o retorno de um objeto do tipo Aluno, com a recuperação da data a partir do valor armazenado no banco, em milissegundos, no campo REGISTRO, além dos campos MATRICULA e NOME.

Um pequeno exemplo de utilização pode ser observado na listagem seguinte.

Javascript

```
let dao = new AlunoDAO();
 dao.obterTodos((alunos)=>
     alunos.forEach((aluno)=> console.log(aluno.nome))
 );
```

A execução do trecho de código apresentado iria recuperar todos os alunos, a partir da base de dados, e imprimiria o nome de cada aluno recuperado por meio da consulta. Note a utilização do método forEach, que segue o padrão de desenvolvimento Iterator.

Aplicativo MVC no React Native

Vamos criar um aplicativo cadastral, na arquitetura MVC, com utilização de banco de dados SQLite. Inicialmente, configuraremos nosso projeto, que executará no modelo nativo, com os comandos apresentados a seguir.

Terminal

```
react-native init CadastroMVC
 cd CadastroMVC
 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
```

Com o projeto configurado, criaremos o diretório cadastro, seguido do subdiretório model, onde colocaremos os arquivos DatabaseInstance.js, Aluno.ts, GenericDAO.ts e AlunoDAO.ts, todos gerados no tópico anterior, definindo nossa camada Model.

Agora, adicionaremos o subdiretório controller em cadastro, e começaremos a definir a próxima camada. Criaremos, no novo subdiretório, o arquivo GenericController.ts, com o código da listagem seguinte.

Javascript

```
import {GenericDAO} from '../model/GenericDAO';

 export default abstract class GenericController<   T,K> {
   protected dao: GenericDAO<T,K>;
   protected abstract setDAO(): void;

   constructor(){
     this.setDAO();
   }
   public async incluir(entidade: T): Promise<boolean>{
     try {
       await this.dao.incluir(entidade);
       return true;
     }catch(error){
```

Nosso controlador genérico é uma variação do padrão Template Method, cuja única lacuna que deverá ser preenchida pelos descendentes é o fornecimento do componente DAO, por meio do método getDAO. Sendo um controlador direcionado, ele receberá as classes da entidade e da chave como parâmetros de tipagem.

A programação é bastante simples, com ampla utilização do componente DAO, em que as chamadas para obter e obterTodos apenas encapsulam os métodos equivalentes, que se encontram na camada Model. Como são métodos baseados no uso de callbacks, não precisam da classificação como assíncronos.

Já os métodos inserir, alterar e excluir são assíncronos, encapsulando os métodos equivalentes da camada Model, e retornando valores booleanos que indicam o sucesso ou não da execução. Nos três métodos, temos a chamada para o componente DAO em um bloco try, em que o erro desviaria para o bloco catch, retornando false, e o sucesso da execução é sequenciado pelo retorno do valor true.

Já podemos criar nosso controlador de alunos, com o nome AlunoController.ts, dentro do subdiretório controller. A listagem do controlador é apresentada a seguir.

Javascript

```
import { Aluno } from '../model/Aluno';
 import AlunoDAO from '../model/AlunoDAO';
 import GenericController from './GenericController';

 export default class AlunoController
                extends GenericController<Aluno,string> {
   protected setDAO(): void {
     this.dao = new AlunoDAO();
   }
 }
```

Tudo que precisamos fazer, para cada novo controlador, é herdar de GenericController, parametrizando com a classe de entidade e o tipo da chave, além de fornecer um DAO, compatível com a parametrização, por meio de getDAO.

Já podemos construir a última camada, responsável pela interface com o usuário, o que será iniciado com a definição do subdiretório view, a partir de cadastro. Nosso primeiro arquivo no subdiretório será **CommonStyles.js**, com a listagem seguinte, em que teremos a concentração de todos os estilos utilizados no aplicativo.

Terminada a definição dos estilos, começaremos a implementação das telas criando o arquivo **AlunoTela.js**, que será utilizado para listagem, inclusão e exclusão de alunos.

Javascript

```
import React, {useState, useEffect} from 'react';
```

```
import { Text, View, TextInput, TouchableOpacity, FlatList }
       from 'react-native';
import { styles } from './CommonStyles';
import { useIsFocused } from '@react-navigation/native';
import AlunoController from '../controller/AlunoController';
import AlunoItem from './AlunoItem';
import { Aluno } from '../model/Aluno';

export default function AlunoTela( { navigation } ) {
  const controller = new AlunoController();
  const [alunos, setAlunos] = useState([]);
  const [matricula, setMatricula] = useState('');
  const [nome, setNome] = useState('');
```

Nossa tela de alunos é constituída de dois componentes TextInput, funcionando como entrada para matrícula e nome, um botão do tipo TouchableOpacity, para adicionar um aluno com os dados digitados, e um componente FlatList, para a listagem de alunos.

Devemos observar as variáveis de estado matricula e nome, ambas do tipo string, que são associadas aos componentes TextInput, além de alunos, o vetor que funciona como origem de dados do FlatList, através do atributo data. Como o campo identificador de Aluno é a matricula, também precisaremos de uma função para extração da chave, com o nome myKeyExtractor, associada ao atributo keyExtractor do FlatList.

> **O método excluirAluno aciona o método excluir do controlador, e na sequência temos a chamada para obterTodos, com a definição dos valores da lista. O preenchimento dos valores também é acionado quando a tela obtém o foco.**

Temos um comportamento similar no método adicionarAluno, em que ocorre a chamada para o método incluir, do controlador, com a passagem de um objeto Aluno, contendo os dados digitados na tela. Na sequência, apenas quando ocorre um retorno verdadeiro, temos a limpeza dos campos, a partir das variáveis de estado, e a atualização da lista.

Ainda precisamos definir o componente para visualização do item de lista, com o nome **AlunoItem.js**, que deverá ficar no subdiretório view, contendo a listagem seguinte.

Javascript

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

 export default function AlunoItem(props){
   const fillZero = (num) => { return ((num<10) ? '0':'')+num; }
   const getStrDate = (data) => {
     return fillZero(data.getDate())+'/'+
            fillZero(data.getMonth()+1)+'/'+data.getFullYear();
   }
   return (
     <View style={styles.container} id={props.aluno.matricula}>
       <Text style={styles.textItem}>
         {props.aluno.matricula} - {props.aluno.nome}</Text>
```

Em termos de componentes nativos, temos apenas dois elementos do tipo Text para exibir os atributos do aluno, fornecido por meio das propriedades (props) da tag, além de um botão que irá acionar a função onDelete, também fornecida pelas props, para excluir o aluno na ocorrência de um clique. As propriedades são associadas ao nível do componente FlatList, no atributo renderItem, com aluno recebendo o item corrente, e onDelete sendo relacionado ao método excluirAluno, com a passagem da matrícula.

Foram definidas duas funções utilitárias, uma para preenchimento de zeros à esquerda, nos números menores que dez, e outra para o retorno da data na forma de texto. Elas serão necessárias para a correta exibição da data de registro do aluno.

Com o objetivo de deixar o sistema mais flexível, permitindo incluir outras entidades, vamos criar uma tela de menu, no arquivo **MenuTela.js**.

Javascript

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

 export default function MenuTela( { navigation } ){
   return (
     <View style={styles.container}>
       <TouchableOpacity style={styles.button}
            onPress={()=>navigation.navigate('AlunoTela')}>
         <Text style={styles.buttonTextBig}>Alunos</Text>
       </TouchableOpacity>
     </View>
   );
 }
```

Como temos apenas um tipo de entidade, utilizamos apenas um botão, contendo o texto Alunos, que responderá ao clique com a navegação para a tela de alunos (AlunoTela).

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

Javascript

```
import {NavigationContainer} from '@react-navigation/native';
 import {createStackNavigator} from '@react-navigation/stack';
 import React from 'react';
 import AlunoTela from './cadastro/view/AlunoTela';
 import MenuTela from './cadastro/view/MenuTela';

 const Stack = createStackNavigator();

 export default function App() {
   return (
     <NavigationContainer>
       <Stack.Navigator initialRouteName='MenuTela'>
         <Stack.Screen name='MenuTela'
          options={{title: 'Menu Principal'}}
```

Temos uma navegação constituída de duas telas, MenuTela e AlunoTela, utilizando os componentes de mesmo nome. A navegação é do tipo pilha (Stack), sendo configurados os títulos das janelas por meio de options, no atributo title.

Com tudo resolvido, podemos executar nosso projeto, onde será exibido inicialmente o menu, permitindo o acesso à tela de alunos a partir do clique no botão.

Telas do aplicativo de exemplo.

Arquitetura MVC no React Native

No vídeo a seguir, exemplificamos a construção de aplicativo usando a arquitetura MVC no React Native.

Vem que eu te explico!

Os vídeos a seguir abordam os assuntos mais relevantes do conteúdo que você acabou de estudar.

Padrões de desenvolvimento X Padrões arquiteturais

3:34min

Padrão DAO no React Native

4:56min

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

A arquitetura MVC é considerada um padrão de grande relevância no desenvolvimento de aplicativos cadastrais, tanto em sistemas Web quanto em desktop ou móveis. Segundo o padrão arquitetural, que divide o aplicativo em três camadas bem definidas, qual camada deveria conter os componentes do tipo DAO?

A Controller

B Model

C Presentation

D View

E Dispatcher

Responder

Questão 2

Quando acessamos um banco de dados a partir de alguma linguagem de programação, é fácil observarmos processos bastante repetitivos, em que as mudanças acabam ocorrendo em pontos específicos, relacionados ao mapeamento da entidade ou comando SQL utilizado para cada operação. Podemos aproveitar toda a parte comum dos processos, e deixar que sejam programadas apenas as especificidades para cada tabela, com a adoção do padrão

A Facade

B Proxy

C Abstract Facade.

D Composite

E Template Method.

Responder

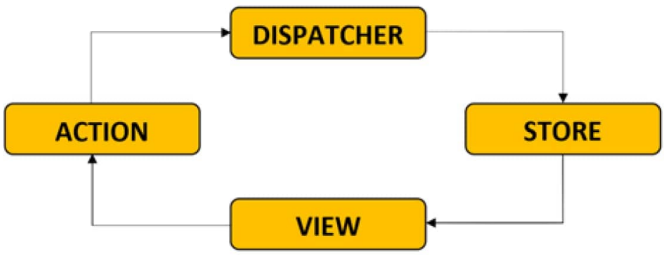
★★★★★

2 - Arquiteturas Flux e Redux com React Native

Ao final deste módulo, você será capaz de empregar arquiteturas Flux e Redux com React Native.

Arquitetura Flux

Flux é a arquitetura utilizada pelo Facebook na construção de clientes Web, tendo como objetivo um fluxo unidirecional de dados. Basicamente, uma tela (View) inicia uma ação (Action) que, por meio de um Dispatcher, inicia um processo no sistema de armazenagem (Store), atualizando a tela inicial, como pode ser observado na figura apresentada a seguir.



Arquitetura Flux e seus componentes.

Enquanto no MVC temos a lógica de negócios definida a partir do Controller, no Flux temos a classe Store assumindo a responsabilidade sobre as regras de negócio. Porém, ao contrário da natureza síncrona do MVC, temos no Flux uma arquitetura voltada para comportamento totalmente assíncrono.

> **No componente Store temos os dados e a lógica da aplicação. Por exemplo, uma tela poderia obter a lista de alunos a partir de uma classe Store apropriada, e a mesma classe deve agrupar os métodos para manipulação dos registros de alunos, o que não impede que seja utilizada em conjunto com um componente no padrão DAO.**

Com base no padrão Observer, quando os dados são atualizados, por meio dos métodos da Store, devemos ter a atualização automática da informação em todo o conjunto de telas associadas. Essa característica garante a consistência dos dados em todo o sistema, não importando de qual tela foi disparada uma alteração qualquer.

Os componentes nativos do React Native determinarão as classes View, ou telas, de forma equivalente à camada View do MVC. Em outras palavras, temos novamente um isolamento da interface gráfica, garantindo o reuso simplificado das regras de negócio.

Comentário

Todas as solicitações feitas à classe Store devem ser coordenadas por meio de um canal central, responsável por despachar as mensagens necessárias, onde entram em ação os componentes do tipo Dispatcher. Em termos práticos, ele não é um canal inteligente, pois simplesmente recebe as solicitações e repassa para a classe Store.

