Conexão Remota com React Native

Componentes e recursos para conexão remota presentes no framework de desenvolvimento para dispositivos móveis React Native.

Prof. Alexandre Paixão

1. Itens iniciais

Propósito

Conhecer os componentes, as ferramentas e as técnicas inerentes ao desenvolvimento de aplicativos mobile utilizando o framework React Native que façam uso de recursos de conexão remota.

Preparação

Para acompanhamento do conteúdo e da codificação dos exemplos a serem apresentados ao longo deste conteúdo, será necessária a utilização de uma IDE (*Integrated Development Environment*), sendo recomendado o Visual Studio Code – software gratuito. Além disso, será preciso configurar o ambiente de desenvolvimento e testes, onde diferentes configurações e ferramentas poderão ser usadas. Para mais detalhes a respeito dessa etapa, o site oficial do React Native poderá ser consultado.

Objetivos

- Listar os principais componentes para conexão em rede
- Aplicar a persistência remota usando a arquitetura REST com controle de acesso
- Descrever o modelo Offline First

Introdução

Um software, e consequentemente um aplicativo, é composto por diversas funcionalidades que, no que tange à visão dos programadores, são representadas por códigos-fonte (dependendo do paradigma de programação utilizado) separados em classes, métodos ou funções, cada um com a sua responsabilidade.

No modelo cliente X servidor, temos os códigos responsáveis pela interação com o usuário (cliente ou front-end) e os responsáveis pelo processamento, pelas regras de negócio, conexão com bancos de dados etc. (o servidor ou back-end). Nesse sentido, podemos dizer que um aplicativo mobile, seguindo as melhores práticas, pode conter tanto códigos de front-end quanto de back-end.

Neste conteúdo, serão discutidos tais aspectos no que tange à comunicação do aplicativo com recursos externos e quanto a possuir em si a lógica que, normalmente, fica no back-end (modelo Offline First). Inicialmente, serão listados os componentes da linguagem React Native para conexão em rede, prosseguindo com a aplicação prática da persistência remota com controle de acesso, utilizando uma API REST. Por fim, será apresentado o modelo Offline First.

Introdução

Antes de conhecermos alguns componentes para acesso a recursos em rede, é importante contextualizar os tipos de recursos que, normalmente, são consumidos em um aplicativo/aplicação. Nesse sentido, e não limitados a eles, podermos citar os seguintes exemplos, dentre outros:

- Login / Validação de credenciais de acesso.
- Consumo de dados armazenados em bancos de dados ou em APIs de terceiros.
- Consumo de recursos disponíveis por meio de APIs de terceiros.
- Persistência de dados.

Em cada um dos exemplos citados, nosso aplicativo, seguindo o mesmo modelo de uma página Web, executará o fluxo composto por realizar uma requisição, seguido do recebimento, tratamento e, na maioria das vezes, da exibição do seu retorno. Tais requisições poderão ser feitas a APIs ou WebServices (sejam eles SOAP ou REST), ou, menos comum, diretamente a scripts escritos em linguagens de programação de back-end e disponíveis por uma URL.

Componentes para conexão

No framework React Native, está disponível, nativamente, um componente que permite a conexão com recursos remotos, o Fetch API (também disponível no JavaScript em ambiente Web). Por meio do Fetch, é possível consumir e enviar dados utilizando os diferentes métodos HTTP (GET, POST etc.) e em diferentes formatos (JSON, XML, texto puro etc.). Além desse componente nativo, estão disponíveis bibliotecas como a Axios – uma das mais utilizadas –, entre outras. A seguir, será demonstrada a utilização da Fetch API e da Axios.

Fetch API

Como mencionado, a Fetch API é uma biblioteca disponível nativamente em React Native para o consumo de recursos externos. O fragmento de código a seguir mostra a sintaxe de uma requisição simples a uma API REST utilizando tal componente:

```
javascript

import React, { useEffect, useState } from 'react';
 import { FlatList, Text, View } from 'react-native';

 export default App = () => {
   const [isLoading, setLoading] = useState(true);
   const [data, setData] = useState([]);

   useEffect(() => {
     fetch('https://api.stackexchange.com/2.3/articles?
order=desc&sort=activity&site=stackoverflow')
       .then((response) => response.json())
       .then((json) => setData(json))
       .catch((error) => console.error(error))
       .finally(() => setLoading(false));
   }, []);
   return (


       {isLoading ? Loading... :
       (
           Dados do StackOverFlow:
            index}
             renderItem={({ item }) => (
               {'[' + item.tags + ']' + '\n' + item.title + '\n\n' }
             )}
           />

       )}

   );
 };
```

No exemplo, podemos ver a sintaxe da Fetch API ao recuperar os recursos de uma API Rest. Nesse caso, foi utilizado o método HTTP GET e setado, a partir do objeto “response”, o método “json()”, informando o tipo de dado a ser transferido. Em seguida, o retorno foi atribuído ao state data, utilizado como datasource do componente FlatList.

Enviando dados em uma conexão remota

O próximo exemplo demonstrará a utilização da Fetch API para o envio de dados utilizando o método HTTP POST para um endpoint REST. Vamos ao fragmento de código:

```
javascript

fetch('https://api.com/endpoint/', {
 method: 'POST',
 headers: {
     Accept: 'application/json',
     'Content-Type': 'application/json'
 },
 body: JSON.stringify({
     nome: 'Alexandre',
     cpf: '000.000.000-00'
 })
 });
```

Note que, para requisições POST, há pequenas diferenças na sintaxe do Fetch:

- método HTTP definido pelo parâmetro “method”;
- cabeçalho da requisição e o tipo de dado a ser transferido definidos pelo parâmetro “header” e “Accept”;
- definição dos dados a serem enviados, formatados como string JSON e definidos pelo parâmetro “body”.

Outros recursos da Fetch API

Além dos recursos demonstrados nos exemplos anteriores (uma requisição GET e outra POST), a Fetch API dispõe de uma série de outros parâmetros e funcionalidades. Logo, para se aprofundar, visite a sua documentação a partir do site oficial do React Native.

Biblioteca Axios

Além da Fetch API, há outras bibliotecas disponíveis em React Native para a conexão e utilização de recursos remotos. Entre elas, destaca-se a Axios.

Antes de usarmos essa biblioteca, precisaremos realizar sua instalação. Para isso, execute o comando a seguir (ou o equivalente com o seu gerenciador favorito de dependências):

```
npm install axios
```

Atenção

Cabe ressaltar que a Axios e outras bibliotecas fazem uso da API nativa XMLHttpRequest.

