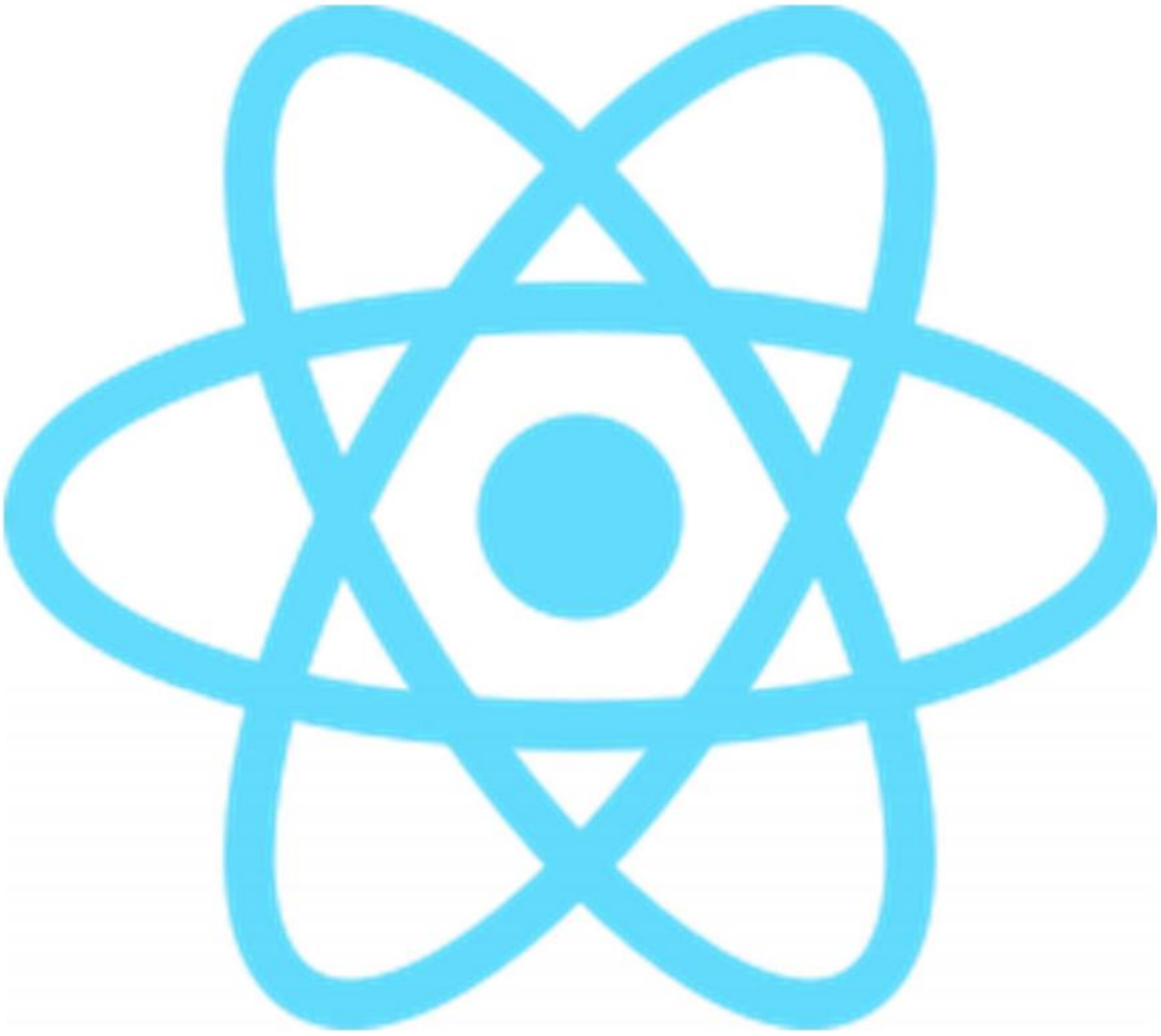
React Native

Conexões Remotas



Gabriel C. Azevedo

É muito difícil qualquer aplicativo desenvolvido atualmente não fazer ao menos uma requisição a um serviço web. Muito comumente fará um login ou consumirá uma API para qualquer outro serviço. A exemplo temos: IoT, jogos, check-outs, transmissões ao vivo, sites, chats, e-mails e outros.

Vamos explorar as duas principais ferramentas para conexão remota no React Native: a API Fetch, nativa do JavaScript, e a biblioteca Axios, bastante popular e avançada. Abordaremos desde os conceitos básicos até técnicas avançadas, com exemplos práticos e explicações aprofundadas sobre arquitetura REST, tratamento de respostas, configuração de requisições, autenticação e muito mais.

Introdução à Arquitetura REST

REST (Representational State Transfer), é um **modelo de arquitetura** para APIs remotas que utilizam o protocolo de comunicação HTTP. Este modelo propõe um padrão bem aceito pelo mercado. Entender estes princípios é essencial para desenvolver aplicações escaláveis e manuteníveis. É a base da comunicação CLIENTE-SERVIDOR. Onde o cliente sempre começa a comunicação enviando uma requisição e o servidor envia uma resposta. Neste caso o cliente é uma aplicação (chamada aplicação cliente) que consome o recurso do servidor (aplicação servidor). É na resposta que recebemos o recurso. Este pode ser uma informação, um registro, um arquivo ou até mesmo uma página de site. Não trabalharemos com recebimento de páginas já que a aplicação cliente é o aplicativo que fará as requisições.

Estrutura de uma Requisição HTTP

Vejamos a composição de uma requisição HTTP:

* **Linha Inicial (Start Line):** Contém o método HTTP (GET, POST, PUT, DELETE, etc.), a versão do HTTP e o endpoint (URL) do recurso desejado.
* **Cabeçalho (Headers):** Contém metadados sobre a requisição, como o tipo de conteúdo do corpo, informações sobre o agente cliente e de autenticação.
* **Corpo (Body):** Contém dados a serem enviados para o servidor. É restrito a alguns métodos. Faz a aplicação retornar um erro caso haja uma tentativa de envio ou definição com um método inadequado.