Finalmente, temos as ações, com as classes do tipo Action, visando à padronização da comunicação entre as telas e o Dispatcher. Nossas telas deverão iniciar ações, que delegam para o Dispatcher a responsabilidade de se comunicar com a Store, onde serão executados os processos de atualização dos dados, refletindo nas telas associadas.

A tabela seguinte apresenta um resumo das responsabilidades de cada componente em uma arquitetura Flux.

Componente	Descrição
View	Interface do usuário, com eventos que iniciam Actions.
Action	Pacote de comunicação enviado para o Dispatcher.
Dispatcher	Repassa as solicitações para todas as Stores registradas.
Store	Recebe as solicitações, executa os processos requeridos sobre os dados, e emite eventos para a atualização das Views.

Quadro: Componentes da arquitetura Flux.
Elaborado por: Denis Cople.

Ao adotar a arquitetura Flux, eliminamos os relacionamentos bidirecionais do MVC, além de ser uma solução muito escalonável, aplicável a sistemas com grande crescimento.

Cadastro com Flux

Criaremos um cadastro na arquitetura Flux, aproveitando alguns dos componentes do MVC, executando no modelo nativo. Inicialmente, vamos configurar nosso projeto.

Terminal

```
react-native init CadastroFlux
 cd CadastroFlux
 npm install react-native-sqlite-storage @types/react-native-sqlite-storage @types/react --save
 npm install flux @types/flux --save
 npm install events --save
 npm install @react-navigation/native @react-navigation/stack
 npm install react-native-reanimated react-native-gesture-handler react-native-screens react-native-safe
```

Copiaremos, do exemplo de MVC, o arquivo App.js, além dos diretórios cadastro/model e cadastro/view. A camada Model será mantida na forma original, mas precisaremos de leves alterações na camada View, algo que será visto posteriormente neste tópico.

Agora podemos nos preocupar com o Flux, iniciando com a construção do Dispatcher, que é o passo mais simples. Crie um subdiretório com o nome dispatcher, a partir de cadastro, e dentro dele o arquivo **AlunoDispatcher.ts**.

Javascript

```
import { Dispatcher } from ´flux´;

 const alunoDispatcher = new Dispatcher();

 export default alunoDispatcher;
```

Como podemos observar, foi necessária apenas uma instância global de Dispatcher, com o nome alunoDispatcher, a qual será importada pelos demais componentes do Flux.

Definido nosso canal de comunicação, vamos criar um subdiretório action, a partir de cadastro, e dentro dele o arquivo **AlunoAction.ts**.

Javascript

```
import alunoDispatcher from '../dispatcher/AlunoDispatcher';
 import { Aluno } from '../model/Aluno';

 export default class AlunoActions{
   public criarAluno(aluno: Aluno) {
     alunoDispatcher.dispatch({
       actionType: 'CRIAR_ALUNO',
       value: aluno
     });
     alunoDispatcher.dispatch({
       actionType: 'OBTER_ALUNOS'
     });
```

```
13      }
14      public excluirAluno(matricula: string) {
```

A classe AlunoActions irá encapsular todas as ações referentes à gerência de alunos, ou seja, organizará as chamadas para alunoDispatcher. Nossa classe segue o padrão de desenvolvimento Singleton, com a utilização de um construtor privado, uma instância privada estática, e um método estático para obtenção da instância.

> **Os métodos de AlunoActions visam apenas ao envio das mensagens corretas para o Storage, por meio do método dispatch, de alunoDispatcher. Cada mensagem apresenta uma identificação, no atributo actionType, podendo acrescentar algum valor de parâmetro no atributo value.**

Analisando o método criarAluno, enviamos uma mensagem do tipo CRIAR_ALUNO, com value recebendo a instância de Aluno que será adicionada. Em seguida, enviamos uma mensagem OBTER_ALUNOS, para provocar a atualização dos valores nas telas.

Algo similar ocorre para excluirAluno, mas a mensagem inicial é EXCLUIR_ALUNO, com a passagem da chave. Já no método obterAlunos, temos apenas a mensagem OBTER_ALUNOS, para atualização das telas.

Nosso próximo passo será a criação do arquivo AlunoStore.ts, em um subdiretório store, a partir de cadastro, contendo a listagem seguinte.

Javascript

```
1  import alunoDispatcher from '../dispatcher/AlunoDispatcher';
2   import { Aluno } from '../model/Aluno';
3   import AlunoDAO from '../model/AlunoDAO';
4   import { EventEmitter } from 'events';
5
6   export default class AlunoStore extends EventEmitter {
7     private dao = new AlunoDAO();
8     private alunos: Aluno[] = [];
9     public getAlunos(): Aluno[]{ return this.alunos; }
10
11    public addChangeListener(callback:()=>void) {
12      this.on('ALUNOS_CHANGE', callback);
13    }
14    public removeChangeListener(callback:()=>void) {
```

Utilizamos novamente o padrão Singleton, na definição da classe AlunoStore, derivada de EventEmitter. Note a necessidade de registro do método dispatcherCallback, a partir de alunoDispatcher, para que as solicitações sejam interceptadas e tratadas.

Atenção

O tratamento das solicitações é simples, pois tudo que temos em dispatcherCallback é o reconhecimento de actionType, e acionamento do método correto, a partir de uma estrutura switch, com a passagem de value para o método que foi acionado, quando disponível.

Os componentes da View são avisados das mudanças no conjunto de dados por meio de eventos personalizados, e o registro do componente ouvinte ocorrerá a partir de addChangeListener, com a passagem da callback que será iniciada pelo evento. A callback é associada pelo método on, de EventEmmiter, em que os parâmetros são a mensagem ALUNOS_CHANGE e o método de resposta.

Da mesma forma que precisamos registrar um componente como ouvinte, devemos ser capazes de interromper o recebimento das mensagens. A desassociação é feito pela invocação do método removeChangeListener, em que ocorre a chamada para removeListener.

Nos métodos incluirAluno e excluirAluno apenas repassamos a solicitação para o DAO, invocando os métodos incluir e excluir, respectivamente. Já no método obterAlunos, invocamos o método obterTodos do DAO, e associamos o resultado à coleção interna da Store, emitindo a mensagem ALUNOS_CHANGE, na sequência, para notificar as interfaces registradas.

Como passo final, precisamos efetuar uma leve modificação em AlunoTela.js, para que adote os componentes do Flux.

Javascript

```
import React, {useState, useEffect} from 'react';
 import {Text, View, TextInput, TouchableOpacity, FlatList}
        from 'react-native';
 import {styles} from './CommonStyles';
 import AlunoItem from './AlunoItem';
 import {Aluno} from '../model/Aluno';
 import AlunoStore from '../store/AlunoStore';
 import AlunoActions from '../action/AlunoActions';

 export default function AlunoTela( { navigation } ) {
   const [alunos, setAlunos] = useState([]);
   const [matricula, setMatricula] = useState('');
   const [nome,setNome] = useState('');
   const [listenerAdded, setListenerAdded] = useState(false);
```

Além da modificação em algumas importações, temos a remoção do controller, que não é mais utilizado, e a adição de uma variável de estado com o nome listenerAdded, utilizada para evitar associações recursivas com a Store.

Na implementação de useEffect, associamos o listener à Store, por meio de addChangeListener, fornecendo uma callback na qual a coleção de alunos é obtida, via getAlunos, e utilizada para preencher a coleção interna da tela. Na sequência, modificamos o valor de listenerAdded para true, e temos a chamada para obterAlunos, de AlunoActions, causando a atualização inicial.

Atenção

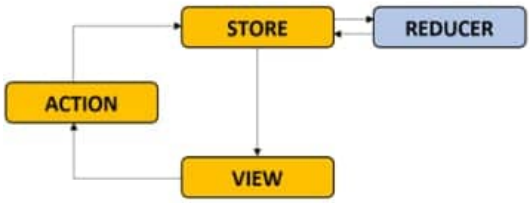
A implementação das funções excluirAluno e adicionarAluno se tornou mais concisa, com o simples acionamento dos métodos apropriados na classe AlunoActions, e como sempre ocorre o despacho de uma mensagem de atualização ao final, os dados são consultados, e o evento personalizado ALUNOS_CHANGE acontece, garantindo a atualização da lista. Na função utilizada para inclusão, ainda temos a limpeza dos campos, como ocorria anteriormente.

Nada mais precisa ser modificado na camada View, e já podemos executar o aplicativo, obtendo as mesmas funcionalidades, mas agora com fluxo unidirecional.

Arquitetura Redux

Uma alternativa ao Flux é a arquitetura que ficou conhecida como Redux. Embora siga princípios equivalentes, não temos a figura do Dispatcher, e a manipulação dos estados do sistema ocorre pelas funções Reducer. A arquitetura Redux tem uma abordagem funcional, e a criação de classes para definir ações não traria benefícios. Toda ação será definida em termos de uma função, com retorno de um objeto, contendo o tipo de mensagem no atributo type e os dados opcionais em payload.

Para o Redux, uma classe Store gerencia os estados do sistema de forma centralizada, e qualquer manipulação deve ocorrer por meio de funções puras, as quais irão constituir o grupo classificado como Reducer. Além de armazenar os estados, a classe Store deve notificar mudanças ocorridas para os componentes visuais, que foram associados via assinatura (subscribe).



Arquitetura Redux e seus componentes.

Saiba mais

Um Reducer funciona como filtro de ações, com o processamento da solicitação, a partir de seu tipo, efetuando as modificações necessárias sobre os dados, e retornando o estado modificado.

O componente View deve assinar uma Store, e a partir dela despachar uma Action, para que seja interceptada e tratada no Reducer, podendo alterar o estado do aplicativo e forçando a Store a notificar uma atualização para a View.

A simplicidade da arquitetura, o tamanho reduzido da biblioteca e a qualidade da documentação, levaram o Redux a ganhar popularidade rapidamente. Ele gerencia o estado do aplicativo com uma única árvore de estados imutável, que só pode ser alterada por meio de ações e redutores.

Entre as vantagens do Redux, podemos destacar a previsibilidade da saída, visto que temos apenas uma fonte de informação, facilidade para a manutenção do sistema, ambiente propício para os testes unitários, comunidade de desenvolvedores bem estruturada e melhor organização do código. Embora muitas vantagens já fossem proporcionadas pelo Flux, no Redux teremos a implementação simplificada, descartando o Dispatcher e os eventos personalizados.

Cadastro com Redux

Para nosso exemplo de Redux, utilizaremos o banco de dados Realm, com funcionalidades mais adequadas à arquitetura. Inicialmente, criaremos o projeto, no modelo nativo do React Native, e efetuaremos as importações necessárias.

Terminal

```
react-native init CadastroRedux
 cd CadastroRedux
 npm install realm --save
 npm install redux react-redux --save
 npm install @types/redux @types/react-redux --save
 npm install @react-navigation/native @react-navigation/stack
 npm install react-native-reanimated react-native-gesture-handler react-native-screens react-native-safe
```

Vamos reaproveitar alguns componentes do exemplo MVC, com leves alterações. Precisamos copiar o diretório cadastro/view, além do arquivo Aluno.ts, no diretório cadastro/model, e o arquivo **App.js**. Como utilizaremos Realm, apenas a entidade será aproveitada.

Agora, vamos criar o arquivo **DatabaseInstance.js**, no subdiretório model, para a gerência da conexão com o banco de dados.

Javascript

```
import Realm from 'realm';

 var db = new Realm({
   path: 'EscolaDB.realm',
   schema: [
     {
     name: 'Aluno', primaryKey: 'matricula',
     properties:
       {matricula: 'string', nome: 'string', registro: 'int'}
     }
   ]
 });

 export default db;
```

Temos uma conexão com o Realm, na variável db, apontando para o arquivo **EscolaDB.realm**, e tendo no esquema do banco apenas a classe Aluno. Com a conexão definida, vamos criar o arquivo **AlunoDAO.ts**, também no subdiretório model, contendo a listagem seguinte.