Vejamos um exemplo utilizando Axios – o mesmo exemplo de GET mostrado anteriormente:

```
javascript

import React, { useEffect, useState } from 'react';
 import { FlatList, Text, View } from 'react-native';
 import axios from 'axios';

 export default App = () => {
     const [isLoading, setLoading] = useState(true);
     const [data, setData] = useState([]);

     useEffect(() => {

     axios.get('https://api.stackexchange.com/2.3/articles?
order=desc&sort=activity&site=stackoverflow')
          .then(function (response) {
          // handle success
          console.log(response.data);
          setData(response.data);
          })
          .catch(function (error) {
          // handle error
          console.log(error);
          })
          .then(function () {
          // always executed
          setLoading(false);
          });

     }, []);
     return (


          {isLoading ? Loading... :
          (
              Dados do StackOverFlow:
               index}
              renderItem={({ item }) => (
                  {'[' + item.tags + ']' + '\n' + item.title + '\n\n' }
              )}
              />

          )}

     );
 };
 ;
```

Ao analisarmos o código, podemos perceber que, em linhas gerais, o funcionamento da biblioteca Axios é semelhante ao da Fetch. Uma diferença a ser destacada é a “Response Schema” – repare que no caso de sucesso é usado no método “setData” ou “response.data”. O objeto Response tem a seguinte estrutura:

```
javascript

{
 // `data` contém a resposta proveniente do recurso externo / recurso acessado através da
requisição
 data: {},

 // `status` contém o código do status HTTP referente à resposta recebida a partir da
requisição
 status: 200,

 // `statusText` contém o texto do status HTTP referente à resposta recebida a partir da
requisição
 statusText: 'OK',

 // `headers` contém o cabeçalho HTTP retornado a partir da requisição
 headers: {},

 // `config` contém os parâmetros passados através do Axios na requisição
 config: {},

 // `request` contém a requisição original que gerou a resposta em questão
 request: {}
 }
```

Cada um desses parâmetros pode ser acessado conforme mostrado a seguir:

```
javascript

axios.get('/user/12345')
   .then(function (response) {
     console.log(response.data);
     console.log(response.status);
     console.log(response.statusText);
     console.log(response.headers);
     console.log(response.config);
   });
```

Realizando uma requisição POST com Axios

A exemplo do que vimos com a requisição GET, a requisição POST utilizando Axios é bastante similar à realizada com a Fetch API. Veja o fragmento a seguir:

```javascript
const dados = { 'nome': 'Alexandre', 'cof': '000.000.000-00' };

axios.post('https://api.com/endpoint/', dados)
```

Dica

Após executado o método, a resposta (“response”) poderá ser tratada conforme visto no método GET.

Gerenciando múltiplas instâncias Axios

Normalmente, em um projeto real, é comum realizarmos diversas chamadas para o mesmo endpoint / recurso externo. Nesses casos, é conveniente criarmos e utilizarmos uma única instância do Axios e, então, importarmos tal instância em todos os lugares em que precisarmos de uma conexão com o recurso remoto em questão. O fragmento de código abaixo demonstra como criar uma instância e, em seguida, é possível ver como importar tal instância no script em que ela será utilizada.

```javascript
import axios from 'axios';

const instancia = axios.create({
  baseURL: 'https://api.com/endpoint/',
  timeout: 1000,
  headers: {'X-Custom-Header': 'foobar'}
});

export default instancia;
```

```javascript
import instancia_axios from 'axios_instancia';

instancia_axios.get('/user?ID=12345')
  .then(function (response) {
    // handle success
    console.log(response);
  })
  .catch(function (error) {
    // handle error
    console.log(error);
  })
  .then(function () {
    // always executed
  });
```

Repare que no último fragmento de código, no lugar de importar diretamente a biblioteca Axios, é feita a importação do componente declarado no código imediatamente anterior, ou seja, “axios_intancia.js”. Com isso, a chamada ao recurso externo é feita por meio da instância denominada “instancia_axios” – à qual a importação do “axios_instancia.js” foi atribuída.

Componente Fetch API para conexão com recursos remotos

No vídeo a seguir, abordaremos as funcionalidades do componente Fetch API do React Native, incluindo demonstração de GET E POST.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vem que eu te explico!

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

Apresentação da biblioteca Axios

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Gerenciando múltiplas instâncias com Axios

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Estruturas condicionais

Neste vídeo, você vai compreender as estruturas condicionais do JavaScript, entre elas as instruções if, else, else if e switch. Confira!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Segundo Flanagan (2011), as estruturas de decisão, também conhecidas como condicionais, são instruções que executam ou pulam outras instruções, dependendo do valor de uma expressão especificada. São os pontos de decisão do código, também conhecidos como ramos, uma vez que podem alterar o fluxo do código, criando um ou mais caminhos.

Aprofundando o conceito

Para melhor assimilação do conceito de estruturas condicionais, vamos usar um exemplo a partir do código construído anteriormente, como veremos a seguir:

```
javascript




 Resultado da Multiplicação:
```

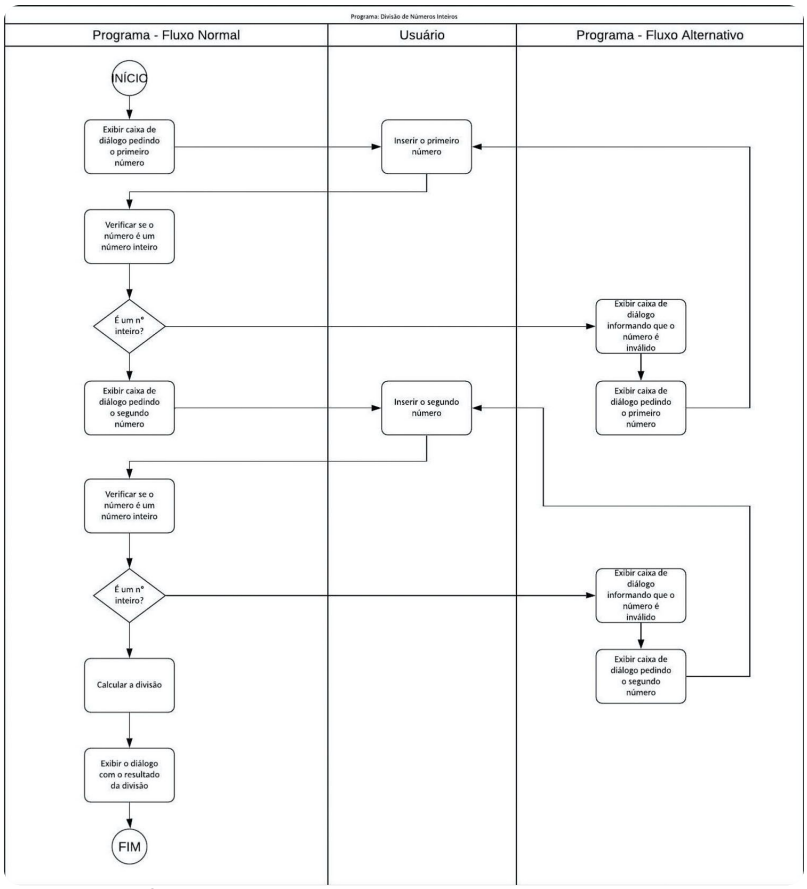
As orientações do programa afirmam que deve ser realizada a divisão de dois números inteiros positivos.

O que acontece se o usuário inserir um número inteiro que não seja positivo? Ou como forçá-lo a inserir um número positivo?

Para essa função, podemos utilizar uma condição, ou seja, se o usuário inserir um número inteiro não positivo, deve-se avisar que o número não é válido, solicitando que seja inserido um número válido.

Nesse caso, o fluxo normal do programa é receber dois números positivos, calcular a divisão e exibir o resultado. Perceba que a condição cria um novo fluxo, um novo ramo, em que outro diálogo é exibido, e o usuário é levado a inserir novamente o número.

O fluxo normal e o fluxo resultado da condicional podem ser vistos na imagem a seguir, em que são apresentados os passos correspondentes ao nosso exercício, separando as ações do programa e as do usuário.