Exemplos de Estruturas de Requisições

Os exemplos abaixo são representações de requisições ainda independente de linguagem de programação ou qualquer ferramenta de desenvolvimento. Vale ressaltar que antes de confeccionar uma aplicação cliente que vá consumir uma API, estuda-se a documentação dessa determinada API. Os exemplos são baseados numa API fictícia que hipoteticamente **instrui o desenvolvedor da aplicação**

cliente a construir as requisições exatamente com essas possibilidades. Ou seja, é requisitado somente o que o sistema pode responder. Mas essas possibilidades não são aleatórias. São padronizadas de acordo com a, mencionada anteriormente, arquitetura REST.

GET (Buscar)

É possível encontrar servidores de API que possuem apenas uma ação em sua interface. Necessitando apenas de um único end-point bastante simples. Na verdade, é muito mais comum em blogs ou sites Single-page. Mas voltemos ao fato que estamos nos conectando para consumir serviços de API.

Este poderia ser o caso citado acima:

```
GET /usuarios HTTP/1.1
Host: lib.com/api/
```

Listar todos os livros desta biblioteca:

```
GET /usuarios HTTP/1.1
Host: lib.com/api/books
```

GET com 1 parâmetro. um ISBN-*International Standard Book Number* fictício de um livro que retorna apenas um livro específico:

```
GET /usuarios HTTP/1.1
Host: lib.com/api/books/851598524568523
```

GET com 2 parâmetros. Neste caso volta a receber uma lista de livros. Livros lançados depois de 2024 e em poruguês.

```
GET /usuarios HTTP/1.1
Host: lib.com/api/books/^2024/Pt-br
```

GET com os parâmetros nomeados. Então chamamos de **query**. Produz resultado equivalente à requisição anterior. Não deve ser confundida com uma requisição com corpo. Não deve ser utilizada para alterar o estado do sistema se a API segue os padrões REST.

```
GET /usuarios HTTP/1.1
Host: lib.com/api/books?launch=^2024&lang=Pt-br
```

POST (Criar)

Requisição com caráter de alterar o sistema. Comumente usado para criar um novo registro no sistema. Os dados no corpo, podendo ser um json, xml ou até conter um binário contendo um audio, imagem, vídeo e outros, viajam de forma mais segura que os parâmetros enviados na uri. Em comunicação onde a segurança é um requisito prioritário esses dados ainda viajam criptografados. Veremos mais a frente sobre isso. Veja uma requisição POST com um .json no corpo. O endpoint é

'/usuário' e quando o servidor verifica o verbo, que é POST, direciona para a funcionalidade de criar um novo usuário:

```
POST /usuarios HTTP/1.1
Host: exemplo.com/api/
Content-Type: application/json

{
  "nome": "Novo Usuário",
  "email": "novo@exemplo.com"
}
```

PUT (Atualizar)

Frequentemente uma requisição PUT é usada para atualizar um registro. Perceba que o endpoint ainda é '/usuário', mas há dois detalhes novos. O verbo é PUT, então o servidor direciona os dados recebidos no corpo para a funcionalidade de atualizar um usuário de acordo com os dados. E, semelhante ao que vimos no GET, há um parâmetro '/34' que identifica qual registro será atualizado.

```
PUT /usuarios/34 HTTP/1.1
Host: exemplo.com/api/
Content-Type: application/xml

<usuario>
  <nome>Usuário Atualizado</nome>
  <email>atualizado@exemplo.com</email>
</usuario>
```

Você deve ter notado o corpo em **xml**. Poderia ser **json**. Foi só para mostrar como seria.

DELETE (deletar)

Sem corpo, o parâmetro na uri indica qual registro usuário será deletado. Mesmo sendo o mesmo end-point o verbo enviado ao servidor indicará a ação a ser executada.

```
DELETE /usuarios/34 HTTP/1.1
Host: exemplo.com/api/
```

Cabeçalhos Comuns

* `Content-Type`: Indica qual sintaxe compatível com o conteúdo, o 'data', do corpo da requisição (ex: `application/json`, `application/xml`).
* `Accept`: Da mesma forma é necessário indicar qual o formato/sintaxe de conteúdo o cliente aceita no corpo da resposta (ex: `application/json`, `*/*`).
* `Authorization`: Contém a chave criptografada que portege a API, fornecida pela API já com a autorização (ex: `Bearer <token>`). Veremos mais a frente detalhes sobre isso.

* `User-Agent`: Identifica o agente cliente (navegador, aplicativo, etc.).
* `Cookie`: informa detalhes sobre cookies armazenados.

Corpo da Requisição

O corpo da requisição contém os dados a serem enviados para o servidor. Os formatos largamente acitos pelo mercado são:

* **Formulário (application/x-www-form-urlencoded):** No cURL possui uma sintaxe igrual a uma query vista em GET: **nome=Anacleto&idade=41&cidade=Recife**
* **JSON (application/json):** Formato leve e amplamente utilizado para troca de dados.
* **XML (application/xml):** Formato mais complexo e estruturado, utilizado em aplicações legadas ou que exigem maior formalismo.
* **Texto Simples (text/plain):** Útil para disparar qualquer processamento que vai além dos verbos explanados acima ou serielizar dados.
* **Dados Binários (application/octet-stream):** Conhecidos como Blob (Binary Large Object). São arquivos como imagem, vídeos, áudio, etc.

Sobre os formatos **JSON** e **XML**:

JSON (JavaScrip Object Notation), como sugere o nome e, para quem já conhece a linguagem de programação javascript, é praticamente a sintaxe um objeto literal dentro de um texto (em termo abrasileirado, um objeto "stringuificado"). O XML, se assemelha bastante ao HTML. É verboso e customizável.

Status de Resposta

O servidor responde à requisição com um código de status indicando o andamento do processamento. Com isso é possível definir comportamentos para a aplicação, **considerando que esta suporta o processo assíncrono de requisição e resposta**, muito empregado para desenvolver telas de loadding ou carregamento de uma parte, componente, da tela:

* **2xx (Sucesso):**
    - 200 (OK): Requisição bem-sucedida.
    - 201 (Criado): Recurso criado com sucesso (usado em POST).
    - 204 (Nenhum Conteúdo): Requisição bem-sucedida, sem conteúdo a retornar.
* **3xx (Redirecionamento):**
    - 301 (Movido Permanentemente): Recurso movido para outro URL.
    - 302 (Encontrado): Recurso encontrado em outro URL (temporário).