Javascript

Javascript

```
import { Aluno } from './Aluno';
 import db from './DatabaseInstance';

 export default class AlunoDAO {
   public incluir(entidade: Aluno) {
     let criacao = new Date();
     let aluno = {matricula: entidade.matricula,
                  nome: entidade.nome,
                  registro: criacao.getTime()}
     db.write(() => db.create('Aluno', aluno));
   }
   public excluir(chave: string) {
     db.write(() => db.delete(
        db.objects('Aluno').filtered('matricula = $0',chave))
```

Nossa classe DAO envolve apenas as ações de inclusão, exclusão e consulta geral, sendo necessário utilizar a transação de escrita para modificações, via método write. Para o método incluir, a data corrente é capturada, montamos um objeto JSON com os dados necessários, e adicionamos o objeto na coleção Aluno, enquanto o método excluir faz a filtragem da coleção pela matrícula, e invoca o método delete para o objeto retornado.

A consulta, feita por meio de obterTodos, recupera todos os elementos da coleção Aluno, e transforma cada um na entidade correta, com a chamada para getAluno, antes de adicionar ao vetor de retorno.

Agora, criaremos o arquivo **AlunoActions.ts**, no diretório cadastro/action, contendo a listagem seguinte.

Javascript

```
import {Aluno} from '../model/Aluno';

 export const CRIAR_ALUNO   = 'CRIAR_ALUNO';
 export const EXCLUIR_ALUNO = 'EXCLUIR_ALUNO';
 export const OBTER_ALUNOS  = 'OBTER_ALUNOS';

 export const actionCriarAluno = (aluno: Aluno) => ({
   type: CRIAR_ALUNO,
   payload: {aluno}
 })

 export const actionExcluirAluno = (matricula: string) => ({
   type: EXCLUIR_ALUNO,
   payload: {matricula}
```

Observe a abordagem funcional, em que temos as ações CRIAR_ALUNO, EXCLUIR_ALUNO e OBTER_ALUNOS, e as funções necessárias para o encapsulamento delas. Em cada função ocorre o retorno de um objeto, em que o atributo type recebe a ação, e payload os parâmetros.

O próximo componente é o mais complexo, referente ao Reducer. Iremos criar um arquivo **AlunoReducer.ts**, no diretório cadastro/reducer, com o código apresentado a seguir.

Javascript

```
import {CRIAR_ALUNO, EXCLUIR_ALUNO, OBTER_ALUNOS}
        from '../action/AlunoActions';
 import AlunoDAO from '../model/AlunoDAO';

 const dao: AlunoDAO = new AlunoDAO();

 export const reducer = (state = [],
              action: {type: string, payload: any}) => {
   switch(action.type) {
     case CRIAR_ALUNO:
       dao.incluir(action.payload.aluno);
       state = dao.obterTodos();
       break;
     case EXCLUIR ALUNO:
```

Definimos uma função reducer, tendo como parâmetro o estado inicial, em state, inicializado com um vetor vazio, e uma ação, composta de type e payload. Podemos observar as chamadas efetuadas ao DAO, de acordo com o tipo da ação, e o retorno do estado modificado ao final.

Para uma ação do tipo CRIAR_ALUNO, chamamos o método incluir, com a passagem do aluno fornecido no payload, enquanto na ação EXCLUIR_ALUNO vemos a chamada para o método excluir, com a passagem da matrícula. Tanto para as duas ações anteriores quanto na ação OBTER_ALUNOS, o estado final recebe a consulta geral sobre a coleção.

No passo seguinte, vamos criar o arquivo AlunoStore.ts, no diretório cadastro/store.

Javascript

```
import {createStore} from 'redux';
 import {reducer} from '../reducer/AlunoReducer';

 export const store = createStore(reducer);
```

Note a simplicidade na criação do objeto store, com a chamada para createStore, tendo a função reducer como parâmetro. O objeto é exportado, e será utilizado para gerenciar os estados de qualquer componente visual associado via subscribe.

Na camada View teremos apenas a alteração do arquivo **AlunoTela.js**.

Javascript

Javascript

```
import React, {useState, useEffect} from 'react';
 import {Text, View, TextInput, TouchableOpacity, FlatList}
        from 'react-native';
 import {styles} from './CommonStyles';
 import AlunoItem from './AlunoItem';
 import {Aluno} from '../model/Aluno';
 import {actionCriarAluno, actionExcluirAluno,
        actionObterAlunos} from '../action/AlunoActions';
 import {store} from '../store/AlunoStore';

 export default function AlunoTela( { navigation } ) {
   const [alunos,setAlunos] = useState([]);
   const [matricula,setMatricula] = useState('');
   const [nome,setNome] = useState('');
```

A alteração efetuada é muito similar à do Flux, pois temos para a inclusão e para a exclusão o despacho de ações, agora representadas pelas funções actionCriarAluno e actionExcluirAluno. Note que não temos um Dispatcher, mas sim o método dispatch do objeto store.

Na implementação de useEffect, efetuamos a assinatura, por meio de subscribe, fornecendo uma callback que associa a coleção interna da tela à coleção do objeto store, obtida por meio de getState. Temos, na sequência, a modificação do valor de listenerAdded para true, e despacho da ação OBTER_ALUNOS, causando a atualização inicial da lista.

Já podemos executar nosso cadastro, agora na arquitetura Redux.

Emprego do React Native em aplicações baseadas em fluxo

No vídeo a seguir, abordamos o uso das arquiteturas Flux e Redux no React Native em aplicações baseadas em fluxo.

Vem que eu te explico!

Os vídeos a seguir abordam os assuntos mais relevantes do conteúdo que você acabou de estudar.

Arquitetura Flux

4:32min

Arquitetura Redux

4:01min

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Embora a arquitetura MVC forneça uma boa organização para sistemas cadastrais, o que fica evidente com sua grande aceitação no mercado, uma arquitetura baseada em fluxo de dados unidirecional é mais adequada ao React Native. Um exemplo de arquitetura com fluxo unidirecional é o Flux, em que o componente Dispatcher se torna responsável por

A repassar as solicitações para os componentes Store.

B atualizar as informações da base de dados.

C iniciar as ações do sistema.

D efetuar as consultas na base de dados.

E exibir as informações para o usuário.

Responder

Questão 2

Embora a arquitetura Flux, criada pelo Facebook, seja muito eficiente para a garantia de um fluxo unidirecional de execução, o Redux trouxe um modelo mais simples e adequado ao React Native. Qual das opções não representa uma característica do Redux?

A Garante um fluxo unidirecional de execução.

B A manipulação de estados ocorre a partir de funções Reducer.

C Precisa de uma estrutura de eventos personalizados.

D Um componente View representa a interface de usuário.

E Utiliza um modelo de programação predominantemente funcional.

Responder

★★★★★

3 - Criptografia com React Native

Ao final deste módulo, você será capaz de aplicar criptografia na persistência e no controle de acesso com React Native.

Fundamentos de criptografia

> **A palavra codificação significa transformação ou modificação de formato, como na tradução de um algoritmo em uma linguagem de programação. Um dos exemplos mais relevantes, para o armazenamento e a transmissão de dados, é a codificação para Base64.**

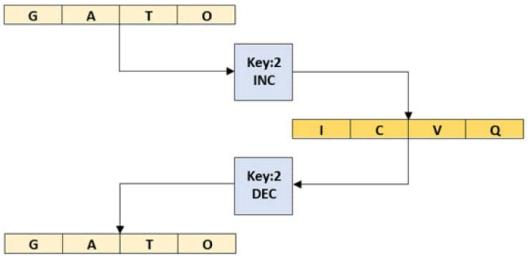
Qualquer codificação envolve uma função de transformação reversível, para a obtenção da representação desejada, em que a recuperação do dado original é conhecida como decodificação. Com a biblioteca react-native-base64, obtemos um objeto de nome base64, sendo utilizados os métodos encode e decode, respectivamente, para a codificação e decodificação.

Javascript

```
1  import base64 from 'react-native-base64';
2
3   let texto = 'APENAS UM TESTE';
4   let txtb64 = base64.encode(texto);
5   let txtdec = base64.decode(txtb64);
6   console.log(txtb64); //QVBFTkFTIFVNIFRFU1RF
```

```
7   console.log(txtdec); //APENAS UM TESTE
```

Enquanto a codificação envolve a utilização de operações matemáticas reversíveis de forma direta, na criptografia precisamos de uma chave para efetuar as transformações, o que garante o sigilo. Um exemplo seria o deslocamento de caracteres na tabela ASCII, cuja chave seria o número de posições adicionadas.



Exemplo de criptografia extremamente simples.

Na criptografia simétrica temos uma mesma chave para efetuar tanto a criptografia quanto a decriptografia. A chave é conhecida como Secret Key, e temos opções como 3DES, AES e RC4.

Recomendação

Quando utilizamos um par de chaves, temos a criptografia assimétrica, em que as chaves costumam ser geradas a partir de grandes números primos. Enquanto a chave privada é armazenada de forma segura, a chave pública é distribuída, permitindo o envio de informações criptografados por qualquer pessoa, mas que só poderão ser abertas pela chave privada do destinatário. No caminho contrário, apenas a chave privada é capaz de assinar um documento, enquanto a chave pública permite conferir a assinatura.

A utilização do algoritmo assimétrico RS é comum no mercado corporativo, e as chaves públicas costumam ser disponibilizadas em uma árvore LDAP.

Ainda temos a criptografia destrutiva, também conhecida como hash, onde ocorre a perda de fragmentos dos dados originais, impedindo a decriptografia, o que a torna útil para guarda de senhas. Em termos práticos, quando o usuário digita a senha novamente, ela é criptografada e comparada com o valor armazenado no banco, também criptografado, já que não será possível recuperar o valor original, tendo como exemplos comuns os algoritmos MD5 e SHA1.

Criptografia no React Native

Existem diversas bibliotecas de criptografia para React Native, mas uma das mais conhecidas é a react-native-crypto-js. Ela disponibiliza algoritmos para hash, como MD5 e SHA1, e algumas opções para criptografia simétrica, como AES, 3DES e RC4.

> **Um dos aspectos mais frágeis da criptografia é a geração de chaves, pois o tamanho da chave e a qualidade dos números aleatórios serão essenciais para dificultar a quebra por ataques de força bruta. É necessário definir o vetor de inicialização, chamado de IV, que funciona como a semente para randomização, além de uma sequência de bytes gerada aleatoriamente, conhecida como SALT, a qual será anexada ao resultado do hash.**

Vamos começar a definir nossa biblioteca de criptografia com uma classe para representar a chave simétrica, no arquivo **StoredKey.ts**. Na classe, teremos os campos iv e key, com texto em formato Base64, guardando o vetor de inicialização e a chave.

Javascript

```
export default class StoredKey {
     public iv: string;
     public key: string;
 }
```

Ao lidar com elementos binários é comum a diferença de formato entre bibliotecas, o que irá ocorrer entre o codificador Base64, que trabalha com 8 bits, e o algoritmo de criptografia AES, com representação de 32 bits. Para resolver o problema de uma forma reutilizável, criaremos o arquivo **Utils.ts**, com a listagem seguinte.

Javascript

```
import CryptoJS from 'react-native-crypto-js';

 export default class Utils {
   public static getByteArray(valoresWord: any): Uint8Array{
     const buffer = new ArrayBuffer(valoresWord.sigBytes);
     const bufferI32 = new Int32Array(buffer);
     bufferI32.map((valor,index,array)=>
               array[index]=valoresWord.words[index]);
     return new Uint8Array(buffer);
     }
   public static getWordArray(valores: Uint8Array): any{
     const buffer = new ArrayBuffer(valores.length);
     const bufferUI8 = new Uint8Array(buffer);
```

```
14        buffonUI8.map((valor,index,array)=>
```

No método getByteArray, recebemos um elemento do tipo WordArray, com valores de 32 bits, e retornamos um vetor do tipo Uint8Array. Para viabilizar a divisão dos 32 bits em quatro posições de 8 bits, utilizamos um buffer genérico, que é compartilhado entre os dois tipos de vetor, reinterpretando a informação.

Um processo similar ocorre em getWordArray, em que temos um vetor com valores de 8 bits, do tipo Uint8Array, e retornamos um vetor do tipo WordArray.

Atenção

O preenchimento de buffers com uso de map permite efetuar operações sobre cada elemento do vetor de forma direta, segundo o paradigma funcional.

O método getWordArrayS foi criado apenas para se adequar à forma de trabalho da biblioteca de criptografia, em que o elemento do tipo WordArray é exportado como texto, sendo necessário converter para Uint8Array, com a subsequente chamada para getWordArray.