Fluxo normal e fluxo alternativo.

Repare que a verificação “é um nº inteiro positivo” permite apenas duas respostas: “sim” e “não”. Essa condição, mediante a resposta fornecida, é responsável por seguir o fluxo normal do código ou o alternativo.

O fluxograma de exemplo foi simplificado para fornecer mais detalhes. Logo, a respectiva notação padrão não foi utilizada em sua confecção.

Nas linguagens de programação, utilizamos as instruções condicionais para implementar o tipo de decisão apresentado no exemplo. Em JavaScript, estão disponíveis as instruções "if/else" e "switch", como veremos a seguir.

Instrução “If”

A sintaxe da instrução "if/else" em JavaScript possui algumas formas. A primeira e mais simples é apresentada do seguinte modo:

if (condição) instrução

Nessa forma, é verificada uma única condição. Caso seja verdadeira, a instrução será executada. Do contrário, não. Antes de continuarmos, cabe destacar os elementos da instrução:

- É iniciada com a palavra reservada “if”.
- É inserida, dentro de parênteses, a condição (ou condições).
- É inserida a instrução a ser executada, caso a condição seja verdadeira.

Outro detalhe importante: caso exista mais de uma instrução para ser executada, é necessário envolvê-las em chaves. Veja o exemplo:

```javascript
if (condição1 && condição2){
instrução1;
instrução2;
}
```

Nesse segundo caso, além de mais de uma instrução, também temos mais de uma condição. Quando é necessário verificar mais de uma condição, em que cada uma delas precisa ser verdadeira, utilizamos os caracteres “&&”.

Na prática, as instruções 1 e 2 só serão executadas caso as condições 1 e 2 sejam verdadeiras. Vamos a outro exemplo:

```javascript
if (condição1 || condição2){
instrução1;
instrução2;
}
```

Repare que, nesse código, os caracteres “&&” foram substituídos por “||”. Esses últimos são utilizados quando uma ou outra condição precisa ser verdadeira para que as instruções condicionais sejam executadas.

> E o que acontece se quisermos verificar mais condições?

Nesse caso, podemos fazer isso tanto para a forma em que todas as condições precisam ser verdadeiras, separadas por “&&”, quanto para a forma em que apenas uma deve ser verdadeira, separadas por “||”. Além disso, é possível combinar os dois casos na mesma verificação. Veja o exemplo:

```javascript
if ( (condição1 && condição2) || condição3){
instrução1;
instrução2;
}
```

Nesse fragmento, as duas primeiras condições são agrupadas por parênteses. A lógica aqui é a seguinte:

Execute as instruções 1 e 2 SE as condições 1 e 2 forem verdadeiras OU se a condição 3 for verdadeira.

Por fim, há outra forma: a de negação.

> Como verificar se uma condição é falsa (ou não verdadeira)?

Veremos a seguir:

```
javascript

if (!condição1){
instrução1;
instrução2;
}
```

O sinal “!” é utilizado para negar a condição. As instruções 1 e 2 serão executadas caso a condição 1 não seja verdadeira.

Vamos praticar?

Nos três emuladores de código a seguir, apresentamos as estruturas de decisão vistas até o momento. No primeiro emulador, temos o uso da estrutura de decisão “if” de maneira simples, contendo apenas uma única condição:

Conteúdo interativo

esse a versão digital para executar o código.

Já no emulador seguinte, a estrutura de decisão “if” é implementada com duas condições, além dos operadores lógicos AND (&&) e OR (||):

Conteúdo interativo

esse a versão digital para executar o código.

Por fim, no emulador a seguir, temos a estrutura “if” sendo usada de uma maneira mais elaborada, com mais de duas condições, combinação dos operadores && e ll, assim como o uso do operador lógico de negação NOT (!):

Conteúdo interativo

esse a versão digital para executar o código.

Instrução “else”

A instrução “else” acompanha a instrução “if”. Embora não seja obrigatória, como vimos nos exemplos, sempre que “else” for utilizado, deve vir acompanhado de “if”. O “else” indica se alguma instrução deve ser executada caso a verificação feita com o “if” não seja atendida. Vejamos:

```
javascript

if(número fornecido é inteiro e positivo){
     Guarde o número em uma variável;
}else{
     Avise ao usuário que o número não é válido;
     Solicite ao usuário que insira novamente um número;
}
```

Perceba que o “else” (senão) acompanha o “if” (se). Logo, SE as condições forem verdadeiras, faça isto; SENÃO, faça aquilo.

No último fragmento, foi utilizado, de modo proposital, **português-estruturado** nas condições e instruções. Isso porque, mais adiante, você mesmo codificará esse "if/else" em JavaScript.

Português estruturado

Linguagem de programação ou pseudocódigo que utiliza comandos expressos em português.

Instrução “else if”

Veja o exemplo a seguir:

```
javascript

if (numero1 < 0){
     instrução1;
     }else if(numero == 0){
          instrução2;
     }else{
          instrução3;
     }
```

Repare que uma nova instrução foi usada no fragmento. Trata-se de “else if”, instrução utilizada quando queremos fazer verificações adicionais sem agrupá-las todas dentro de um único “if”. Além disso, ao utilizarmos essa forma, caso nenhuma das condições constantes no “if” e no(s) “if else” seja atendida, a instrução “else” será executada obrigatoriamente ao final.

Recomendação

Otimize os códigos presentes nos emuladores anteriores usando o “else if”. Como exemplo, apresentamos o código do primeiro emulador modificado, no qual as quatro estruturas de decisão com “if” foram transformadas em uma única estrutura de decisão.

Note que, antes, eram geradas duas saídas redundantes (“a é maior que b” e “b é menor que a”), pois se tratava de quatro estruturas independentes. Por isso, todas elas eram avaliadas. Isso não ocorrerá mais com o uso de uma estrutura de decisão composta de “if” e “else if”, pois, quando a primeira condição verdadeira for encontrada (“a é maior que b”), nenhuma das outras condições será avaliada. Logo, teremos:

```javascript
var a = 10;
var b = 3;

console.log ("if com uma única condição:");
if (a > b){
     console.log("a é maior que b");
} else if (a == b){
     console.log("a é igual a b");
} else if (a < b){
     console.log("a é menor que b");
} else if (b < a){
     console.log("b é menor que a");
}
```

Instrução “switch”

A instrução “switch” é bastante útil quando uma série de condições precisa ser verificada. É bastante similar à instrução “else if”. Vejamos:

```javascript
switch(numero1){
     case 0:
          instrução1;
          break;
     case 1:
          instrução2;
          break;
     default:
          instrução3;
          break;
}
```

De maneira geral, o switch é usado quando há uma série de condições, nas quais diversos valores para a mesma variável são avaliados. Vamos detalhar o código anterior:

- Após o “swicth” dentro de parênteses, temos a condição a ser verificada.
- A seguir, temos os “case”, em quantidade equivalente às condições que queremos verificar.
- Depois, dentro de cada “case”, temos a(s) instrução(ões) e o comando “break”.
- Por fim, temos a instrução “default”, que será executada caso nenhuma das condições representadas pelos “case” seja atendida.