* **4xx (Erro do Cliente):**
    - 400 (Solicitação Inválida): Requisição malformada.
    - 401 (Não Autorizado): Necessário autenticação.
    - 403 (Proibido): Acesso negado.
    - 404 (Não Encontrado): Recurso não encontrado.
* **5xx (Erro do Servidor):**
    - 500 (Erro Interno do Servidor): Erro genérico no servidor.
    - 503 (Serviço Indisponível): Servidor temporariamente indisponível.

HTTP é Stateless

Falamos muito sobre a estrutura de uma requisição. Mas agora vamos entender uma característica importante do comportamento. Dizer que HTTP é stateless significa que o servidor não mantém informações sobre as requisições anteriores que o cliente, em nosso caso é o aplicativo, fez. O app envia uma requisição para o servidor (ex: um app que consome um serviço de uma pizzaria que entrega em domicílio envia um comando correspondente à "busque uma lista de opções de pizzas para inserir no pedido") que a processa e responde (ex: um array de objetos correspondente às pizzas). Caso o mesmo dispositivo e app repita a requisição (ex: um botão de refresh), espera-se que o servidor processe como que a primeira não tivesse existido.

Isso vai resultar em:

* Classificação das requisições em **idempotente** e **não idempotente**
* Necessidade do uso de portar uma autorização (token) para rotas protegidas do sistema.

Método Idempotente

É aquela que se reenviada, mesmo com a configuração idêntica, não altera mais o estado do servidor. Com "estado" aqui se quer dizer "dados persistidos".

São estes métodos: GET, PUT, DELETE.

Cada vez que se solicita um DELETE do mesmo recurso, a partir da segunda tentativa haverá o lançamento de exceção no servidor por que não dá para excluir o mesmo registro duas ou mais vezes.

No caso de repetir uma mesma requisição GET, obviamente, só busca recurso para exibição. E por fim, PUT apenas alteraria o registro para o mesmo estado em que já se encontra.

POST não é idempotente. Caso uma mesma requisição para criação de um novo registro se repita, um banco de dados pode simplesmente atender a solicitação. Isso resulta em duplicidade de dados, embora não seriam a mesma chave primária dentro do banco. É claro que é comum os sistemas serem programados para evitar isso, dependendo das regras de negócio.

Existem outros métodos de requisição: HEAD, OPTIONS e TRADE - que são idempotentes. E PATCH que não é idempotente.

Programação Assíncrona

Antes de seguirmos com exemplos de requisições, é necessário explanar um pouco sobre o processo assíncrono do javascript já que ele marca grande presença quando se trata de processos que podem demorar indefinidamente ou retornar um erro.

Ao contrário da a programação síncrona, onde as tarefas são executadas em sequência, a programação assíncrona executa múltiplas tarefas de forma concorrente, sem que uma precise esperar o término da outra. Isso pode ser útil quando uma operação pode levar um tempo considerável para ser concluída, isso se aplica às conexões remotas, onde o tempo de resposta varia dependendo da latência da rede e do processamento interno do sistema.

Essa característica resolve problemas de travamento de tela que aguarda o segundo plano. Ele permite que o código continue seu fluxo programático até que a operação de segundo plano finalize. Então um evento ou call-back é acionado, fazendo que o sistema processe os resultados da operação.

Async/Await

Uma forma de escrever o código assíncrono em Javascript é fazendo o uso de operadores **async** e **await**.

O operador **async** é utilizado para declarar uma função que é assíncrona. Ela sempre retorna um objeto Promise, ainda que não seja usado de forma explicita a palavra-chave **return**. Em caso de retorno de valor, este será encapsulado em uma Promise resolvida. Caso seja lançada uma exceção, a Promise retornada será rejeitada.

```
async function obterDados() {
  // ...
}
```

O operador **await** Só pode ser usada dentro da função assíncrona. A função aguardará onde estiver o **await** a Promise ser resolvida ou rejeitada. No caso de resolvida, o valor resolvido é retornado. No caso de rejeitada, a exceção é lançada.

```
async function obterDados() {
      await fazAlgumaCoisa();
}
```

Tratamento de Erros

Try/Catch

Blocos try/catch lidando com erros em código assíncrono:

```
async function obterDados() {
  try {
    const resposta = await facaAlgo()
    return resposta;
  } catch (erro) {
    console.error('Erro ao obter restposta:', erro);
    // Lidar com o erro de forma apropriada
  }
}
```

Tratamento de Erros e Programação Assíncrona

A estrutura then/catch, facilmente, trabalha com a resposta de uma requisição assíncrona, ou o erro, caso ocorra.

Then/Catch

Antes do async/await, a forma mais comum de lidar com Promises era através dos métodos **then** e **catch**.

O método **then** é utilizado para registrar uma função que será executada quando a Promise for resolvida. Ele recebe dois argumentos: uma função para lidar com o caso de sucesso (resolução da Promise) e uma função para lidar com o caso de erro (rejeição da Promise).

```
fetch('https://exemplo.com/api/dados')
  .then(response => {
    if (!response.ok) {
      throw new Error('Erro na requisição'); // Lança um erro se a resposta não
for OK
    }
    return response.json();
  }, error => {
    console.error('Erro na requisição:', error); // Lida com erros na requisição
  })
  .then(data => {
    console.log('Dados recebidos:', data); // Exibe os dados no console
  });
```

O **error** faz a 'captura' do erro semelhante ao **catch** de **try-catch**. Mas podemos substituir por **catch**.

O método **`catch`** define uma função que será executada em caso de Promise ser rejeitada. É mais específica com os erros que o modelo anterior.

```
fetch('https://exemplo.com/api/dados')
  .then(response => {
    if (!response.ok) {
      throw new Error('Erro na requisição'); // Lança um erro se a resposta não
for OK
    }
    return response.json();
  })
  .catch(erro => {
    console.error('Erro na requisição:', error); // Lida com erros na requisição
  });
```

Seguiremos a partir daqui para os exemplos. Todos usando '**.then().catch()'** de forma simples. O suficiente para entender as requisições.

Requisições com Fetch

Vamos começar pela API Fetch, nativa do Javascript.