Criaremos nosso gerador de chaves no arquivo **KeyGenerator.ts**, cujas chaves criptográficas devem ser geradas com boa randomização e armazenamento seguro.

Javascript

```
1  import CryptoJS from 'react-native-crypto-js';
2   import EncryptedStorage from 'react-native-encrypted-storage';
3   import base64 from 'react-native-base64';
4   import StoredKey from './StoredKey';
5   import Utils from './Utils';
6
7   export default class KeyGenerator {
8     public async restoreKey(nome: string): Promise<StoredKey>{
9       try {
10        const chave = await EncryptedStorage.getItem(nome);
11        return JSON.parse(chave);
12      } catch {return null;}
13    }
14
```

O componente EncyptedStorage funciona de forma similar ao AsyncStorage, mas os valores armazenados são criptografados, de forma transparente, com base nas chaves armazenadas pela Key Store do aplicativo. Quando executado no ambiente Android, um componente do tipo EncryptedSharedPreferences é acionado, enquanto no iOS temos o Keychain.

Ao criar a chave, em createKey, com a passagem do nome da chave, adotamos um processo totalmente aleatório para geração

dos parâmetros iv e salt, invocando o método random, de WordArray. Em seguida, uma chave compatível com AES é gerada, por meio do método EvpKDF, utilizando mil iterações randômicas.

Na sequência, temos a criação de um StoredKey, com os valores codificados para Base64, e, ao final do bloco protegido, gravamos o objeto de forma segura, com EncryptedStorage, para que possa ser recuperado posteriormente no método restoreKey.

A utilização do gerador de chaves pode ser observada no fragmento de código seguinte, com base no operador then para controle do fluxo. Tentamos recuperar a chave, e, caso ela não exista, criamos uma, o que fará com que seja reconhecida em uma execução subsequente.

Javascript

```
var strKey;
 var generator = new KeyGenerator();
 generator.restoreKey('PWDKEY8734$_AL').then((valor) => {
   if(valor!=null)
     strKey = valor;
   else{
     console.log('CHAVE INEXISTENTE');
     generator.createKey('PWDKEY8734$_AL').then(
                                  (valor) => strKey=valor);
   }
 });
```

Com a chave gerada, podemos efetuar as ações de criptografia, o que faremos por meio da classe Cipher, que deverá ser criada no arquivo **Cipher.ts**.

Javascript

```
import base64 from 'react-native-base64';
 import CryptoJS from 'react-native-crypto-js';
 import StoredKey from './StoredKey';
 import Utils from './Utils';

 export default class Cipher {
  public criptografar(strKey: StoredKey,
                      texto: string): string{
    const iv = Utils.getWordArrayS(base64.decode(strKey.iv));
    const key = Utils.getWordArrayS(base64.decode(strKey.key));
    const textoCripto = CryptoJS.AES.encrypt(
                                    texto,key,{iv: iv});
    return base64.encode(textoCripto.toString());
```

O método criptografar recebe os valores iv e key, a partir de um StoredKey, os quais devem ser decodificados e convertidos para o formato da biblioteca de criptografia. Em seguida, invocamos o método encrypt, no algoritmo AES, passando o texto fornecido nos parâmetros, a chave e o vetor de inicialização, e retornamos o valor criptografado, convertido para Base64.

Um processo semelhante é adotado no método decriptografar, onde já sabemos que o texto criptografado está em Base64, devendo também ser decodificado, juntamente aos atributos da StoredKey. O método decrypt é invocado e o resultado é convertido para texto plano antes de retornar o valor ao chamador.

O fragmento de código seguinte apresenta um exemplo de utilização para Cipher.

Javascript

```
function executar(strKey){
      var cipher = new Cipher();
      var mensagem = 'APENAS UM TESTE';
      var textoCripto = cipher.criptografar(strKey,mensagem);
      var textoDecripto = cipher.decriptografar(
                                  strKey,textoCripto);
      console.log(textoCripto);
      console.log(textoDecripto);
 }
```

Cadastro criptografado

Vamos modificar nosso cadastro inicial, na arquitetura MVC, incorporando elementos de criptografia. A configuração do novo projeto é apresentada a seguir.

Terminal

```
react-native init CadastroCripto
 cd CadastroCripto
 npm install react-native-sqlite-storage @types/react-native-sqlite-storage @types/react --save
 npm install react-native-base64 --save
 npm install react-native-crypto-js --save
 npm install react-native-encrypted-storage --save
 npm install @react-navigation/native @react-navigation/stack
 npm install react-native-reanimated react-native-gesture-handler react-native-screens react-native-safe
```

Copiaremos o arquivo **App.js** e o diretório cadastro, com todos os seus subdiretórios e arquivos, para o novo projeto. Também precisamos criar um subdiretório cripto, a partir de cadastro, onde serão colocados os arquivos **StoredKey.ts**, **Utils.ts**, **KeyGenerator.ts** e **Cipher.ts**.

Agora, vamos modificar a forma de gravação do campo nome, para que trabalhe com os novos recursos criptográficos desenvolvidos no tópico anterior. A modificação ocorrerá de forma pontual, na classe AlunoDAO, de acordo com a listagem seguinte.

Javascript

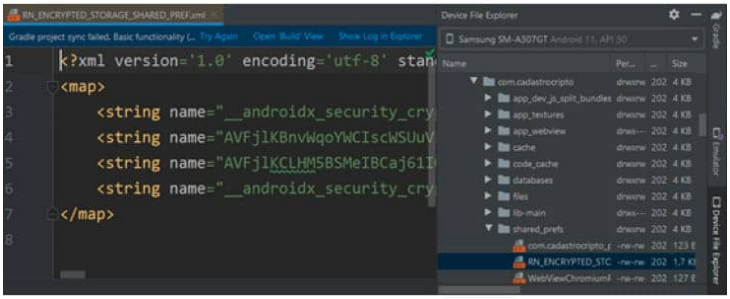
```
import { Aluno } from './Aluno';
 import { GenericDAO } from './GenericDAO';
 import Cipher from '../cripto/Cipher';
 import StoredKey from '../cripto/StoredKey';
 import KeyGenerator from '../cripto/KeyGenerator'

 export default class AlunoDAO extends GenericDAO<Aluno,string>{
   protected getCreateSQL(): string {
     return ' CREATE TABLE ALUNO (MATRICULA VARCHAR(10) '+
            ' PRIMARY KEY,NOME VARCHAR(20),REGISTRO INTEGER)';
   }
   protected getTableName(): string {
     return 'ALUNO';
   }
```

O campo nome deve ser criptografado nos métodos getInsertParams e getUpdateParams, bem como na recuperação da entidade, em getEntidade, com a decriptografia do nome. Efetuando os processos ao nível do DAO, estamos garantindo que o mapeamento objeto-relacional implemente a criptografia, deixando o processo transparente para as demais camadas.

Claro que precisaremos de uma chave armazenada e um objeto do tipo Cipher, para dar suporte às ações relacionadas à criptografia. Da mesma forma que garantimos a criação da tabela no construtor, aqui acrescentamos a criação ou recuperação da chave, para que ela esteja disponível durante todo o ciclo de vida do DAO.

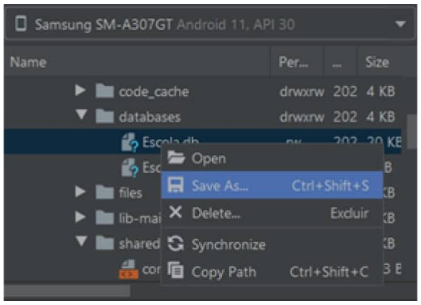
No Android Studio, painel Device File Explorer, podemos visualizar os dados criptografados.



Captura de tela do Android Studio - Acesso ao arquivo de preferências.

Navegando na árvore de diretórios do dispositivo, os arquivos locais estarão no pacote **com.cadastrocripto**, a partir de data/data, e no subdiretório shared_prefs poderemos clicar sobre o arquivo **RN_ENCRYPTED_STORAGE_SHARED_PREF**, abrindo no editor os valores das chaves armazenados de forma criptografada.

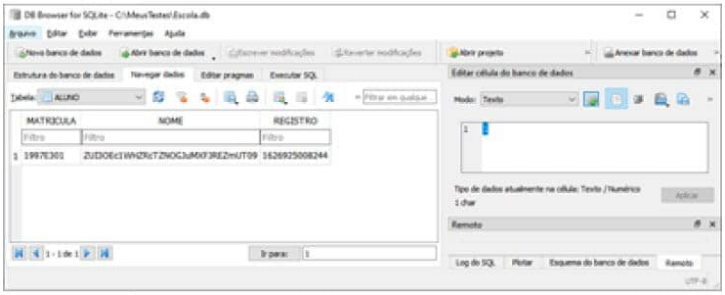
Para verificar a criptografia de valores no banco de dados, iniciaremos com o download de **Escola.db**, no subdiretório databases, pela opção Save As.



Captura de tela do Android Studio - Download de arquivo do dispositivo.

Após salvar o arquivo, precisaremos de um programa capaz de abri-lo, como o SQLite Browser, disponível em **https://sqlitebrowser.org**. Lembre-se de não utilizar a versão Cipher, pois ela lida com encriptação do banco completo, segundo algoritmos próprios.

Com o programa baixado, execute-o e utilize a opção Abrir Banco de Dados, a partir de Arquivo, escolhendo a base **Escola.db**. Abrindo a tabela ALUNO, na aba Navegar Dados, teremos acesso aos valores do campo NOME, criptografados e codificados para Base64.



Captura de tela do SQLite Browser - Visualização do arquivo de banco de dados.

Criptografia no Expo

Quando trabalhamos com o Expo, temos uma biblioteca denominada expo-crypto, que permite efetuar diversos processos de criptografia com grande simplicidade. Com base na biblioteca, vamos definir uma estrutura básica de login, com a gravação das senhas no banco de dados SQLite, utilizando criptografia destrutiva SHA512.

A configuração de nosso projeto é apresentada a seguir. Observe a inclusão dos recursos para suporte ao Type Script, por meio de bibliotecas como typescript e types/react, com base em versões específicas, algo que é sugerido pelo console de execução do Expo.

Terminal

```
expo init CadastroUsuario
 cd CadastroUsuario
 expo install expo-crypto
 expo install expo-sqlite
 npm install @react-navigation/native @react-navigation/stack
 expo install react-native-gesture-handler react-native-reanimated react-native-screens react-native-saf
 npm install --save-dev typescript@~4.0.0 @types/react@~16.9.35 @types/react-native@~0.63.2
```

Precisamos definir a mesma estrutura de diretórios do exemplo de MVC, ou seja, vamos criar o diretório cadastro, e dentro dele os subdiretórios controller, model e view. Como a conexão com SQLite pelo Expo apresenta diferenças sutis, não poderemos aproveitar a conexão, sendo necessário utilizar o código seguinte no arquivo **DatabaseInstance.js**.

Javascript

```
import * as SQLite from 'expo-sqlite';

 var db = SQLite.openDatabase('Escola.db');

 export default db;
```

Os arquivos **GenericDAO.ts** e **GenericController.ts** serão reutilizados sem modificações, com a mesma estrutura de diretórios.

Em seguida, vamos criar o arquivo **Usuario.ts**, no subdiretório model, para a definição da entidade Usuario, composta dos atributos login, que será o identificador, nome e senha.

Javascript

```
export class Usuario{
      login: string;
      nome: string;
      senha: string;

      constructor(login: string, nome: string, senha: string){
          this.login = login;
          this.nome = nome;
          this.senha = senha;
      }
      }
```

Passamos para a definição do DAO, no arquivo **UsuarioDAO.ts**, subdiretório model, fornecendo os complementos necessários, e

acrescentando um método para alteração da senha.

Javascript

```
import db from './DatabaseInstance';
 import { Usuario } from './Usuario';
 import { GenericDAO } from './GenericDAO';

 export default class UsuarioDAO extends
                      GenericDAO<Usuario,string>{
   protected getCreateSQL(): string {
     return ' CREATE TABLE USUARIO(LOGIN VARCHAR(10) '+
            ' PRIMARY KEY, '+
            ' NOME VARCHAR(20), SENHA VARCHAR(128))  ';
   }
   protected getTableName(): string {return 'USUARIO';}
   protected getInsertSQL(): string {
     return 'INSERT INTO USUARIO VALUES ( ?, ?, ? )';
```

Serão fornecidos o nome da tabela, que no caso é USARIO, comandos SQL e parâmetros necessários, além de um método para obter a entidade a partir do registro.