Entendendo o que é REST

Antes de tratarmos da aplicação prática da persistência remota, é importante conhecermos um pouco mais sobre o REST. Essa sigla, cujo significado em português é “Transferência de Estado Representacional” (do inglês *Representational State Transfer*), pode ser definida como uma abstração da arquitetura da Web, composta por um conjunto de princípios, restrições e definições, cuja principal função é permitir a comunicação entre diferentes aplicações.

> Em termos mais técnicos, REST é uma arquitetura de software usada para a criação de serviços web (WebServices) que permitem o acesso e a manipulação de recursos (chamados recursos Web) identificados por suas URLs.

Veja este exemplo de requisição REST:

Exemplo

GET http://localhost/usuarios

No exemplo, temos:

- o verbo HTTP usado na requisição – GET
- a URL do recurso requisitado, composta pelo endereço “http://localhost”
- o recurso “usuarios”

Ao analisar essa requisição, podemos entender que está sendo solicitada uma listagem de usuários. Veja este novo exemplo:

Exemplo

POST http://localhost/usuarios - data {nome: alexandre}

Aqui temos uma requisição hipotética, cuja função é demonstrar os componentes de uma requisição que utiliza o verbo HTTP POST. Nesse caso, no lugar de recuperar informações (como realizado no primeiro exemplo), é realizada a persistência de um dado. Vamos aos detalhes da requisição:

- Verbo HTTP: POST.
- URL do recurso: http://localhost.
- Recurso: usuários.
- Dados (representados pela variável data e em formato de string JSON): {nome: alexandre}.

Requisições e Respostas na arquitetura REST

Em linhas gerais, e conforme visto nos exemplos, podemos dizer que essa arquitetura faz uso do modelo “Cliente x Servidor”: de um lado temos o cliente que realiza uma requisição, e do outro lado temos o Servidor que recebe, processa e devolve uma resposta.

Nesse processo de requisição e resposta, temos, entre outros componentes, os verbos HTTP, o cabeçalho da requisição e o formato de transmissão de dados. Em REST, utiliza-se, por padrão, o formato JSON. A string {nome: alexandre} é um exemplo desse formato, composto por par (ou pares) de chave: valor. Logo, tanto a requisição, quando envia dados do Cliente para o Servidor, como a resposta fazem uso dele.

Persistindo dados remotamente com React Native

Para realizar a persistência de dados em um aplicativo escrito com o framework React Native, faremos uso dos componentes para conexão em rede vistos anteriormente. Logo, alguns exemplos e também boas práticas no que diz respeito ao envio de requisições e tratamento de respostas – em caso de sucesso ou não - serão demonstrados.

Antes de continuarmos, cabe ressaltar que os códigos referentes ao lado Servidor, ou seja, os códigos da API que receberá as nossas requisições, não serão apresentados, uma vez que tal assunto foge ao escopo de nosso conteúdo. Entretanto, não se preocupe: serão demonstradas algumas bibliotecas para funções específicas e que são disponibilizadas gratuitamente por seus fornecedores.

Enviando dados para uma API REST

Inicialmente, vamos usar o mesmo código visto no módulo 1, em que uma requisição POST é submetida. Para facilitar o entendimento, segue a descrição da requisição:

- Componente para conexão remota: axios (lembre-se de adicioná-lo ao projeto – npm install axios).
- Método HTTP: Post.
- URL do recurso externo: https://reqbin.com/echo/post/json (recurso de teste que devolve como resposta uma mensagem de sucesso).
- Dados enviados: string JSON contendo as chaves Id, Customer, Quantity e Price.

```javascript
import React, { useEffect, useState } from 'react';
 import { StyleSheet, Text, View } from 'react-native';
 import axios from 'axios';

 export default function App() {
   const [isLoading, setLoading] = useState(true);
   const [data, setData] = useState('');

   useEffect(() => {
     setLoading(true);

     axios.post('https://reqbin.com/echo/post/json', {
       data: {
         ''Id'': 78912,
         ''Customer'': ''Jason Sweet'',
         ''Quantity'': 1,
         ''Price'': 18.00
       },
     })
       .then(function (response) {
         // handle success
         console.log(response.data);
         setData(response.data);
       })
       .catch(function (error) {
         // handle error
         console.log(error);
       })
       .then(function () {
         // always executed
         setLoading(false);
       });

   }, []);

   return (

     Resultado da Requisição: {data.success}

   );
 }

 const styles = StyleSheet.create({
   container: {
     flex: 1,
     backgroundColor: '#fff',
     alignItems: 'center',
     justifyContent: 'center',
   },
 });
```

A resposta da requisição está disponível na primeira instrução “.then”, por meio do objeto “response”. Mediante essa instrução, e da instrução “catch”, tratamos o retorno da requisição realizada, tanto em termos de sucesso quanto em termos de erro.

Atenção

Tratar os possíveis erros de uma requisição é fundamental para fornecer a melhor experiência possível ao usuário. Do contrário, e caso aconteça um erro sem que o tratemos, o usuário ficará com a sensação de que a requisição não foi realizada ou que está demorando muito, podendo, inclusive, desistir de utilizar o APP em virtude desses fatores. Outra prática importante é fornecer um elemento visual que informe ao usuário que a requisição está sendo processada. Normalmente, utilizamos a imagem conhecida como “loading” ou mesmo uma mensagem textual com tal informação. Olhando o código, podemos controlar a exibição desse elemento por meio do state “isLoading”, definido como “true” no momento que a requisição é iniciada e como “false” quando ela é finalizada.

A API remota utilizada nesse exemplo retorna uma simples mensagem {success: true}. É comum termos APIs que retornam um objeto mais complexo, como a representação de um novo recurso que tenha sido persistido por meio de nossa requisição – um novo usuário, por exemplo. Em ambas as situações, podemos tanto tratar o retorno apenas verificando, dentro do próprio componente utilizado para conexão o retorno, como o utilizar para exibição em algum outro componente de nossa aplicação. No código, o retorno é inserido dentro do state “data”, que é utilizado no componente . Com isso o retorno será exibido diretamente na tela do aplicativo.

Persistindo dados com Autenticação

O exemplo de persistência demonstrado no exemplo anterior é totalmente funcional. Entretanto, dificilmente um recurso REST remoto é disponibilizado para utilização sem que seja necessário autenticar o usuário que está consumindo-o. Em termos de autenticação, há vários formatos disponíveis para proteger APIs, como autenticação básica (HTTP Basic Auth), autenticação por Token (HTTP Bearer Tokens, OAuth2 etc.), entre outros. Veremos no próximo exemplo como realizar uma requisição POST, para um novo recurso, utilizando autenticação com Bearer Token.

Dica

Para executar o código seguinte e realizar chamadas para API “gorest.co.in”, é necessário acessar o site e criar, gratuitamente, um token de acesso. Após isso, substitua o token fictício constante no código pelo seu token.