Sintaxe da requisição Fetch

```
let promise = fetch(url, [options])
```

GET

```
fetch('https://exemplo.com/api/dados', {
  method: 'GET',
  headers: {
    'X-Custom-Header': 'valor-do-meu-cabecalho',
  }
})
.then(response => {
  // Lida com a resposta do servidor
})
.catch(error => {
  // Lida com erros
});
```

Quando não é definido o **method**, automaticamente, fetch envia uma requisição GET. É possível, e comum, enviar a requisição sem o 2° parâmetro, o **options**.

```
fetch('https://exemplo.com/api/usuarios')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Vejamos mais exemplos GET. Desta vez seguindo o contexto da api de livraria das requisições no título **GET Buscar**:

Requisição comum em API com apenas uma funcionalidade:

```
fetch('https://lib.com/api')
  .then(response => response.json())
  .then(data => console.log(data));
```

Listar todos os livros desta biblioteca:

```
fetch('https://lib.com/api/books')
  .then(response => response.json())
  .then(data => console.log(data));
```

GET com 1 parâmetro. Um ISBN fictício de um livro que retorna apenas um livro específico:

```
fetch('https://lib.com/api/books/851598524568523')
  .then(response => response.json())
  .then(data => console.log(data));
```

GET com 2 parâmetros. Livros lançados depois de 2024 e em português:

```
fetch('https://lib.com/api/books/^2024/Pt-br')
  .then(response => response.json())
  .then(data => console.log(data));
```

GET com os parâmetros nomeados (query).

```
fetch('https://lib.com/api/books?launch=^2024&lang=Pt-br')
  .then(response => response.json())
  .then(data => console.log(data));
```

POST:

Agora considere uma API de cadastro de usuários para as próximas requisições. Veja que o parâmetro **options** é um objeto literal que configura principalmente **method**, **headers** e **body**.

```
fetch('https://exemplo.com/api/usuarios', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ nome: 'Novo Usuário' })
})
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

PUT:

Atente-se ao parâmetro na url que indica o ID do usuário a ser atualizado.

```
fetch('https://exemplo.com/api/usuarios/1', {
  method: 'PUT',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ nome: 'Usuário Atualizado' })
})
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Essa é uma requisição 'clássica' PUT. Mas veja que é possível configurar na url, parâmetros e queries do mesmo jeito que em GET, e enviar no corpo os mesmos dados que POST. Isso abre um leque de possibilidades na comunicação cliente-servidor.

DELETE:

Embora seja uma operação crítica, é bastante simples:

```
fetch('https://exemplo.com/api/usuarios/1', { method: 'DELETE' })
  .then(response => console.log(response.status))
  .catch(error => console.error(error));
```

Requisições com Axios

Axios faz que sua instância, que veremos mais a frente sobre isso, chame diretamente o verbo/método: `axios.get()`, `axios.delete()` ...

GET:

Sintaxe:

```
axios.get(url [, options])
```

GET para listar todos os usuários de uma tabela, por exemplo:

```
axios.get('https://exemplo.com/api/usuarios')
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

Abaixo, uma requisição GET busca o recurso '/usuario', mas diferente do exemplo anterior, esta requisição busca especificamente o usuário de ID n°5.

```
axios.get('https://exemplo.com/api/usuarios/5')
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

GET com query, caso o serviço consumido permita, faz uma filtragem. Observe a sintaxe que busca usuários ativos, no contrato de n° 56 e cadastro em 2009. É claro que, num sistema real, onde busca-se o dinamismo, esta string da url seria composta por algumas variáveis nos valores da query, e não uma string constante.

```
axios.get('https://exemplo.com/api/usuarios?status=ativo&contrato=56&cadastro=20
09')
  .then(response => {
    // Lógica para lidar com a resposta da API
  })
  .catch(error => {
    // Lógica para lidar com erros na requisição
  });
```

O exemplo anterior é equivalente ao próximo. Este, tem uma sintaxe mais clara com a segregação das partes que compõem o endereço. Inclusive a questão da dinâmica conta aqui também. Vejamos:

```
axios.get('https://exemplo.com/api/usuarios', {
  params: {
    status: 'ativo',
    contrato: 56,
    cadastro: 2009
```

```
  }
})
  .then(response => {
    // Lógica para lidar com a resposta da API
  })
  .catch(error => {
    // Lógica para lidar com erros na requisição
  });
```

Surtindo o mesmo efeito das duas anteriores, há uma possibilidade de que o servidor documente essa requisição assim:

```
axios.get('https://exemplo.com/api/usuarios/ativo/56/2009')
  .then(response => {
    // Lógica para lidar com a resposta da API
  })
  .catch(error => {
    // Lógica para lidar com erros na requisição
  });
```

POST:

Sintaxe:

```
axios.post(url[, data[, config]])
```

Já deixamos claro a diferença entre data e query numa requisição. Esta é enviada junto à uri, e aquela é o que compõe o corpo da requisição e que é restrito a alguns verbos.

```
axios.post('https://exemplo.com/api/usuarios',                //url
           { nome: 'Novo Usuário' },                          //data
           {
             headers: {                                        //config
                        'Content-Type': 'application/json'
             }
           })
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

PUT:

Sintaxe:

```
axios.put(url[, data[, config]])
```

Neste exemplo vamos alterar o nome do usuário identificado como 1.

```
axios.post('https://exemplo.com/api/usuarios/1',                //url
           { nome: 'Novo Usuário' },                             //data
           {
             headers: {                                          //config
                        'Content-Type': 'application/json'
             }
           })
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

As seguintes requisições abaixo são equivalentes. Atenção à query.