Adicionamos um método para alteração da senha, com o nome alterarSenha, recebendo a chave (login) e a nova senha. Nos sistemas para controle de acesso é uma boa prática separar o tratamento da senha, nos comandos UPDATE, de qualquer outro campo da entidade.

A criptografia será gerenciada no controlador, já que a senha é armazenada e recuperada na forma criptografada, eliminando a necessidade de tratá-la na obtenção da entidade. O código do controlador, no arquivo **UsuarioController.ts**, é apresentado a seguir.

Javascript

```
import {Usuario} from '../model/Usuario';
 import UsuarioDAO from '../model/UsuarioDAO';
 import GenericController from './GenericController';
 import * as Crypto from 'expo-crypto';
 import {CryptoEncoding} from 'expo-crypto';

 export default class UsuarioController extends
                      GenericController<Usuario,string>{
   private async getCripto(senha: string): Promise<string>{
     const senhaCripto = await Crypto.digestStringAsync(
               Crypto.CryptoDigestAlgorithm.SHA512,senha,
               {encoding: CryptoEncoding.BASE64});
     return senhaCripto;
   }
```

Nosso controlador apresenta um método interno para obtenção do valor criptografado, com base na biblioteca expo-crypto, adotando algoritmo SHA512, o qual gera um hash com alto nível de segurança, e codificação Base64. Qualquer operação que envolva o uso da senha deverá invocar o método getCripto sobre o valor fornecido.

> **No método setDAO:**
>
> **Temos o fornecimento de um objeto do tipo UsuarioDAO, definindo as funcionalidades básicas do controlador MVC, mas o método incluir deve ser alterado, já que temos o uso da senha. A alteração é simples, com a criptografia do campo senha da entidade, antes de invocar o método de inclusão herdado do controlador genérico.**

Temos dois novos métodos em nosso controlador, um para alteração da senha, e outro para recuperação do usuário a partir do login e da senha. No método alterarSenha, foi adotado o mesmo modelo de programação das alterações genéricas, mas a senha deve ser criptografada, antes da invocação do método subsequente no DAO, e como não era um método previsto no modelo genérico, ocorre a necessidade da conversão de tipo.

O método obterLoginSenha utiliza o modelo de callback do SQLite, com o retorno da entidade para o método de tratamento. Efetuamos a consulta pela chave e comparamos a senha armazenada com o valor criptografado da senha fornecida, retornando o usuário obtido quando os valores são iguais, ou nulo para qualquer situação diferente.

Atenção

Utilizamos o operador de decisão para substituir a estrutura tradicional de seletores, no método obterLoginSenha, diminuindo muito a quantidade de código.

Agora, já podemos definir nossa camada View, começando pela criação de **CommonStyles.js**, no subdiretório view, com o código seguinte.

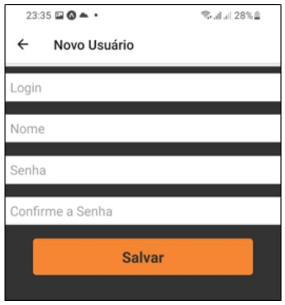
Concluído o conjunto de estilos global, vamos definir a tela para inclusão de usuário, no arquivo **UsuarioForm.js**, subdiretório view.

Javascript

```
1  import React, {useState} from 'react';
2   import {Text, View, TextInput, TouchableOpacity, Alert}
3          from 'react-native';
4   import UsuarioController
5          from '../controller/UsuarioController';
6   import {Usuario} from '../model/Usuario';
7   import {styles} from './CommonStyles';
8
9   export default function UsuarioForm({navigation}) {
10    const gestor = new UsuarioController();
11    const[login,setLogin] = useState('');
12    const[nome,setNome] = useState('');
13    const[senha,setSenha] = useState('');
14    const[senhaConf,setSenhaConf] = useState('');
```

Definimos um controlador com o nome gestor, além das variáveis de estado login, nome, senha e senhaConf. Também temos o método salvar, onde inicialmente testamos a confirmação da senha, emitindo um alerta para valores diferentes, ou efetuando a inclusão do usuário, com os dados digitados, levando ao retorno para a tela de login no caso de sucesso.

Quanto ao componente de visualização, temos quatro campos de entrada de texto, que estão associados às variáveis de estado, por meio de value e onChangeText, e um botão para acionar o método salvar.



Captura de tela do React Native - Tela para cadastro de usuários.

Para criar a tela de acesso ao sistema, vamos definir o arquivo **Login.js**, no subdiretório view, contendo a listagem seguinte.

Javascript

```
import React, {useState} from 'react';
 import {TextInput, TouchableOpacity, Alert, Text, View}
        from 'react-native';
 import {styles} from './CommonStyles';
 import UsuarioController
        from '../controller/UsuarioController';

 export default function Login({navigation}) {
   const gestor = new UsuarioController();
   const[login,setLogin] = useState('');
   const[senha,setSenha] = useState('');

   const logar = () => {
     gestor.obterLoginSenha(login,senha,(entidade)=>{
```

Nossa tela é composta de entradas para login e senha, um botão para autenticação, e outro para cadastro de usuário. O botão de cadastro irá apenas navegar para UsuarioForm, apresentando o componente descrito anteriormente.

Já o outro botão, com o texto Entrar, invoca o método logar, em que os dados das variáveis de estado são utilizados na chamada ao método obterLoginSenha, com o tratamento do retorno sendo efetuado na callback. Para uma entidade nula, apenas mostramos um alerta indicando erro no usuário ou senha, enquanto para um login bem-sucedido, limpamos os campos da tela de acesso e navegamos para a tela principal, definida em **MenuTela.js**, arquivo que deverá ser criado no subdiretório view, contendo a listagem apresentada a seguir.

Javascript

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

```
5
6   export default function MenuTela({navigation}){
7     return (
8       <View style={styles.container}>
9         <TouchableOpacity style={styles.menuButton}
10          onPress={()=>Alert.alert('Cadastro de Alunos')}>
11          <Text style={styles.buttonTextBig}>Alunos</Text>
12        </TouchableOpacity>
13        <TouchableOpacity style={styles.menuButton}
14          onPress={()=>Alert.alert('Cadastro de Professores')}>
```

Nosso menu não faz nada de especial, pois apenas apresenta alertas indicando as telas que seriam abertas em um sistema completo. A única funcionalidade relevante está no botão com o texto Logout, em que ocorre a navegação para a tela de acesso.

Agora só falta alterar o conteúdo de **App.js**, no diretório raiz, de acordo com a listagem seguinte. Observe que teremos a definição da navegação, com as três telas criadas, mas em duas delas anulamos a ação de retorno, com base em headerLeft.

Javascript

```
1  import {NavigationContainer} from '@react-navigation/native';
2   import {createStackNavigator} from '@react-navigation/stack';
3   import React from 'react';
4   import Login from './cadastro/view/Login';
5   import MenuTela from './cadastro/view/MenuTela';
6   import UsuarioForm from './cadastro/view/UsuarioForm';
7
8   const Stack = createStackNavigator();
9
10  export default function App() {
11    return (
12      <NavigationContainer>
13        <Stack.Navigator initialRouteName='Login'>
14          <Stack.Screen name='Login' component={Login}
```

Já podemos executar o sistema, criar um usuário e efetuar o login a partir de suas credenciais.

Captura de tela do React Native - Tela de login e menu principal do aplicativo.

Criptografia no armazenamento de dados

No vídeo a seguir, abordamos o uso de recursos de criptografia na persistência e no controle de acesso a dados.

Vem que eu te explico!

Os vídeos a seguir abordam os assuntos mais relevantes do conteúdo que você acabou de estudar.

Criptografia no React Native

4:12min

Criptografia no Expo

3:41min

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

O algoritmo 3DES caracteriza-se por ser um processo reversível, em que uma mesma chave, normalmente denominada Secret Key, é utilizada para criptografar e para recuperar os dados originais. Esse tipo de processo é conhecido como

A criptografia assimétrica.

B codificação simples.

C assinatura digital.

D criptografia simétrica.

E criptografia destrutiva.

Responder

Questão 2

No ambiente do Expo, temos uma biblioteca muito útil, denominada expo-crypto, que permite efetuar processos criptográficos no modelo destrutivo. Com base na biblioteca, qual seria o comando necessário para a geração de um hash de forma assíncrona?

A digestStringAsync

B CryptoDigestAlgorithm

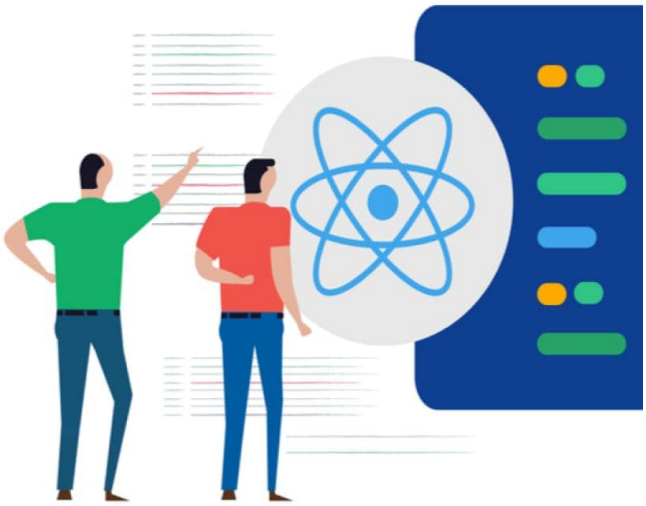
C encoding

D CryptoDigestOptions

E CryptoEncoding

Responder

★★★★★



4 - Publicação de aplicativos com React Native

Ao final deste módulo, você será capaz de empregar testes e análise de performance na publicação de aplicativos com React Native.

CI/CD

Hoje em dia é comum a adoção dos termos integração contínua, ou CI, e entrega contínua, ou CD, utilizados em metodologias ágeis, como Scrum e XP (Extreme Programming). O princípio básico é a entrega de funcionalidades completas, a cada novo ciclo, priorizadas de acordo com as necessidades do cliente.

Algo evidente na adoção dessas metodologias é a necessidade de indicadores palpáveis para a garantia da qualidade de cada grupo de funcionalidades entregue, em que as metodologias de teste de software apresentam grande relevância no desenvolvimento do sistema. Antes que um novo módulo seja aceito, todos os testes precisam ser completados com sucesso, e alguns testes podem envolver mais de um módulo, já que uma funcionalidade pode impactar outras.

Embora alguns testes tenham como objetivo aferir apenas a eficácia, ou seja, verificar se uma funcionalidade atingiu seu objetivo, outros levam em conta a eficiência. De nada adiantaria criar um sistema que cumprisse com todas as regras de negócio, mas exigindo um tempo tão longo que inviabilizasse sua utilização pelos usuários.

> **O termo usabilidade se tornou comum na área de desenvolvimento, e ele expressa o nível de resposta às ações do usuário em termos de agilidade e ergonomia. Toda ação oferecida pelo sistema deve ser de simples utilização, e a resposta deve ocorrer no menor tempo possível.**

Para garantir a fluidez, precisamos efetuar testes de execução, com ferramentas apropriadas, visando determinar a evolução do consumo de memória e o tempo em cada ação. A análise dos aspectos gerais do software em tempo de execução permitirá a detecção de gargalos e a definição de estratégias apropriadas para melhoria da performance, algo que ficou conhecido como performance tuning.

Atenção

Entre os problemas comuns de performance, temos o memory leak, ou vazamento de memória, uso de eventos incorretos, causando execuções recursivas, e gasto excessivo de tempo nas operações de entrada e saída. As soluções encontradas podem envolver a adoção de bibliotecas alternativas, substituição do motor de execução, ou até mesmo a modificação de algoritmos.

Ao final de um ciclo de desenvolvimento, cada módulo deve estar testado e otimizado, além de compatível com os demais módulos já desenvolvidos, com a garantia da qualidade necessária para que seja integrado ao restante do sistema. É o momento em que ocorre o empacotamento, ou shipping, com a criação e disponibilização de uma versão do sistema.