Alguns comentários prévios sobre o código:

- A constante “token” armazena o token fornecido pela API. Lembre-se de substituir seu valor pelo seu próprio token antes de executar o código.
- Na requisição do Axios, foi adicionado um novo parâmetro, “headers”, por meio do qual, entre outras informações, podemos incluir o tipo de autenticação usado.
- O retorno da API (mostrado logo depois do código), em caso de sucesso, consiste em uma nova instância do recurso “usuário” que criamos por meio de nossa requisição, contendo os dados dele.

javascript

```
import React, { useEffect, useState } from 'react';
 import { StyleSheet, Text, View } from 'react-native';
 import axios from 'axios';

 export default function App() {
   const [isLoading, setLoading] = useState(true);
   const [data, setData] = useState();

   useEffect(() => {
     setLoading(true);

     const token = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX';
     const dados = { 'name': 'Alexandre', 'gender': 'male', 'email':
'alexandre@email.com.br', 'status': 'active' };

     axios.post('https://gorest.co.in/public/v1/users', dados, {
       headers: {
         'Authorization': `Bearer ${token}`
       },
     })
       .then(function (response) {
         // handle success
         console.log('Resultado: ');
         console.log(response.data);
         setData(response.data);
       })
       .catch(function (error) {
         // handle error
         console.log(error);
       })
       .then(function () {
         // always executed
         setLoading(false);
       });

   }, []);

   return (

       {JSON.stringify(data)}

   );
 }

 const styles = StyleSheet.create({
   container: {
     flex: 1,
     backgroundColor: '#fff',
     alignItems: 'center',
     justifyContent: 'center',
   },
 });
```

```
javascript

                        {
 ''meta'': null,
 ''data'': {
     ''id'': 2365,
     ''name'': ''Alexandre'',
     ''email'': ''alexandre@email.com.br'',
     ''gender'': ''male'',
     ''status'': ''active''
 }
 }
```

Utilizando o método PUT

Por convenção, embora não seja uma regra, na conexão com recursos REST utilizamos o método POST para a persistência de dados e o método PUT para a atualização. Em termos práticos, o método POST funcionaria nas duas situações. Para fins de prática, e já tendo visto um exemplo de requisição POST, veremos a seguir um exemplo utilizando o PUT.

Vamos ao código, precedido de alguns comentários:

- Perceba que o método HTTP foi alterado para o PUT.
- A URL contém uma “path variable”, conforme definido pela API, onde o código do usuário a ser alterado deve ser informado (esse código foi retornado na requisição anterior, no momento de sua criação).
- No parâmetro “data”, foi passado um JSON semelhante ao utilizado no método POST. No exemplo a chave “name” foi alterada.
- A exemplo da requisição anterior, também foi passado como parâmetro o token para acesso ao recurso remoto.
- Como resultado da requisição, será retornada uma string JSON (mostrada logo após o código) contendo os dados atualizados do usuário – ou seja, seu “name”.

javascript

```
import React, { useEffect, useState } from 'react';
 import { StyleSheet, Text, View } from 'react-native';
 import axios from 'axios';

 export default function App() {
   const [isLoading, setLoading] = useState(true);
   const [data, setData] = useState();

   useEffect(() => {
     setLoading(true);

     const token = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
     const dados = { 'name': 'Alexandre Paixão', 'gender': 'male', 'email':
'alexandre@email.com.br', 'status': 'active' };

     axios.put('https://gorest.co.in/public/v1/users/2365', dados, {
       headers: {
         'Authorization': `Bearer ${token}`
       },
     })
       .then(function (response) {
         // handle success
         console.log(response.data);
         setData(response.data);
       })
       .catch(function (error) {
         // handle error
         console.log(error);
       })
       .then(function () {
         // always executed
         setLoading(false);
       });

   }, []);

   return (

       {JSON.stringify(data)}

   );
 }

 const styles = StyleSheet.create({
   container: {
     flex: 1,
     backgroundColor: '#fff',
     alignItems: 'center',
     justifyContent: 'center',
   },
 });
```

```
javascript

{
 ''meta'': null,
 ''data'': {
     ''email'': ''alexandre@email.com.br'',
     ''name'': ''Alexandre Paixão'',
     ''gender'': ''male'',
     ''status'': ''active'',
     ''id'': 2365
 }
 }
```

Controle de Acesso com OAuth2

Os últimos exemplos de conexão remota vistos fizeram uso de autenticação, via Bearer Token, para acessar e consumir uma API externa. Esse controle de acesso é um recurso muito utilizado, além de necessário e importante, para proteger e controlar o acesso a APIs e recursos. O método até aqui utilizado consiste na obtenção de um Token diretamente junto ao provedor do recurso e do seu envio nas requisições. Além desse modelo, há outros, sendo o mais utilizado o OAuth2.

> O OAuth2 costuma ser definido como framework de segurança, como protocolo, como estrutura de autorização, entre outras formas.

Independentemente disso, e considerando que sua definição mais apropriada é a de Framework de Autorização (contando, inclusive, com uma **RFC** própria, a RFC6749) é importante, no que tange ao processo de desenvolvimento de aplicativos mobile, e mais especificamente à comunicação deles com recursos remotos, entendermos a estrutura do OAuth2 e como implementá-lo. Nesse sentido, vamos começar pelos papéis existentes no OAuth2. Isso nos ajudará a entender a separação de responsabilidades, ou seja, a perceber que parte da lógica ficará no aplicativo e qual parte dela ficará no servidor que possui os recursos que desejamos acessar.

RFC

Uma RFC, do inglês Request for Comments, é uma série de publicações com a finalidade de documentar padrões, serviços e protocolos oficiais da internet, sendo mantida pela IETF (Internet Engineering Task Force).

Definição de Papéis no OAuth2

No OAuth2, existem quatro papéis:

- O dono/proprietário do recurso
- O Cliente
- O Servidor de Recurso
- O Servidor de Autorização

No primeiro papel, temos o recurso que desejamos consumir. No segundo, Cliente, temos o nosso aplicativo. Os papéis seguintes se destinam a controlar o acesso do Cliente ao Recurso.

Uma das grandes diferenças, e quiçá vantagem do modelo OAuth2, é permitir o acesso a recursos mediante tokens, sem que seja necessário, por exemplo, utilizar credenciais como usuário e senha. Além disso, é comum determinar um tempo de vida útil para cada token. Logo, o acesso aos recursos fica condicionado à obtenção e contínua validação dele.

Esquemas para Utilização do OAuth2

Além dos papéis, o OAuth2 possui ainda alguns diferentes esquemas e fluxos de utilização. É por meio desses esquemas e de seus respectivos fluxos que saberemos como desenvolver nosso aplicativo a fim de consumir um recurso externo. Logo, tal escolha não depende somente do desenvolvedor do aplicativo, mas também do fornecedor do recurso. Outro ponto importante é que cada esquema traz algumas implicações para a arquitetura do aplicativo, como o manuseio e armazenamento de informações sensíveis como senhas de usuários.

A especificação OAuth2 estabelece quatro fluxos de autorização:

1

Autorização por Código

Nesse fluxo, o token de acesso é obtido a partir de um servidor de autorização, que age como intermediário entre o cliente e o fornecedor do recurso. Logo, em vez de fornecer o token diretamente ao cliente mediante solicitação, o fornecedor o redireciona para um servidor de autorização, que, por sua vez, autentica tanto o fornecedor quanto o cliente, fornecendo a este último o código de acesso / token.

2

Fluxo Implícito

Trata-se de uma simplificação do fluxo anterior, otimizado e implementado, normalmente, para rodar em navegadores web. Nesse fluxo, ao interagir em uma página web, o cliente recebe diretamente um token de acesso, sem que seja gerado nenhum código de autorização intermediário (e que, posteriormente, precisaria ser usado para, então, obter o token).