```
axios.post('https://exemplo.com/api/usuarios?ID=1',              //url
           { nome: 'Novo Usuário' },                             //data
           {
             headers: {                                          //config
                        'Content-Type': 'application/json'
             }
           })
  .then(response => console.log(response.data))
  .catch(error => console.error(error));

axios.post('https://exemplo.com/api/usuarios',                   //url
           { nome: 'Novo Usuário' },                             //data
           {
             headers: {                                          //options
                        'Content-Type': 'application/json'
             },
             params:  {
                         ID:1
                       }
           })
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

DELETE:

Sintaxe:

```
axios.delete(url[, config])

axios.delete('https://exemplo.com/api/usuarios/1')
  .then(response => console.log(response.status))
  .catch(error => console.error(error));
```

É claro que, seguindo a lógica dos verbos anteriores, embora sem o parâmetro **data**, mas com **config**:

```
axios.delete('https://exemplo.com/api/usuarios',                //url
           {
             headers: {                                          //options
                       'Content-Type': 'application/json'
             },
             params:  {
                        ID:1
                      }
           })
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

Exportando uma Instância Axios com o Mínimo de Configuração

Instanciar um objeto 'aleatoriamente' gera uma certa complexidade, em muitos sentidos, no sistema. É muito comum concentrar a instância do axios, ou qualquer API ou módulo, para que possa ser exportada e consumida em módulos cliente, de forma controlada. Gerando dessa forma, evitamos a repetição desnecessária do código. No exemplo a seguir, definimos configurações que serão usadas em todas as requisições:

/axiosServiceInstance.js

```
import axios from 'axios';

const api = axios.create({
  baseURL: 'https://api.exemplo.com/api/',          // URL base da sua API
  headers: { 'Content-Type': 'application/json' }   // Cabeçalho básico

export default api;
```

Os módulos clientes podem agora consumir **api** e dessa forma o código está mais manutenível.

Explicação:

1. **Importação:** Importamos a biblioteca `axios`.
2. **Criação da Instância:** Utilizamos `axios.create()` para criar uma nova instância do Axios.
3. **Configuração:**

- `baseURL`: Define a URL base para todas as requisições feitas com essa instância. Isso significa que você não precisará especificar a URL completa em cada requisição, apenas o caminho relativo.
- `headers`: Define os cabeçalhos padrão para todas as requisições. Neste exemplo, definimos o cabeçalho `Content-Type` como `application/json`, indicando que o corpo das requisições será formatado em JSON.

4. **Exportação:** Exportamos a instância `api`, que poderá ser utilizada em outros arquivos da sua aplicação.

Como usar:

Em outros arquivos, você pode importar a instância `axiosServiceInstance` e utilizá-la para fazer requisições. Veja que não será mais necessário definir a parte do endereço que já foi definida no código anterior, o `https://api.exemplo.com/api/` e todas as configurações foram predeterminadas eliminando a repetição de código:

```
import api from './axiosServiceInstance'; // Importa a instância configurada

api.get('/usuarios')   //será considerado ´https://api.exemplo.com/api/usuarios´ na execução.
  .then(response => console.log(response.data))
  .catch(error => console.error(error));

api.post('/produtos', { nome: 'Novo Produto', preco: 10 })
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

O Parâmetro config da Requisição Axios

O parâmetro `config` é um objeto literal que permite personalizar a requisição, garante o controle sobre o comportamento desta. Tem uma gama de atributos customizáveis que podem ser úteis para adequar a requisição as necessidades do desenvolvedor.

Atributos do `config`:

* **`url`:** URL da requisição. Se `baseURL` for definido na instância Axios, este será o que chamamos de rota ou end-point.
* **`method`:** Método HTTP da requisição. Caso não seja especificado, o padrão é GET.

* **`baseURL`:** URL base para a requisição. Se definido, será usado em vez do `baseURL` da instância Axios.
* **`headers`:** Cabeçalhos da requisição. Para adicionar ou sobrescrever os definidos na instância Axios.
* **`params`:** String de consulta (query string).
* **`data`:** Corpo da requisição para métodos POST, PUT, PATCH. Nos formatos já explanados na Introdução à Arquitetura REST.
* **`timeout`:** Tempo limite para retornar em milissegundos. Caso não haja retorno dentro do tempo indicado, será lançado um erro.
* **`auth`:** Credenciais para autenticação. Objeto com `username` e `password` ou um objeto com `token`.
* **`responseType`:** Tipo de resposta esperado. Pode ser `json`, `xml`, `text`, `blob` e outros.
* **`transformRequest`:** Função que permite modificar os dados da requisição antes de serem enviados para o servidor.
* **`transformResponse`:** Função que permite modificar os dados da resposta antes de serem entregues ao código que fez a requisição.
* **`onUploadProgress`:** Função que permite acompanhar e tratar o progresso do upload de dados.
* **`onDownloadProgress`:** Função que permite acompanhar e tratar o progresso do download de dados.
* **`signal`:** Sinal e uma instância de AbortController pode ser usada para cancelar a solicitação.
* **`proxy`:** Define o nome do host, a porta e o protocolo do servidor proxy. Define credenciais e outras funcionalidades.

Há algumas outras opções além dessas elencadas que estão na documentação do Axios, ou sofreram depreciação ou são exclusivas para usar no node.js. como **maxContentLength**, **maxBodyLength** e **cancelToken**.

Exemplo:

```
axios.get('/usuarios', {
  baseURL: 'https://api.exemplo.com', // Sobrescreve o baseURL da instância
  params: { id: 123 }, // Adiciona parâmetros à URL
  timeout: 5000, // Define o tempo limite para 5 segundos
  headers: { 'X-Custom-Header': 'valor' } // Adiciona um cabeçalho personalizado
})
```

```
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

O Objeto Response do Axios

O objeto `response` é o resultado das requisições Axios e contém informações sobre a resposta do servidor. Ele possui os seguintes atributos:

* **`data`:** Dados da resposta. O formato depende do cabeçalho e de como o servidor trata tais informaçoes. Leia a documentação da API.
* **`status`:** Código de status HTTP da resposta (ex: 200, 404, 500) já citado na introdução deste texto.
* **`statusText`:** Texto do status HTTP da resposta (ex: "OK", "Not Found", "Internal Server Error").
* **`headers`:** Cabeçalhos da resposta. Assim como na requisição, a resposta contém metadados para auxiliar o tratamento da mesma.
* **`config`:** Objeto de configuração da requisição que gerou a resposta. Importante para depuração.
* **`request`:** Objeto da requisição original. Ajuda a testar a aplicação cliente.

Exemplo:

```
axios.get('/usuarios')
  .then(response => {
    console.log(response.data); // Dados da resposta
    console.log(response.status); // Código de status
    console.log(response.statusText); // Código de status
    console.log(response.headers); // Cabeçalhos da resposta
    console.log(response.config); // Configuração da requisição
  })
  .catch(error => console.error(error));
```

Ao compreender o parâmetro `config` e o objeto `response`, é possível controlar as requisições Axios, permitindo acessar e configurar cada detalhe e as informações relevantes da resposta do servidor tornando a criação de aplicações React Native robustas e capazes de lidar com variados cenários na comunicação remota.

Repito o concelho. Consulte a documentação não apenas do Axios ou qualquer biblioteca que faz parte do desenvolvimento para obter informações exatas sobre cada funcionalidade. Cursos, fóruns

e outras fontes mais humanizada de informação podem conter equívocos, imprecisões ou um contexto ocultado.

Exemplo com FlatList em Paradigma Funcional

Vamos pôr um pouco de front-end aqui. Considerando que o leitor conheça a utilidade dos **hooks** e **Components**. Não terá uma profunda explicação, pois foge do tema. Vejamos:

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

const Inventario = () => {
  const [items, setItems] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchInventario = async () => {
      try {
        const response = await axios.get('https://api.exemplo.com/inventario');
        setItems(response.data);
      } catch (err) {
        setError(err);
        console.error("Erro ao carregar inventário:", err); // Log do erro para
depuração
      } finally {
        setLoading(false);
      }
    };

    fetchInventario();
  }, []);

  const renderItem = ({ item }) => (
    <View style={styles.item}>
      <Text>{item.nome}</Text>
      <Text>{item.quantidade}</Text>
    </View>
  );
```

```
  if (loading) {
    return (
      <View style={styles.container}>
        <ActivityIndicator size="large" color="#0000ff" />
        <Text>Carregando inventário...</Text>
      </View>
    );
  }

  if (error) {
    return (
      <View style={styles.container}>
        <Text style={styles.errorText}>Erro: {error.message}</Text>
      </View>
    );
  }

  return (
    <View style={styles.container}>
      <FlatList
        data={items}
        renderItem={renderItem}
        keyExtractor={item => item.id.toString()} // Certifique-se de ter um ID
único
      />
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 16,
  },
  item: {
    flexDirection: 'row',
    justifyContent: 'space-between',
    padding: 10,
    borderBottomWidth: 1,
    borderBottomColor: '#ccc',
```

```
  },
    errorText: {
        color: 'red',
        fontSize: 16,
    },
});

export default Inventario;
```

Explicação:

1. **Importações:** Importamos os componentes necessários do React Native e o Axios.
2. **Estados:**
    - `items`: Array de objetos item.
    - `loading`: Flag que indica se a requisição está em andamento.
    - `error`: Errors e exceptions.
3. **`useEffect`:**
    - Faz a requisição GET para a API.
    - Atualiza o estado `items` com a o array recebido recebidos.
    - Lida com erros, caso haja, manipulando o estado `error`.
    - Define `loading` como `false` após a requisição (com sucesso ou falha) interrompendo a tela de loading.
4. **`renderItem`:** Renderiza cada item na `FlatList`.
5. **Renderização Condicional:** Entramos aqui na parte com os `if` :
    - Exibe uma imagem de carregamento (`ActivityIndicator`) enquanto `loading` for `true`.
    - Exibe uma mensagem de erro se `error` não for `null`.
    - Exibe a `FlatList` que é o resultado esperado.
6. **Estilos:** Define os estilos para os componentes.

Observações:

* É obrigatório um ID único para cada item na `FlatList`. Do contrário a tela não funciona.
* Perceba o paradigma funcional com hooks (`useState` e `useEffect`).

Exemplo de Requisição DELETE com Tratamento de Erros

```
import axios from 'axios';

const deletarUsuario = async (id) => {
  try {
    const response = await axios.delete('https://api.exemplo.com/usuarios/'+id);
    console.log("Usuário deletado:", response.status);
    // Lógica adicional após a exclusão (ex: atualizar a lista de usuários)
  } catch (error) {
    console.error("Erro ao deletar usuário:", error);

    if (error.response) {
      // O servidor respondeu com um status de erro
      console.error("Dados do erro:", error.response.data);
      console.error("Status do erro:", error.response.status);
      console.error("Cabeçalhos do erro:", error.response.headers);
    } else if (error.request) {
      // A requisição foi feita, mas não houve resposta
      console.error("Requisição não respondida:", error.request);
    } else {
      // Algo aconteceu ao configurar a requisição e acionou o erro
      console.error("Erro de configuração da requisição:", error.message);
    }
  }
};

// Exemplo de uso:
deletarUsuario(123);
```

Explicação:

1. **`deletarUsuario`:** Função assíncrona que faz a requisição DELETE.
2. **`try...catch`:**
    - O bloco `try` contém a requisição DELETE com Axios.
    - O bloco `catch` lida com qualquer erro que ocorra durante a requisição.
3. **Tratamento de Erros:** Do mais específico ao mais genérico, como recomendado.

- `error.response`: Verifica se o servidor respondeu com um status de erro (ex: 404, 500). Se sim, exibe os dados, status e cabeçalhos do erro.
- `error.request`: Verifica se a requisição foi feita, mas não houve resposta do servidor.
- `error.message`: Se o erro não for relacionado à resposta ou requisição, exibe a mensagem de erro.

Observação:

* Acima, o código mostra um tratamento de erros mais robusto que os exemplos anteriores. Ele diferencia os tipos de erros do mais específico ao mais genérico e exibe informações para depuração.

Enviando uma Chave de Autorização para uma API

Em cenários de desenvolvimento onde a segurança do app é primordial, pode ser que a API exija uma chave criptografada para autenticação fornecida pela mesma. A criptografia garante que apenas o servidor, que possui a chave de descriptografia, acesse essas informações. Não estamos falando de autenticação de usuários neste caso. Algumas APIs solicitam a inscrição do desenvolvedor onde este recebe, por email, uma chave criptografada que o identifica. Para requisitar serviços, basta inserir esta chave na requisição. Isso identifica o **desenvolvedor**, não o **usuário**. Como a chave será enviada em todas as requisições, isso a torna uma configuração que deve ser definida na instância única de axios.