Em termos de dispositivos móveis, a metodologia de desenvolvimento pode ser a mesma de sistemas maiores, embora os aplicativos, muitas vezes, exijam poucos ciclos. Além disso, o sistema de distribuição de aplicativos costuma utilizar lojas virtuais específicas, como a da Apple e a do Google, e o empacotamento deve satisfazer aos requisitos de publicação dessas lojas, incluindo a assinatura digital do aplicativo.

Teste de software

Os testes automatizados visam garantir as funcionalidades essenciais do software ainda durante o processo de desenvolvimento, verificando processos pontuais e o efeito de mudanças sobre o sistema como um todo, além de simular situações de erro com grande facilidade. Em termos práticos, qualquer modificação será aceita apenas quando todos os testes forem satisfatórios.

Existem diversos níveis de teste, que variam em termos do escopo funcional, como pode ser observado no quadro seguinte.

Teste	Descrição
Unitário	Validação de funcionalidades pontuais, baseadas em métodos ou classes específicas.
Integração	Verifica a funcionalidade frente aos demais módulos do sistema, da mesma forma que o teste unitário, mas sem isolamento.
Sistêmico	Traz a perspectiva da execução propriamente dita, com a integração de APIs externas e validação funcional da interface de usuário.

Quadro: Classificação de testes automatizados.
Elaborado por: Denis Cople.

A biblioteca padrão para testes unitários no React Native é o framework Jest, algo que pode ser observado no atributo test, divisão scripts, do arquivo package.json. Podemos definir um script de testes simplesmente utilizando o sufixo test.js no nome do arquivo.

Para exemplificar, vamos criar o arquivo MathOper.test.js, com seu conteúdo definido a partir da listagem seguinte.

Javascript

```
function somar(a,b){
    return a+b;
}

test('Soma Sucesso', () => {
    const result = somar(10,5);
    expect(result).toEqual(15);
})

test('Soma Falha', () => {
    const result = somar(10,5);
    expect(result).toEqual(5);
})
```

Aqui, temos dois testes sobre uma função simples, em que o primeiro terá resultado positivo e o segundo irá gerar uma falha. A

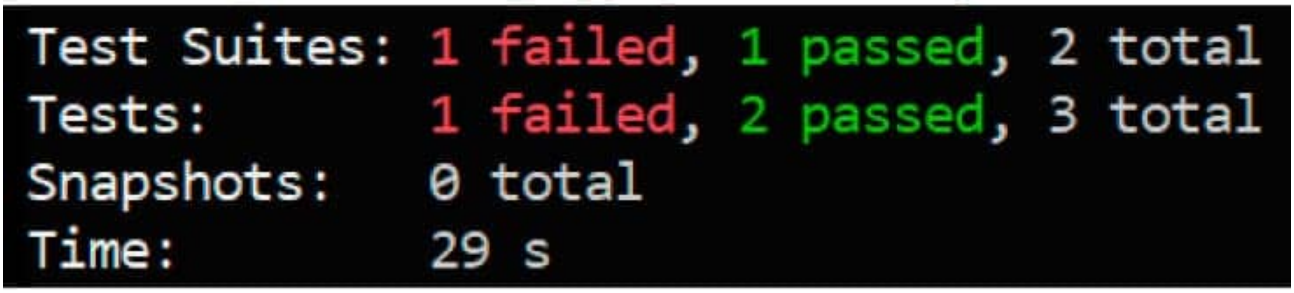
diretiva test aceita um título e um bloco de execução, e no bloco temos o resultado verificado com expect.

Também podemos definir testes sem um sufixo especial no diretório "**__tests__**", como no modelo gerado automaticamente para App.js, com o nome **App-test.js**. Para executar todos os testes do aplicativo, será utilizado o comando seguinte.

Terminal

```
npm test
```

O resultado é exibido no console, com uma síntese da quantidade de falhas ou sucessos, como pode ser observada a seguir. Cada arquivo contendo testes é considerado um Test Suite, e se qualquer teste unitário falha, todo o conjunto é considerado falho.



Execução de testes com base em Jest no console.

Nosso primeiro teste poderia ser escrito de outra forma, adotando **it** e **toBe**.

Javascript

```
it('Soma Sucesso 2',()=>{
      expect(somar(10,5)).toBe(15);
 });
```

No quadro seguinte, temos alguns dos operadores disponíveis para efetuar testes pelo Jest.

Operador	Descrição
toBe	Verifica se o valor retornado é igual ao de teste.
toBeFalsy	Testa valores falsos, nulos ou indefinidos.
toBeGreaterThan	Verifica se o valor de retorno é maior que o fornecido.
toBeInstaceOf	Verifica se o objeto é instância de uma classe ou descendente.
toBeLessThan	Verifica se o valor de retorno é menor que o fornecido.
toHaveProperty	Testa a existência ou valor de uma propriedade.
toMatch	Testa a presença do texto ou expressão regular.
toThrow	Testa a ocorrência de uma exceção durante a execução.

Quadro: Alguns operadores de teste disponíveis no Jest.
Elaborado por: Denis Cople.

No quadro seguinte, temos alguns dos operadores disponíveis para efetuar testes pelo Jest.

Embora o uso de expect permita efetuar testes unitários e de integração, precisaremos de uma estrutura capaz de simular as chamadas para componentes externos para criar os testes sistêmicos, como servidores HTTP. Objetos mock, ou simulados, serão utilizados para fornecer as respostas previstas por determinado componente, atuando como servidores falsos, na maioria das vezes, o que elimina a necessidade de construir um servidor físico para os testes.

No código seguinte, temos um método estático, para consulta ao endereço REST, que retorna o conjunto de usuários no formato JSON.

Javascript

```
import axios from 'react-native-axios';

 export class Usuario{
   login: string;
   nome: string;
   constructor(login: string, nome: string){
     this.login = login;
     this.nome = nome;
   }
   static all(): Array<Usuario>{
     return axios.get('https://servidor/Usuarios').then(
                 resp => resp.data.map(x => JSON.parse(x)));
   }
 }
```

Temos uma chamada HTTPS, via método GET, e a resposta, recebida como um vetor JSON, é mapeada, gerando uma coleção de objetos. Sem a criação do servidor, o método all não poderia ser verificado durante a execução.

Uma solução simples é o teste por um mock, como pode ser observado a seguir.

Javascript

```
import axios from 'react-native-axios';
 import {Usuario} from '../model/Usuario';

 jest.mock('react-native-axios');

 test('Retornando os usuários...', () => {
   const usuario = new Usuario('bob', 'Bob Sauro');
   const usuariosJSON = [JSON.stringify(usuario)];
   const resp = {data: usuariosJSON};
   axios.get.mockImplementation((url) => {
     if(url==='https://servidor/Usuarios')
       return Promise.resolve(resp);
     else
       return Promise.resolve(null);
```

Inicialmente aplicamos o mock sobre a biblioteca react-native-axios, para que aceite o desvio implementado posteriormente. No teste geramos um vetor de usuários no formato JSON, e definimos um objeto, com o atributo data recebendo o vetor, para fornecer a resposta.

Ao invocarmos mockImplementation para o método get de axios, implementamos um retorno assíncrono, com base em Promise, fornecendo nossa resposta predefinida, para substituição do comportamento padrão do axios. Na chamada para o método all, temos

a invocação para get desviada para a resposta de teste, a qual é mapeada para um vetor de usuários, permitindo a comparação com o objeto original do teste.

No código, foi possível observar uma das formas adotadas para os testes assíncronos, com base no operador then para execução sequencial, mas existem outras formas para implementação assíncrona, como o uso de resolves, o que exigiria a mudança do retorno na primeira versão.

Javascript

```
test('Retornando os usuários...', () => {
      const usuario = new Usuario('bob', 'Bob Sauro');
      const usuariosJSON = [JSON.stringify(usuario)];
      const resp = {data: usuariosJSON};
      axios.get.mockImplementation((url) => {
          if(url==='https://servidor/Usuarios')
          return Promise.resolve(resp);
          else
          return Promise.resolve(null);
      });
      return expect(Usuario.all()).resolves.toEqual([usuario]);
  });
```

Por fim, temos os testes sobre componentes nativos, ou seja, a verificação das interfaces de usuário. Para compreender as técnicas, vamos criar o arquivo UsuarioForm.js, de acordo com a listagem seguinte, e adicionar o arquivo CommonStyles.js, criado anteriormente.

Javascript

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

 export default function UsuarioForm( ) {
   const[login,setLogin] = useState('');
   const[nome,setNome] = useState('');
   const[senha,setSenha] = useState('');
   const[senhaConf,setSenhaConf] = useState('');
   const salvar = () => {
     if(senha!=senhaConf)
       Alert.alert('Falha na confirmação da senha');
     else
```

Por padrão, nas atuais versões do React Native, é criado um diretório __tests__ com um arquivo **App-test.js**, contendo o teste para o desenho da tela inicial. Iremos acrescentar, no mesmo diretório, o arquivo **UsuarioForm-test.js**, com o código apresentado a seguir.la quis lobortis et, scelerisque ac dolor.

Javascript

```
import 'react-native';
 import React from 'react';
 import UsuarioForm from '../cadastro/view/UsuarioForm';

 import renderer from 'react-test-renderer';

 it('renders correctly', () => {
   renderer.create(<UsuarioForm />);
 });
```

O teste do formulário terá sucesso, já que foi construído corretamente, mas a alteração de um componente TextInput para TextInputX, por exemplo, permitiria gerar uma falha. Outra forma de teste de interface é pelo snapshot, em que geramos um instantâneo no formato JSON, na primeira execução, e qualquer alteração será detectada nas execuções subsequentes.

Javascript

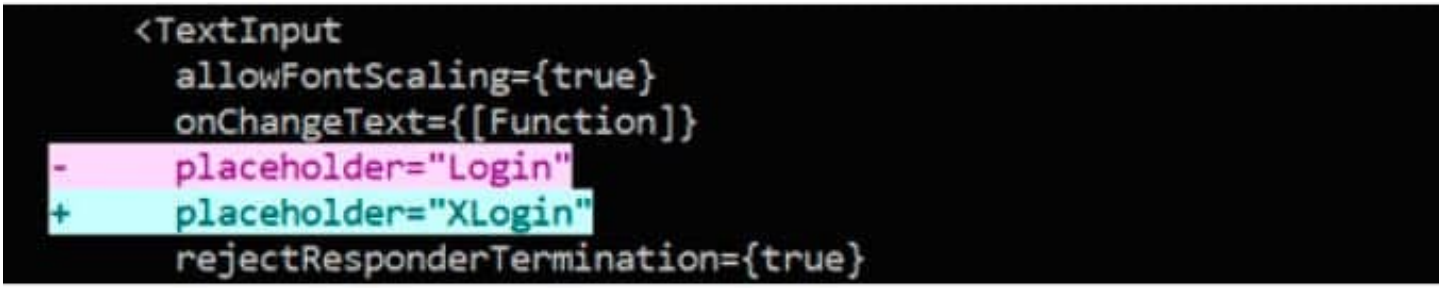
```
import React from 'react';
 import UsuarioForm from '../cadastro/view/UsuarioForm';

 import renderer from 'react-test-renderer';

 it('renders correctly', () => {
   const tree = renderer.create(<UsuarioForm />).toJSON();
   expect(tree).toMatchInlineSnapshot();
 });
```

Executando o teste, teremos a modificação do arquivo, com a inclusão de um snapshot como parâmetro de toMatchInlineSnapshot. Podemos observar, no fragmento apresentado a seguir, parte da alteração decorrente da primeira execução.

Nas execuções subsequentes, teremos a comparação com o snapshot, o que pode ser verificado com a modificação de qualquer atributo de UsuarioForm e nova execução. Com a modificação de um dos atributos placeholder, podemos experimentar a técnica facilmente.



Detecção de mudança na tela via comparação com snapshot.

Embora o modo nativo, nas versões atuais, inclua automaticamente o Jest, para criar testes no Expo é necessário incluir os pacotes corretos.

Terminal

```
npm i jest-expo --save-dev
npm i react-test-renderer --save-dev
```

Também devemos alterar o arquivo **package.json**, acrescentando uma cláusula no bloco scripts, e criando um bloco jest, de acordo com o fragmento apresentado a seguir.

Javascript