3

Credenciais de Senha do Proprietário do Recurso

Com a utilização desse fluxo, o token é obtido mediante a utilização de credenciais (normalmente usuário e senha) fornecidas pelo proprietário do recurso. Tal fluxo é recomendado apenas quando existir um alto grau de confiança entre as partes ou quando outros tipos de fluxos não estiverem disponíveis.

4

Credenciais do Cliente

A especificação do OAuth2 define as Credenciais do Cliente como um fluxo que engloba quaisquer outros fluxos de autenticação não definidos/explicitados anteriormente. Um exemplo de uso poderia contemplar uma prévia definição, entre o Cliente e o Fornecedor, de quais recursos estariam disponíveis, podendo ser acessados diretamente com credenciais do próprio Cliente.

Padrão AppAuth

A RFC6749, que contém as regras do Framework OAuth2, possui uma seção específica para tratar a interação entre aplicativos mobile e servidores de autenticação. Em tal seção, a 9, estão previstas duas abordagens: “embedded user-agent” e “external user-agent”. Além disso, mais recentemente, foi criada outra especificação, a RFC8252 – Oauth 2.0 for Native Apps. Tal RFC apresenta uma série de recomendações como

sugestões a serem seguidas pelos desenvolvedores a fim de garantir a segurança e o controle de acesso no consumo de recursos remotos. Logo, a leitura dessa especificação é fortemente recomendada.

Controle de Acesso na Prática

Vimos, até aqui, tanto a aplicação prática quanto alguns aspectos técnicos e teóricos sobre o controle de acesso em aplicativos mobile para o consumo de recursos remotos. Além disso, existem inúmeros recursos disponíveis para implementação e uso de controle de acesso em React Native, a saber:

Google Firebase Authentication

Serviço de autenticação vinculado ao Google Firebase, plataforma do Google para criação de aplicativos móveis.

Auth0

Plataforma comercial, com plano gratuito, que fornece serviços de autenticação e autorização.

FusionAuth

Plataforma comercial, com plano gratuito, que fornece serviço de gerenciamento de acesso seguro.

Keycloak

Plataforma opensource (e também disponível de forma comercial, como serviço) que fornece serviço de gerenciamento de acesso e identidade.

React-native-app-auth

Biblioteca de “bridge” que facilita a comunicação com provedores de acesso OAuth2.

Arquitetura REST com controle de acesso no React Native

No vídeo a seguir, destacaremos os pontos relevantes da arquitetura REST, incluindo uma demonstração de envio de dados para uma API REST.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vem que eu te explico!

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

Utilizando o método PUT

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Controle de acesso com OAuth2

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Considerando o cenário em que uma conexão remota é estabelecida para a persistência de dados – o cadastro de um novo produto –, assinale a alternativa abaixo que corresponda aos elementos que deverão estar presentes na requisição:

A

Os dados do produto e o endereço do recurso externo.

B

Os dados do produto, o endereço do recurso externo, o tipo de dado a ser transferido e um state para armazenar o retorno da requisição.

C

Os dados do produto, o tipo de dado a ser transferido, o endereço do recurso externo e o método HTTP.

D

Um componente React Native para realizar a conexão, os dados do produto, a definição do tipo de dado a ser transferido, o endereço do recurso externo e a definição do método HTTP.

E

Um componente React Native para realizar a conexão, os dados do produto, o endereço do recurso externo e a definição do método HTTP.

A alternativa E está correta.

Para realizar a persistência de dados por meio de uma conexão remota, o React Native dispõe de alguns recursos/bibliotecas, como a Fetch API e o Axios. As requisições realizadas por meio desses componentes possuem certa similaridade, sendo alguns parâmetros obrigatórios, como os dados a serem persistidos, o endereço do recurso remoto, a definição do método HTTP, além do próprio componente de conexão em si.

Em termos de formato de transmissão de dados, ele pode ser omitido, sendo usado por padrão o formato JSON.

Questão 2

Imagine o cenário em que uma requisição é realizada, a partir de um aplicativo mobile, para um recurso remoto que exija a autenticação para acesso a seus serviços. Qual o retorno da requisição dentro do aplicativo mobile caso o token ou as credenciais de acesso não sejam enviados na requisição ou estejam errados?

A

O retorno esperado – recuperação ou persistência de dados – será obtido, uma vez que existe uma relação de confiança previamente estabelecida entre o aplicativo e a conexão remota.

B

Será exibido na tela um alerta informando que não foi possível se conectar com o servidor remoto.

C

Caso a requisição seja tratada em um bloco “catch” – ou com código/lógica equivalente a ele – será possível obter o erro HTTP correspondente (normalmente o código 401 – Unauthorized) ou uma mensagem personalizada (implementada no lado servidor) que indique que não foi possível realizar a conexão. A partir daí, será possível interagir com o usuário do aplicativo informando o ocorrido.

D

Após o envio da requisição e a ocorrência da falha de acesso, a tela do aplicativo ficará branca, mostrando que não há nenhuma informação a ser exibida.

E

Por padrão, os componentes de conexão remota do React Native executam uma rotina preestabelecida em casos de erros de conexão remota que leva o usuário para a página/tela principal do aplicativo, exibindo uma mensagem de erro padrão na tela.

A alternativa C está correta.

As tarefas que consomem recursos remotos, que necessitam ou não de autorização, precisam ser tratadas individualmente pelo desenvolvedor do aplicativo para o caso de ocorrência de falhas. Considerando que o fluxo de disparo de erro é de responsabilidade do fornecedor do recurso, torna-se necessário entender a API utilizada, além da realização de testes em todos cenários possíveis – sucesso ou falha – para conhecimento de suas respectivas respostas.

Aplicativos Online e Offline

Nos módulos anteriores, foram apresentados os componentes para conexão em rede e persistência de dados para a construção de aplicativos mobile utilizando React Native. Nesse sentido, partimos do princípio, em todos os cenários vistos, que os recursos remotos estarão sempre disponíveis ou que os dispositivos nos quais nossos aplicativos serão instalados estarão sempre conectados à internet (independentemente do tipo).

Comentário

Por mais que a infraestrutura de telecomunicações esteja avançando, e muito, ao longo dos anos, ainda nos deparamos com situações em que ficamos com nossos dispositivos móveis “offline”. Nesses casos, o que acontece com os aplicativos que temos instalados? Caso ainda não tenha feito, faça um teste. Em sua maioria, os aplicativos param de funcionar.

Para contornar essa situação, existe um modelo denominado “Offline First”, cuja principal função é a de permitir que um aplicativo que faça uso de recursos remotos funcione normalmente estando ou não conectado à internet.

Ao longo deste módulo, o modelo Offline First será descrito, assim como as técnicas que nos permitirão utilizá-lo em nosso processo de desenvolvimento.

A Arquitetura Offline First

Em termos conceituais, podemos dizer que um aplicativo desenvolvido seguindo os princípios da Arquitetura Offline First é um aplicativo que funciona de modo semelhante, independentemente de possuir ou não conexão com a internet.

Requisitos únicos

Devemos ter em mente que cada aplicativo tem sua própria arquitetura, seus próprios requisitos e suas funcionalidades.

Estratégia

Nesse sentido, podemos ter aplicativos nos quais, por uma decisão estratégica, a arquitetura Offline First não será aplicada.