Exemplo com Axios:

/axiosServiceInstance.js

```
import axios from 'axios';

const api = axios.create({
  baseURL: 'https://api.exemplo.com/api/',          // URL base da sua API
  headers: {
            'Content-Type': 'application/json',
            'Authorizationh': 'a1b2c3d4e5f6g7h8'
           }

export default api;
```

Enviando um Bearer Token para uma Rota Protegida

Vimos anteriormente que HTTP é stateless. Em vista disso, alguns servidores fazem uso de seção. Embora seja bastante seguro, o uso de seção traz algumas desvantagens. O servidor precisa armazenar as informações que podem gerar problemas no desempenho em aplicações com muitos usuários. Outra solução também segura e com uso em crescimento atualmente é o Bearer Token (pode ser interpretado como "*portador de acesso*"). Durante o que seria uma seção, na verdade a autorização é enviada no cabeçalho de cada requisição. O servidor verifica se é válido e reponde com o recurso solicitado. Em analogia, seria como andar dentro de um prédio com um crachá. Os seguranças analisam o crachá e permitem a passagem, se estiver de acordo.

Vejamos o fluxo de um login tradicional de um sistema que usa Bearer Token:

* O usuário envia as credencias após cadastrado para uma rota não protegida;
* Servidor valida, autentica, e define a autorização - vale a atenção ao fato de que esses passos representam procedimentos diferentes;
* Servidor persiste uma chave criptografada segundo o algoritmo escolhido contendo as informações definidas sobre o titular;
* Servidor envia a chave que deve receber do usuário nas próximas requisições contendo a autorização citada;
* App agora possui o token com a autorização de acesso adequada ao usuário.

Nisso constitui o login. Agora vamos ao acesso:

* Com a chave em posse, o App a envia em cada requisição para rotas protegidas;
* O servidor analisa a chave e sabe, de acordo com ela, quem é o titular;
* O servidor processa e retorna ao cliente com o recurso solicitado.

Exemplo com Axios:

```
import axios from 'axios';

const acessarRotaProtegida = async (token) => {
  try {

    const response = await axios.get('https://api.exemplo.com/rota_protegida', {
      headers: {
        'Authorization': `Bearer ${token}` // Envia o token no cabeçalho
      }
    });

    console.log("Resposta da rota protegida:", response.data);
  } catch (error) {
```

```
    console.error("Erro ao acessar rota protegida:", error);
  }
};

// Exemplo de uso:
acessarRotaProtegida(token); //token fornecido após o login.
```

Explicação:

1. **`acessarRotaProtegida`:** Função assíncrona que acessa a rota protegida.
2. **Requisição GET:** Enviamos o token no cabeçalho `Authorization` da requisição, utilizando a sintaxe `Bearer ${token}`.
3. **Tratamento de Erros:** Lida com erros na requisição.

Observações:

* A API deve validar o token no servidor para autorizar o acesso à rota protegida.

Exemplo de Loading.

O famoso sinal do símbolo que indica que está carregando pode ser aplicado de um módulo nativo do React Native. Trata-se do componente **ActivityIndicator** (*Indicador de Atividade*). É possível customizá-lo:

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

const App = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await
axios.get('https://jsonplaceholder.typicode.com/todos/1');
        setData(response.data);
      } catch (err) {
        setError(err);
```

```
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, []);

  if (loading) {
    return (
      <View style={styles.container}>
        <ActivityIndicator size="large" color="#0000ff" />
        <Text>Carregando...</Text>
      </View>
    );
  }

  if (error) {
    return (
      <View style={styles.container}>
        <Text style={styles.errorText}>Erro: {error.message}</Text>
      </View>
    );
  }

  return (
    <View style={styles.container}>
      <Text>Título: {data.title}</Text>
      <Text>Concluído: {data.completed ? 'Sim' : 'Não'}</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  errorText: {
    color: 'red',
    fontSize:
```

Introdução à Arquitetura Offline First

Um usuário ao usar um aplicativo pode se encontrar em algumas dessas situações:

* Largura de banda de Internet limitada.
* Interrupções de conexão temporárias, como ao entrar em um elevador ou túnel.
* Acesso a dados ocasional, por exemplo, em tablets que operam somente em rede Wi-Fi.

Algumas aplicativos, a depender do problema que eles solucionam, não precisam estar conectados de forma ininterruptas. Neste caso é possível aplicar a arquitetura **Offline First** (*Primeiro Offline*). Em vez de depender continuamente de uma conexão online, o aplicativo entra num ciclo em que persiste os dados e continua processando de acordo com eles, permitindo a continuidade da interação do usuário com o aplicativo sem que haja interrupções. É claro que não estamos falando de uma transmissão, reunião, ou aplicativos de banco e outros.

Fluxo de Funcionamento:

Considerando que um aplicativo exibe uma lista de registros:

1. **Inicialização:** O aplicativo verifica, inicialmente, se há dados armazenados localmente. Se houver, o aplicativo mostra os dados para o usuário, dando-lhe uma experiência imediata e sem interrupções.
2. **Sincronização:** Em seguida, o app verifica a conexão com internet e tenta sincronizar os dados locais com o servidor remoto. "Sincronizar" significa dizer que o aplicativo envia as alterações locais para o servidor (eventos) e recebe os dados já atualizados do servidor.
3. **Eventos em Offline:** Quando offline, o app armazena os eventos no banco de dados e permanece verificando a conexão com a rede. E o fluxo reinicia.

O aplicativo funciona sem interrupção, ainda que não esteja conectado. O carregamento de dados num banco de dados embarcado é muito mais rápido do que no remoto, a aplicação se torna mais ágil e responsiva. O app mais confiável em áreas com cobertura instável ou inexistente.

Complexidades da Arquitetura Offline First:

Já se sabe que Offline First (OF) não deve ser aplicada em todas as soluções. Isso deve ser definido antes da faze de desenvolvimento. Para que não haja incoerências especialmente em app multiusuários. Então deve-se levar em conta a complexidade da persistência e sincronização de dados, o tratamento de conflitos para manter a consistência dos dados de acordo com seus titulares. E por fim a concorrência no sistema.

Arquitetura Offline First aplicada em Redux

A missão aqui não é detalhar sobre Redux, pois isso fugiria do escopo do assunto. Mas com um breve resumo poderemos entender melhor a camada onde OF será aplicada.

As 4 camadas da Arquitetura Redux

1. **Store (Armazém):** É um objeto que contém o estado global da aplicação. Possui a instância da DAO da entidade envolvida na requisição.
2. **Actions (Ações):** Encapsula as chamadas que vão disparar as funcionalidades em Store.
3. **Reducers (Redutores):** São funções que recebem o estado atual e uma Action, e retornam um novo estado. Eles são os responsáveis por "como" o estado deve ser alterado, baseados na Action recebida.
4. **View:** São as telas onde o usuário dispara eventos inscritos em Actions.

Fluxo de Dados Unidirecional

A arquitetura Redux garante um fluxo de dados unidirecional, o que significa que as informações sempre fluem em uma única direção:

1. O usuário interage com um Componente (na View).
2. O Componente despacha uma Action para o Store.
3. O Store envia a Action para todos os Reducers.
4. Os Reducers processam a Action e retornam um novo estado.
5. O Store é atualizado com o novo estado.
6. Os Componentes são notificados da mudança e se atualizam.

Implementação

Como já entendido, implementar OF significa que deverá ser implementada também toda a arquitetura para persistência de dados em banco de dados embarcado. Para isso é necessário implementar uma ORM (Mapeamento Objeto Relacional), responsável por representar os dados a serem manipulados, e a DAO (*Data acces Object*) da entidade em questão. É recomendável definir um gerenciador de estado centralizado em Store. Por esses motivos, OF é facilmente encaixada na arquitetura orientada a eventos Redux.

Vamos ao exemlo clássico de cadastro de usuários. O usuário preenche um formulário com seus dados e, ao clicar em "Salvar", esses dados precisam ser enviados para o servidor através de uma requisição POST. Considerando que na engenharia de requisitos foi determinado que é possível tolerar

um certo atraso no cadastro e que isso não trará danos aos usuários. Então, em acordo, foi determinado que seria aplicada a arquitetura OF. Boltando a olhar o Redux, como visto antes, OF começa a atuar na camada **Actions**.

No exemplo abaixo, uma ação é definida. Mas devemos nos atentar aos metadados **commit** e **rollback**, que poem em ação uma funcionalidade para cada estado da conexão (disponível e não disponível) com a API.

Código:

UserActions.js