```
...
'scripts': {
  ...
  'test': 'jest'
},
'jest': {
  'preset': 'jest-expo'
},
...
```

Configurado o ambiente no Expo, podemos utilizar as técnicas apresentadas anteriormente, com a execução iniciada pelo mesmo comando npm.

Performance Tuning

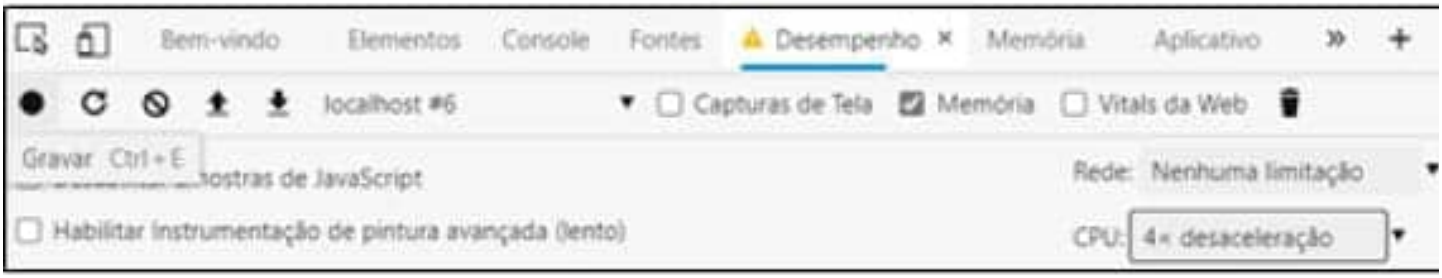
Um princípio básico na construção de aplicativos é o de que a interface gráfica deve ser fluida, de acordo com as possibilidades de exibição do hardware, que hoje trabalha pelo menos a sessenta quadros por segundo, ou 60 FPS. Devemos garantir que o desenho da

tela satisfaça ao mínimo de velocidade de alternância requerida, impedindo a sensação de travamentos durante a execução, o que justifica a adoção de uma thread específica para controle da interface de usuário, normalmente chamada de UI.

> **Nossa preocupação primária é a diminuição do redesenho na UI, com a utilização de estratégias mais organizadas e componentes otimizados. Por exemplo, uma recomendação é a adoção de FlatList para dados multivalorados, principalmente em grandes massas.**

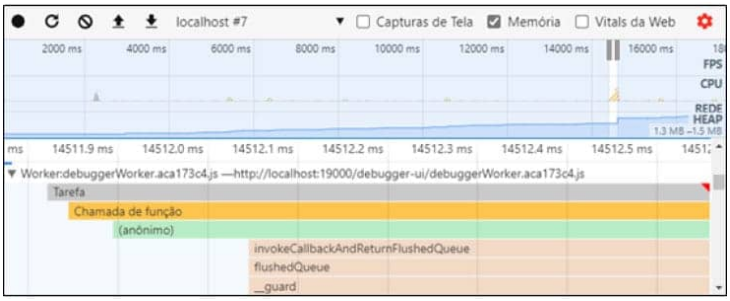
Com relação aos processos executados, devemos observar o uso de memória, complexidade de código e tempo de execução. Podemos analisar, por meio de ferramentas adequadas, o custo em termos de memória e ciclos de máquina, para cada função invocada durante a execução.

Com o uso do DevTools, na opção Desempenho, podemos iniciar a gravação de sequências de ações, quando em modo de depuração. Os resultados são exibidos em uma linha de tempo, permitindo ampliação, redução e movimentação por meio das teclas W, S, A e D.



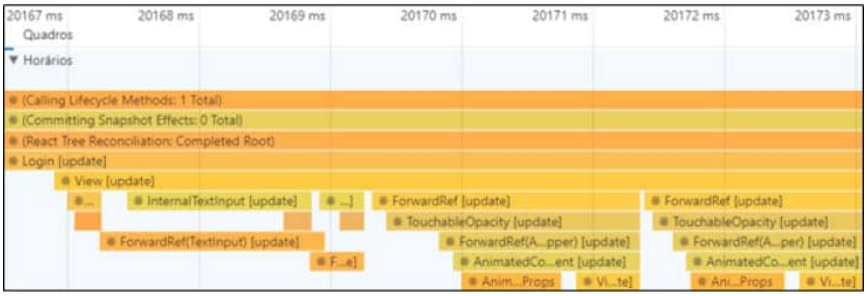
Captura de tela No DevTools - Gravação das ações executadas no aplicativo.

Após parar a gravação, podemos analisar os resultados, movendo para a área de interesse e aplicando o zoom necessário. Como os tempos são ínfimos, parecerão pontos esparsos a cada grupo de análise, algo minimizado com uma desaceleração de fator 4 antes de gravar.



Captura de tela No DevTools - Seleção de área de análise e visualização do tempo em ms.

Alguns ciclos podem apresentar diversas divisões internas, decorrentes das chamadas para subfunções. Podemos observar, a seguir, um trecho de execução de ações sobre a tela de login, criada em nosso controle de acesso.



Captura de tela No DevTools - Acompanhamento da execução da tela de login.

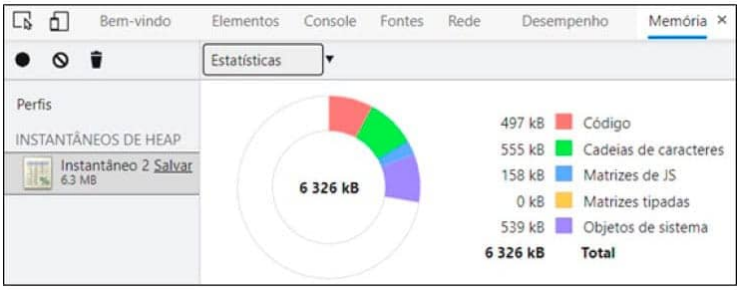
Com a análise, obtemos informações importantes, como o tempo necessário para desenhar os botões, ou a influência do padrão Observer, adotado por useState, no tempo de atualização. Em algumas circunstâncias, é possível verificar problemas de desenho em que componentes são atualizados continuamente.

Entre os grupos de execução, destacamos Timings, traduzido como Horários, acompanhando as tarefas de desenho e os eventos relacionados aos componentes nativos, e Debugger Worker, para analisar as chamadas de funções em Java Script ou Type Script.

O consumo de recursos pelo aplicativo fica no painel inferior. Devemos ficar muito atentos ao consumo de memória, no Heap JS e na GPU, observando sua evolução ao longo do tempo.

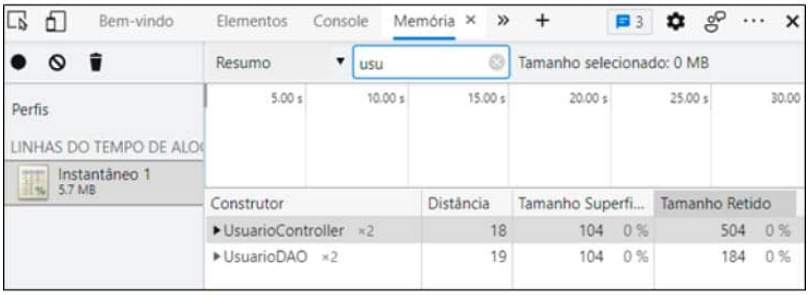
Captura de tela No DevTools - Consumo de recursos pelo aplicativo ao longo do tempo.

Além do acompanhamento da execução na opção Desempenho, temos a possibilidade de tirar instantâneos de alocação na opção Memória. Note que será necessário escolher a máquina virtual JS como Debugger Worker, ao tirar o instantâneo.



Captura de tela No DevTools - Estatística de uso de memória em um instantâneo pelo aplicativo.

Podemos analisar o consumo de memória para alguma classe específica, com a aplicação de um filtro, a partir do Resumo. As duas métricas que devem ser consideradas em nossa análise são o tamanho superficial, representando o espaço alocado em bytes para o objeto, e o tamanho retido, que inclui o espaço utilizado pelos gráficos associados.



Captura de tela No DevTools - Consumo de memória por componentes específicos.

Além das ferramentas de análise, temos técnicas para aumento de performance, como o uso de FlatList, citado anteriormente. Outras boas práticas incluem a retirada de mensagens de log no console, configuração das imagens para menor utilização de recursos, diminuição do redesenho de telas e substituição do motor Java Script pelo Hermes.

A utilização do plugin para remoção de console, parte da plataforma babel, é uma forma simples para eliminar mensagens de log.

Terminal

```
npm add babel-plugin-transform-remove-console
```

Será necessário criar o arquivo "**.babelrc**", na raiz do projeto, com a inclusão do conteúdo apresentado a seguir.

Javascript

```
{
    'env': {
        'production': {
        'plugins': ['transform-remove-console']
        }
    }
 }
```

Redimensionamento e corte de imagens trazem grande custo na execução do aplicativo, o que nos leva à necessidade de trabalhar com imagens já tratadas, por meio de editores, como o Photoshop, para que sigam as proporções adotadas no dispositivo. Também devemos trabalhar com o cache de imagens para solicitações remotas, como exemplificado a seguir.

Javascript

```
<Image
 source={{uri: 'https://facebook.github.io/react/logo-og.png',
          cache: 'only-i-cached'}},
 style={{width: 400, height: 400}} />
```

Os melhores formatos de imagem são PNG (Portable Network Graphics) e vetorial. As imagens vetoriais são ótimas para resoluções múltiplas, com o mínimo de informações e pouca distorção, sendo o padrão adotado nos ícones do Material Design, referência para os designers.

> **Talvez nossa maior preocupação, para a garantia da performance do aplicativo, seja a otimização do redesenho. Por exemplo, podemos utilizar o evento componentWillUpdate na atualização de informações da tela, mas não devemos gerar alterações nos componentes da UI, pois ocorre o redesenho de forma recursiva.**

É possível memorizar os componentes que não apresentem alterações nas informações, como listas de unidades federativas, ou telas com menus de acesso. Basta utilizar **React.memo** para encapsular a exportação do componente, diminuindo consideravelmente o redesenho.

Javascript

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

 function MenuTelaX({navigation}){
   return (
     <View style={styles.container}>
       <TouchableOpacity style={styles.button}
             onPress={()=>navigation.navigate('AlunoTela')}>
         <Text style={styles.buttonTextBig}>Alunos</Text>
       </TouchableOpacity>
     </View>
   );
 }
```

Adotando Hermes, no lugar do motor padrão, temos diversas vantagens, destacando-se o menor tamanho da biblioteca e a diminuição do tempo de carga. A integração no projeto é muito simples, mas o motor é compatível apenas com as versões mais recentes do React Native.

A configuração para Android exige a modificação do arquivo **build.gradle**, no diretório app do setor android, em nossos projetos.

project.ext.react = [
enableHermes: true,
]

Também devemos adicionar as regras listadas a seguir, no arquivo proguard-rules.pro, o qual fica no mesmo diretório do passo anterior.

*-keep class com.facebook.hermes.unicode.** { *; }*
*-keep class com.facebook.jni.** { *; }*

Por fim, a modificação do motor de execução exigirá uma compilação limpa, utilizando os comandos da listagem seguinte.

Terminal

```
cd android
 gradlew clean
```

As aplicações nativas podem utilizar Hermes no iOS, exigindo a modificação de Podfile, do setor ios, no trecho que é apresentado a seguir.

```
use_react_native!(
 :path => config[:reactNativePath],
 :hermes_enabled => true
)
```

Da mesma forma que para o Android, precisamos de uma compilação limpa após a modificação, o que é feito com os comandos da listagem seguinte.

Terminal

```
cd ios
 pod install
```

Para o Expo, a adoção de Hermes para Android exige apenas a inclusão do atributo jsEngine, na configuração android do arquivo **app.json**, posicionado na raiz, conforme o fragmento seguinte, enquanto para iOS, ainda não é possível.

```
"android": {
"jsEngine": "hermes",
 "adaptiveIcon": {
 "foregroundImage": "./assets/adaptive-icon.png",
 "backgroundColor": "#FFFFFF"
 }
 }
```

Por fim, nem sempre a codificação mais elegante é a menos custosa, pois a complexidade tem um efeito drástico sobre o uso de recursos. Um exemplo clássico é a série de Fibonacci, que tem um gasto de memória muito maior em uma implementação recursiva.