Em outros casos, podemos ter uma implementação híbrida, ou seja, alguns recursos funcionariam seguindo os princípios da arquitetura e outros não. E, por fim, podemos ter os aplicativos totalmente funcionais independentemente de estarem ou não conectados.

No momento de planejamento do aplicativo, tenha em mente que há técnicas que podem ser utilizadas para a criação seguindo a arquitetura em questão e que sua utilização não é obrigatória, tratando-se de uma recomendação que traz alguns benefícios, mas que também gera preocupações e código extra.

Fluxo do modelo Offline First

Os aplicativos criados utilizando esse modelo devem seguir um fluxo padrão em seu funcionamento:

Acesso à internet
Ao serem iniciados, os aplicativos devem verificar se o dispositivo possui acesso à internet.

Banco de dados embarcado
Em caso negativo, o aplicativo deverá usar um banco de dados embarcado – que fica salvo e disponível apenas no dispositivo de cada usuário – para armazenar os dados provenientes das ações do usuário, assim como disponibilizar uma série de dados que serão necessários para o seu funcionamento.

Sincronização
O aplicativo deve monitorar constantemente o status de conexão do dispositivo para que, tão logo ele se encontre com acesso à internet, seja executado o processo de sincronização, no qual os dados armazenados no banco embarcado serão sincronizados com o banco de dados/recurso remoto.

> Outro ponto importante nesse fluxo é incluir uma etapa de recuperação de dados remotos e inserção no banco embarcado a fim de que esses dados fiquem disponíveis localmente.

Tal processo deve ser executado no primeiro instante em que se verifique possuir conexão com o servidor remoto e deve ser repetido de tempos em tempos – uma boa dica é executá-lo junto com o processo de sincronização, que então envolverá não somente a sincronização dos dados locais no servidor remoto, mas também a atualização dos dados locais com base nos externos.

Construindo um aplicativo utilizando o modelo Offline First

O fluxo apresentado acima indica quais recursos e componentes precisaremos possuir nos aplicativos que implementem a arquitetura Offline First. Em linhas gerais, precisaremos de um componente para controlar a disponibilidade da conexão à internet, de um banco de dados embarcado e de um componente que realize a sincronização entre os dados salvos localmente com o recurso remoto. Em React Native, existem algumas bibliotecas que nos permitem ter todos esses recursos.

A seguir, veremos algumas opções de componentes para a realização das tarefas citadas a fim de estabelecer um ponto de partida para o desenvolvimento de um aplicativo que siga o modelo Offline First.

Banco de Dados Embarcado

Em termos de banco de dados embarcado, ou seja, de um banco de dados que seja disponibilizado juntamente com o aplicativo, há várias opções disponíveis, incluindo bancos relacionais e não relacionais. A escolha de qual utilizar normalmente seguirá uma série de fatores. A dica aqui é pesquisar opções que já possuam ferramentas que permitam a sincronização remota. Seguem algumas opções entre as mais utilizadas com React Native:

- **AsyncStorage;**
- **SQLite;**
- **Firebase;**
- **Realm;**

- **PouchDB;**
- **Watermelon DB.**

Dica

O Realm e o Watermelon possuem mecanismos de sincronização. Comece estudando essas duas opções e, em seguida, pesquise as características dos demais.

O exemplo seguinte apresenta o código correspondente a uma classe, ProdutoSchema, equivalente a uma “tabela” em um banco de dados relacionais, mas implementada no modelo do Realm Database. Realizar a persistência local consiste em criar esse tipo de classes, mediante as quais definimos que dados desejamos armazenar e que serão utilizadas para a persistência e recuperação de dados no banco embarcado.

```
javascript
class ProdutoSchema extends Realm.Object { }
ProdutoSchema.schema = {
  name: 'Produto',
  properties: {
    produto_id: { type: 'int', default: 0 },
    produto_nome: 'string',
    produto_descricao: 'string',
    produto_preco: 'number',
  }
};

//Listagem de Produtos
let listarProdutos = () => {
  return realm_produto.objects('Produto');
}

//Adição de Produtos
let adicionarProdutos = (nomeProduto, descricaoProduto, precoProduto) => {

  const ultimoId = realm_produto.objects('Produto').sorted('produto_id', true)[0];
  const maiorId = ultimoId == null ? 1 : ultimoId.produto_id;
  const proximoId = maiorId != 1 ? maiorId + 1 : maiorId;

  realm_produto.write(() => {
    const prod = realm_produto.create('Produto', {
      produto_id: proximoId,
      produto_nome: nomeProduto.produto_nome,
      produto_descricao: descricaoProduto.produto_descricao,
      produto_preco: precoProduto.produto_preco,
    });
  });
}

export {
  ProdutoSchema,
  listarProdutos,
  adicionarProdutos
}
```

Lógica de Back-end no Aplicativo

Uma característica dos aplicativos que aplicam o modelo Offline First é possuir uma estrutura normalmente vista em aplicações/APIs que ficam no back-end.

Tal estrutura diz respeito aos modelos de dados – normalmente chamados models ou entities, e cujo exemplo vimos anteriormente, muito usados na técnica chamada ORM (Mapeamento Objeto Relacional) e responsáveis por representar os dados a serem manipulados.

Os detalhes dessa modelagem fogem ao escopo deste conteúdo. Logo, é recomendado que você obtenha maiores informações a seu respeito antes de codificar o seu aplicativo.

Gerenciador de Estados

Recomendação

Embora não se trate de um componente obrigatório, pode ser interessante utilizar um gerenciador de estados na aplicação a fim de controlar e centralizar os dados em um “store” que fique disponível em todas as telas da aplicação.

O benefício dessa centralização vem do fato de que, em vez de termos que nos preocupar em recuperar os dados de diferentes componentes, podemos ter acesso a eles a partir de um único lugar.

Em relação às opções para gerenciamento de estados, temos o Context e o Redux. Este último é uma biblioteca externa que fornece as mesmas funcionalidades do Context, mas que oferece mais recursos.

Controle das Funcionalidades Offline versus Online

Para a implementação das funcionalidades responsáveis por permitirem que nosso aplicativo funcione tanto Online quanto Offline, existem duas principais bibliotecas disponíveis em React Native:

React-native-offline	Redux-offline

Essas duas bibliotecas utilizam o Redux e proveem uma série de funcionalidades como permitir que seja verificado, de tempos em tempos, se o aplicativo possui ou não conexão com a internet, além da definição dos métodos a serem executados em cada situação. Veja o fragmento a seguir em que é definido um método para persistência de dados que faz uso de “Redux actions”:

```javascript
const saveData = data => ({
 type: 'SAVE_DATA',
 payload: { data },
 meta: {
     offline: {
     // Aqui é definido o endereço da API remota e a ação a ser executada para
persistência
     effect: { url: '/api/save-data', method: 'POST', json: { data } },
     // Aqui é definida a ação a ser executada após a realização da persistência:
     commit: { type: 'SAVE_DATA_COMMIT', meta: { data } },
     // Aqui é definida a ação a ser executada caso não seja possível, de forma
definitiva, conectar com
     //  o servidor remoto e realizar a persistência dos dados:
     rollback: { type: 'SAVE_DATA_ROLLBACK', meta: { data } }
     }
 }
 });
```

O fluxo definido no código acima segue uma ordem de execução: inicialmente é executada a ação effect, responsável por salvar os dados através da conexão remota. Se essa ação for executada com sucesso, então a ação commit é acionada.