```
export default const registerUser = (userData) => ({
  type: 'REGISTER_USER',
  payload: { userData },
  meta: {
    offline: {
      effect: {                //dispara a tentativa de envio da requisição.
        url: '/api/users',
        method: 'POST',
        json: userData,
      },
      commit: { type: 'REGISTER_USER_COMMIT', meta: {userData}},    //Sucesso.
Agora persiste no DB local.
      rollback: { type: 'REGISTER_USER_ROLLBACK', meta: {userData}},    //Falha.
Vai ficar no DB para uma sincronização futura.
    },
  },
});
```

1. **`registerUser(userData)`:**
    - Recebe os dados do usuário (`userData`) como argumento.
    - Define o tipo da action como `REGISTER_USER`.
    - Inclui os dados do usuário no payload da action.
    - Utiliza a meta propriedade `offline` do Redux Offline para configurar a requisição POST:
        - `effect`: Define a URL (`/api/users`), o método HTTP (`POST`) e os dados a serem enviados (`userData`).
        - `commit`: Define a action a ser despachada após a requisição POST bem-sucedida (`REGISTER_USER_COMMIT`).

- `rollback`: Define a action a ser despachada caso a requisição POST falhe (`REGISTER_USER_ROLLBACK`).

*reducer.js_*

```
const initialState = {
  users: [],
  loading: false,
  error: null,
};

const usersReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'REGISTER_USER':
      return { ...state, loading: true, error: null };
    case 'REGISTER_USER_COMMIT':
      return { ...state, loading: false }; // Limpa o loading após o sucesso
    case 'REGISTER_USER_ROLLBACK':
      return { ...state, loading: false, error: action.payload }; // Define o
erro
    default:
      return state;
  }
};

export default usersReducer;
```

2. **usersReducer:**
    - Caso a action seja `REGISTER_USER`:
        - **loading** é uma flag que vai indicar que a requisição está em andamento.
        - **error: null** Limpa qualquer erro anterior.
    - Caso a action seja `REGISTER_USER_COMMIT`:
        - Define `loading` como `false` para indicar que a requisição foi concluída com sucesso.
        - Também deve ser usada para realizar outras ações adicionais após o sucesso da requisição. Especialmente chamar a exibição da tela ou parte dela com dados atualizados.
    - Caso a action seja `REGISTER_USER_ROLLBACK`:

- **loading** é uma flag que vai indicar que a requisição falhou.
- Define **error** com o payload da action, que deve possuir as informações sobre o erro.
- Além disso, pode chamar funcionalidades com exibir uma mensagem de erro para o usuário ou permitir que ele tente novamente.

Verificando o Estado da Rede com NetInfo

Para finalizar, uma bilbioteca também bastante usada para verificar o estado da conexão com a rede é a **NetInfo**. Existem aplicações que podem já buscando essas informações antes mesmo que haja a tentativa de consumo do serviço. É o caso dos apps totalmente ou parcialmente dependentes de conexão. Neste último, é possível incluir arquitetura OF. NetInfo é prático e possui muitas maneiras de ser usado. Vamos ver o fragmento do código onde se faz o padrão **verificação única**:

Tela.js

```
import NetInfo from "@react-native-community/netinfo";

NetInfo.fetch().then(state => {
  console.log("Connection type", state.type);
  console.log("Is connected?", state.isConnected);
});
```

Explicação:

3. **Importação:** Importamos `NetInfo`.
4. **`NetInfo.fetch()`:** Retorna uma Promise e resolve com um objeto literal com informações sobre o estado da conexção coma a rede.
5. **`state.type`:** Tipo de conexão: `wifi`, `cellular`, `unknown`.
6. **`state.isConnected`:** Mostra se está conectado: true ou false.

Conclusão

Em suma, mostramos como funcionam as principais ferramentas de conexão no React Native. O Fetch é uma solução nativa e de baixo custo computacional para requisições HTTP, o Axios, por sua vez, destaca-se pela sua robustez em interceptar requisições, gerenciar erros de forma elegante.

Também vimos a arquitetura Offline First, que é um paradigma fundamental para o desenvolvimento onde não pode se pressupor a conectividade constante. Implementá-lo quando cabível não apenas melhora significativamente a experiência do usuário, como também otimiza a aplicação no dispositivo. Esta abordagem em questão transcende o aspecto meramente técnico, mas alcança uma mudança filosófica de projetos de aplicativos que prioriza a autonomia independentemente do status da rede.