Shipping

Após efetuar todos os testes necessários, analisar os contextos de execução, e melhorar o desempenho do aplicativo, podemos publicá-lo. As ações de publicação, ou shipping, envolvem o empacotamento e preenchimento de requisitos definidos na loja da plataforma.

Antes de mais nada, precisamos ter uma conta de desenvolvedor na loja da Apple, para o sistema iOS, ou na loja do Google, para Android. A loja do Google requer o pagamento e uma taxa única, enquanto a loja da Apple trabalha com uma taxa anual.

Uma exigência comum é a assinatura do aplicativo, e para o Android será necessário criar um certificado digital por meio do keytool, oferecido na plataforma JDK. Precisamos de um par de chaves RSA, com 2048 bits e validade mínima de 10000 dias, armazenado em uma keystore.

```
keytool -genkeypair -v -keystore upkeyfile.keystore -alias upkey -keyalg RSA -keysize 2048 -validity 10000
```

Durante a criação do certificado digital serão solicitados muitos dados, e o primeiro deles será a **senha**, para a qual adotaremos a sequência "**abc123456**".

Terminal

```
Enter keystore password:
 Re-enter new password:
 What is your first and last name?
   [Unknown]:  Denis Cople
 What is the name of your organizational unit?
   [Unknown]:  Develop
 What is the name of your organization?
   [Unknown]:  ABCD
 What is the name of your City or Locality?
   [Unknown]:  Rio de Janeiro
 What is the name of your State or Province?
   [Unknown]:  RJ
 What is the two-letter country code for this unit?
   [Unknown]:  BR
```

O arquivo **upkeyfile.keystore** deve ser copiado para a **pasta app**, na divisão android, de nosso projeto. Em seguida, vamos acrescentar algumas linhas ao arquivo build.gradle, na mesma pasta, para que seja configurada a compilação com assinatura do aplicativo.

```
...
android{
 ...
 defaultConfig {...}
 signingConfigs {
...
 release {
 storeFile file('upkeyfile.keystore')
 storePassword 'abc123456'
 keyAlias 'upkey'
 keyPassword 'abc123456'
 }
}
 buildTypes {
 release {
...
 signingConfig signingConfigs.release
 }
 }
}
```

Um arquivo do tipo AAB (Android Application Bundle) precisa ser gerado, por meio dos comandos listados a seguir, executados no console, a partir da raiz do projeto. Será criado o arquivo **app-release.aab** em app/build/outputs/bundle/release.

Terminal

```
cd android
```

```
gradlew clean
gradlew bundleRelease
```

No console do Google Play deve ser criado um aplicativo, definida uma versão de teste interno, e adicionado o arquivo AAB. Devemos adotar o modelo de assinatura automática do Google para que a plataforma controle a certificação digital a partir do envio do arquivo.

Recomendação

Você também pode abrir o projeto da divisão android no Android Studio, e seguir o processo tradicional de compilação e publicação.

No ambiente da Apple, o controle é feito pelo XCode, com a associação da ferramenta a uma conta de desenvolvimento da plataforma, e utilização de assinatura automática, mas é possível gerar um arquivo IPA (IPhone Application), com alguns passos realizados em um Mac, começando pela execução do aplicativo, por meio do comando apresentado a seguir.

Terminal

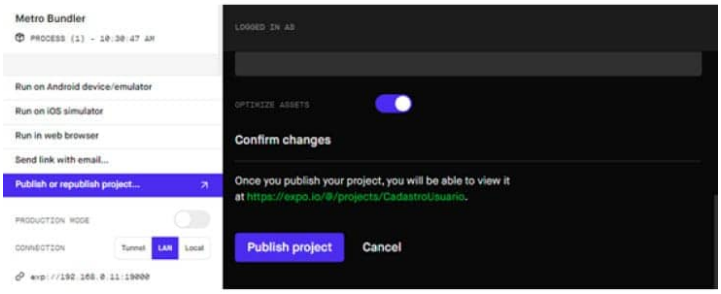
```
react-native run-ios --configuration=release
```

Em seguida, crie uma pasta Payload e copie o arquivo app, que foi gerado no processo anterior em Build/Products/Release, para a pasta. Por fim, comprima a pasta Payload, e renomeie o arquivo compactado para um nome de sua escolha, com extensão ipa.

Atenção

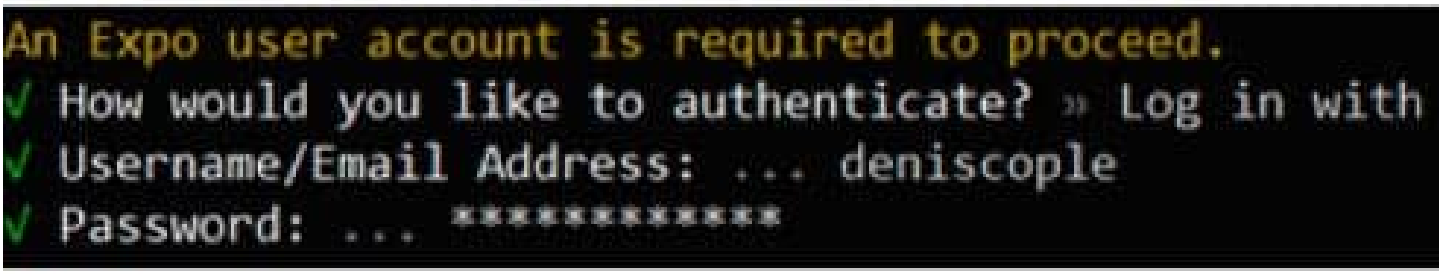
No ambiente Mac é mais indicado trabalhar apenas com o XCode, com a abertura de xcodeproject, ou xcworkspace para CocoaPods, ambos os arquivos localizados na divisão ios.

Já no Expo, o processo de publicação pode ser feito a partir da linha de comando, ou a partir da interface Web, onde o processo é iniciado com o clique no botão de publicação.



Captura de tela No Expo - Publicação do aplicativo através da interface Web do Expo.

No passo seguinte, é requerido o login no ambiente remoto, devendo ser criada uma conta no primeiro acesso à plataforma. Após o login, todo o processo ocorre de forma automática.



Captura da tela No Expo - Escolha da conta para publicação através do console.

Ao final será exibida uma mensagem na interface Web, com a indicação do endereço de acesso, onde será exibido o QR Code, que deve ser fotografado a partir do celular.

Para publicação nas lojas será necessário utilizar a opção build, do Expo, com o pacote assinado automaticamente, iniciando pelo comando apresentado a seguir.

Terminal

```
1  expo build:android
```

Apesar de ser um processo bastante demorado, as informações solicitadas são poucas, na verdade, apenas o nome do pacote, modelo de distribuição (APK ou AAB), e a opção por assinatura com criação da keystore. A partir do fornecimento das informações, a compilação é iniciada, e pode ser acompanhada pelo endereço exibido no console.

Captura da tela No Expo - Geração da keystore no processo de compilação para o ambiente Expo.

Ao final, é exibido o link para baixar o arquivo, bastando acessá-lo e fazer o upload do arquivo baixado para a loja do Google. Note que não tivemos que nos preocupar em gerar a chave nem configurar o projeto para uma compilação assinada.

Captura da tela No Expo -Término do processo de compilação para o ambiente Expo.

Um procedimento similar é oferecido para iOS, devendo ser definido um nome para o pacote, escolhida a opção entre gerar o arquivo ou publicar na loja Apple, e introduzidas as credenciais da conta como Apple Developer. O pacote pode ser assinado diretamente pela loja Apple, ou utilizando um certificado fornecido pelo ambiente Expo.

O comando para gerar a distribuição do iOS é apresentado a seguir.

Terminal

```
1  expo build:ios
```

Testes no React Native

No vídeo a seguir, demonstraremos o uso de testes na publicação de aplicativos com boa qualidade, usando React Native.

Vem que eu te explico!

Os vídeos a seguir abordam os assuntos mais relevantes do conteúdo que você acabou de estudar.

Performance Tuning

4:14min

Shipping no React Native

4:54min

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

No ambiente do React Native, a biblioteca padrão para testes unitários é o Jest, que utiliza o comando expect, recebendo a chamada de uma função a ser testada como parâmetro, e compara o resultado da execução através de operadores específicos. Qual dos operadores seria utilizado para verificar se o valor resultante é igual ao esperado?

A toThrow

B toBeGreaterThan

C toHaveProperty

D toBeLessThan

E toBe

Responder

Questão 2

A construção de uma interface fluida e consistente é um fator essencial para o sucesso de um aplicativo, e a utilização de aplicativos para análise de performance tem papel relevante na garantia dessa fluidez. Por meio do Dev Tools, temos acesso a diversas opções para análise de performance durante a execução, e uma característica do ambiente é a de que

A trabalha apenas com dados em modo texto, sem o fornecimento de análises gráficas.

B não mapeia as chamadas internas, decorrentes da invocação de subfunções, viabilizando apenas a análise macroscópica de cada processo.

C permite visualizar a síntese do consumo de memória a partir de instantâneos salvos durante a execução.

D trabalha com amostragens de execução em tempos fixos de um segundo.

E analisa as chamada para funções Type Script ou Java Script através da opção Timings.

Responder

★★★★★

Considerações finais

Para trabalhar com grandes empresas de desenvolvimento, principalmente as que adotam metodologias ágeis, é necessário conhecer metodologias de testes, análise de performance e processo de empacotamento, assuntos que foram analisados aqui, no ambiente do React Native, com a utilização de ferramentas apropriadas, como Jest e Dev Tools.

Além de adotar processos de desenvolvimento organizados, é fundamental conhecer as arquiteturas mais populares, como MVC, Flux e Redux, amplamente utilizadas no mercado, assim como a implementação com base em padrões de desenvolvimento. Os exemplos que foram apresentados demonstram a utilização de arquiteturas e padrões de forma simples, com bases pequenas e interfaces gráficas contendo apenas componentes essenciais.

Outro assunto de grande importância para as empresas é o sigilo das informações, algo que exige a utilização de diferentes tipos de criptografia, e aqui discutimos os fundamentos que norteiam a área, bem como a aplicação prática de recursos criptográficos no React Native.

O conjunto de conhecimentos adquiridos trata de assuntos considerados avançados, em termos de programação no React Native, aumentando de forma significativa a possibilidade de participação em equipes de desenvolvimento para grandes empresas do mercado móvel.

Podcast

Ouça o podcast. Nele, apresentamos um resumo dos recursos avançados do React Native que foram estudados neste conteúdo.

00:00 12:37 1x

Referências

BECK, K. **TDD**: Desenvolvimento guiado por testes. 1. ed. Porto Alegre: Bookman, 2010.

BODUCH, A.; DERKS, R. **React and React Native**. 3. ed. Birmingham, UK: Packt Publishing, 2020.

BUGL, D. **Learning Redux**. 1. ed. Birmingham, UK: Packt Publishing, 2017.

ESCUDELARIO, B.; PINHO, D. **React Native**: Desenvolvimento de Aplicativos Móveis dom React. 1. ed. São Paulo: Casa do Código, 2020.

GRZESIUKIEWICZ, M. **Hands-On Design Patterns with React Native**. 1. ed. Birmingham, UK: Packt Publishing, 2018.

PAUL, A; NALWAYA, A. **React Native for Mobile Development**. 2. ed. New York: Apress, 2019.

ZOCHIO, M. **Introdução à Criptografia**. 1. ed. São Paulo: Novatec, 2016.

Explore +

Leia os artigos:

- **MVC vs Flux vs Redux - The Real Differences**, escrito por Vinugayathri, com uma comparação muito interessante das arquiteturas.
- **Redux: Um tutorial prático e simples**, de Pablo Cavalcante, apresentando os elementos essenciais da arquitetura.
- **Shipping React Native Apps with Fastlane**, de Carlos Cuesta e conheça a ferramenta Fastlane para publicação de aplicativos React Native.

Acesse a documentação oficial do Expo-Crypto, com todos os algoritmos disponíveis.

Busque a documentação oficial do CryptoJS, no GitBook, com diversos exemplos envolvendo todos os algoritmos oferecidos pela biblioteca.

Baixar conteúdo