Redux-offline

Caso o aplicativo esteja offline, a biblioteca redux-offline provê os métodos necessários para se aguardar o estabelecimento da conexão a fim de se persistir os dados. Em caso de falha, ou seja, caso não seja possível se conectar, o método rollback é acionado.

Reversão ao estado inicial

Com isso, os dados serão revertidos ao estado anterior ao momento em que o fluxo foi iniciado. Além disso, por meio dessa ação, é possível adicionar uma rotina de notificação ao usuário, informando que não foi possível realizar a operação online, pedindo a ele que tente novamente quando a conexão for reestabelecida.

Outras considerações sobre o Modelo Offline First

Ao longo deste módulo, os conceitos do modelo Offline First e algumas técnicas para sua aplicação foram apresentados. A ideia na apresentação desse conteúdo foi introduzir o assunto e destacar alguns dos cuidados que devemos tomar na construção de nossos aplicativos. Entretanto, é importante ressaltar que há outras questões, a serem consideradas em nosso dia a dia. A seguir, algumas dessas questões serão listadas e apresentadas brevemente:

Autenticação de Usuários e Aplicativos Multiusuários

Tenha em mente que um aplicativo, normalmente, é multiusuário. Logo, ao persistir um dado localmente para posterior sincronização, lembre-se de identificar o usuário que está manipulando cada informação. Sem isso, você poderá ter vários problemas de consistência de dados.

Estrutura da API remota

Caso você não tenha acesso, em termos de programação, à API ou às APIs remotas utilizadas por seu aplicativo, tome o cuidado de estudar as suas documentações a fim de melhor planejar a aplicação do modelo Offline First. Isso é importante porque as diferentes maneiras como as APIs são implementadas podem trazer impactos para o seu aplicativo. Além disso, lembre-se de que você estará reproduzindo parte do modelo de dados utilizado na API de forma local. Logo, eventuais mudanças no modelo remoto implicarão a necessidade de atualização no modelo local.

Interface Otimista

Essa técnica consiste em tornar mais fluída a interação dos usuários com os aplicativos, sobretudo quando os aplicativos dependem do consumo de recursos remotos e, mais ainda, quando tais recursos não estiverem online. Em linhas gerais, a interface otimista consiste em fazer parecer ao usuário que a aplicação é mais rápida do que de fato é.

Para ficar mais fácil de entender, veja o exemplo do aplicativo do Instagram.

Nesse aplicativo, mesmo que você esteja offline, é possível curtir publicações. Quer dizer, a resposta da interface indica, imediatamente, que determinada publicação foi curtida, uma vez que o ícone que indica o sucesso de tal ação é modificado imediatamente após tocado. Entretanto, essa mudança na interface não significa que, de fato, essa informação foi persistida. Por outro lado, a satisfação do usuário foi garantida, cabendo ao aplicativo implementar, então, a persistência real e também informar posteriormente ao usuário caso ela, por algum motivo, não tenha sido realizada de fato.

O modelo Offline First no React Native

Os pontos relevantes e de atenção no desenvolvimento de um aplicativo utilizando o modelo Offline First serão abordados no vídeo a seguir.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vem que eu te explico!

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

Controle das funcionalidades Offline vs Online

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Outras considerações sobre o modelo Offline First

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Em relação à implementação do Modelo Offline First em aplicativos escritos com o framework React Native, é correto afirmar que:

A

O React Native provê mecanismos nativos capazes de, por si só, tornarem possível a implementação do modelo Offline First.

B

O Modelo Offline First é, antes de mais nada, um modelo conceitual, que descreve os cuidados a serem tomados na construção de um aplicativo que utilize recursos remotos. Em React Native, tanto os componentes para conexão remota como os demais envolvidos na implementação do modelo precisam ser instalados de forma adicional.

C

A implementação do modelo Offline First em React Native traz consigo algumas preocupações, incluindo a plataforma – Android ou iOS – para a qual implementaremos nosso aplicativo.

D

A principal limitação para a implantação do modelo Offline First em aplicativos escritos com React Native é a dependência do recurso remoto a ser utilizado. Logo, é correto afirmar que não é possível tornar qualquer aplicativo em um aplicativo que utilize o modelo em questão.

E

A grande vantagem de se aplicar o modelo Offline First usando React Native é que, ao escrevermos localmente os mesmos códigos existente na API remota, tornamos nosso aplicativo mais robusto e totalmente independente de recursos remotos.

A alternativa B está correta.

A codificação de um aplicativo usando o Modelo Offline First consiste na combinação de uma estratégia bem definida e na escolha dos componentes que permitam a aplicação eficiente da estratégia em questão. Em linhas gerais, não é possível aplicar o modelo utilizando apenas os componentes nativos do React Native, sendo necessário instalar bibliotecas adicionais.

Questão 2

Podemos dizer que um aplicativo foi codificado utilizando o Modelo Offline First quando

A

consome recursos disponíveis remotamente; possui um mecanismo para verificar a existência de conexão à internet; possui um banco de dados embarcado; possui mecanismo para sincronização bidirecional de dados.

B

utiliza mecanismos de gestão de dados e um “store” para a centralização de todos os dados de que faz uso.

C

possui um banco de dados embarcado. Em outras palavras, todo aplicativo que possui um banco de dados embarcado é um aplicativo que faz uso do modelo Offline First.

D

possui em si lógica e código normalmente encontrado no back-end.

E

possui uma interface otimista, por meio da qual todas as ações são executadas de maneira ágil, melhorando a experiência do usuário.

A alternativa A está correta.

O conceito de Modelo Offline First se aplica a situações em que há o consumo de recursos externos a partir de um aplicativo. Logo, se um aplicativo não faz uso de recursos remotos, não há por que falar no modelo em questão. Além disso, outras características do Modelo Offline são a existência de mecanismos para a verificação constante da conexão, a utilização de um banco de dados embarcado e a sincronização bidirecional de dados.

Considerações finais

Ao longo deste conteúdo, foram apresentados os recursos do framework React Native para a realização de conexão remota tendo em vista o consumo de recursos externos. Nesse sentido, foram listados os componentes, tanto nativos como bibliotecas externas, que permitem a conexão em rede e a persistência de dados.

Além disso, tanto de maneira prática como conceitual, foram aplicados e descritos alguns mecanismos de controle de acesso inerentes ao processo de consumo de recursos externos.

Por fim, foi descrito o Modelo Offline First, por meio do qual aplicações mobile que fazem uso de conexão remota podem ser escritas, a fim de permitirem o seu uso normal com ou sem conexão à internet.

Podcast

Ouça o podcast a seguir, no qual trataremos da utilização dos tópicos abordados no desenvolvimento mobile com React Native.

Conteúdo interativo

Acesse a versão digital para ouvir o áudio.

Explore +

Visite a documentação oficial do React Native disponível no seu website.

Conheça a especificação técnica que estabelece o modelo OAuth 2.0 disponível no website IETF Datatracker.

Veja também a especificação que trata da aplicação do OAuth2 em Aplicativos, disponível no website IETF Datatracker.

Referências

DENNISS, W. ; BRADLEY, J. **OAuth 2.0 for Native Apps**. Internet Engineering Task Force (IETF): Best Current Practice. United States: October, 2017.