Mobile

Importância dos smartphones

No ano de 2007, a Apple lançou o primeiro iPhone, e com ele, uma nova era de dispositivos móveis.

Desde então, os smartphones se tornaram uma parte essencial da vida cotidiana, e a maioria das pessoas não consegue imaginar a vida sem eles.

pedro.mateus@unicesumar.edu.br

Grande parte da população mundial possui um smartphone, e a tendência é que esse número cresça cada vez mais.

Existem diversos estudos estimando que mais da metade da população mundial já possui e usa um smartphone.

Isso significa que a maioria das pessoas tem um dispositivo móvel, e que a maioria das pessoas passa a maior parte do tempo usando aplicativos.

Reflexão

- *Você já parou para pensar na importância dos smartphones e aplicativos?*
- *Quais aplicativos você conhece que mudaram a maneira que as pessoas vivem?*
- *Quais aplicativos você usa no seu dia a dia?*

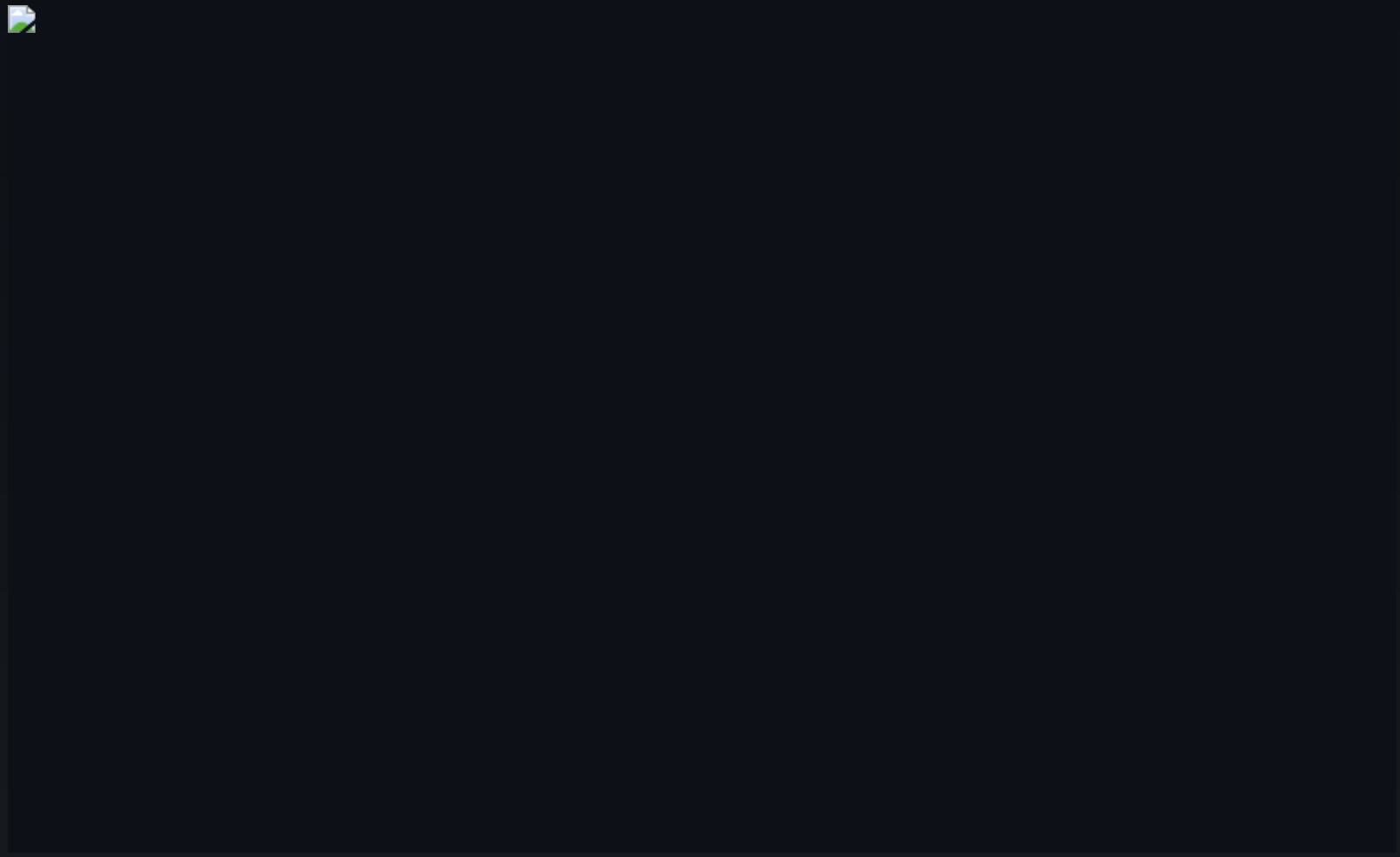
Apps nativos e híbridos

> Apps nativos

São aplicativos desenvolvidos para uma plataforma específica, como Android ou iOS, escritos em linguagens de programação específicas para cada plataforma.

> Apps híbridos

São aplicativos desenvolvidos com tecnologias web/não nativas, como HTML, CSS e JavaScript, e são encapsulados em um aplicativo nativo, como Cordova, PhoneGap.



Diferença entre web app, app nativo e híbrido

React Native

> Por que React Native ?

- É uma framework open source, criada pelo Facebook (Meta) que segue os princípios do React;
- Com um único código, é possível criar aplicações Android e iOS, salvando tempo e esforço que dois projetos separados podem causar, o mesmo vale para manutenção do código;

- Seu desempenho é bem semelhante ao desempenho de aplicações nativas, pois são transpiladas em componentes nativos de cada plataforma (iOS / Android);
- Comunidade grande e ativa, o ecossistema é enorme e possui diversas bibliotecas e soluções já prontas para muitos casos de uso, integração de serviços, etc;
- Quantidade de documentação, tutorial e fóruns disponíveis;
- O react native consegue integrar com bases de códigos que já utilizam código nativo, sendo possível criar funcionalidades usando a framework em projetos nativos já existentes.

Pré requisitos

1. Conhecimento básico de ***JavaScript/TypeScript***, pois é a linguagem utilizada.

2. ***NodeJS*** e ***npm***, pois Node é onde o código JS é interpretado e executado, fora dos navegadores e ***npm*** que é o gerenciador de pacotes, responsável para instalar e gerenciar bibliotecas terceiras.

3. Expo / React Native CLI.

4. Editor de código (VSCode).

5. Android Studio para Android e XCode para iOS.

2. Começando com React Native

> Básico de JavaScript

Para trabalhar com react native é importante ter um bom entendimento de ***JavaScript***, pois é a linguagem utilizada para desenvolver as aplicações.

- ***Variáveis*** e ***tipos de dados***;
- ***Funções***;
- ***Objetos***;
- ***Arrays***;
- ***Controles de fluxo***;

> ES6

O ***ES6*** (ECMAScript 6) é uma versão do JavaScript que foi lançada em 2015, e trouxe várias novas funcionalidades para a linguagem, essas são bem úteis para o desenvolvimento, como:

- ***Arrow functions***;
- ***Template strings***;
- ***Desestruturação***;
- ***Operador spread***;

Exemplos JavaScript

> Componentes nativos

O react native aproveita ***componentes nativos***, blocos para construção de telas, fornecidos pelas plataformas móveis nativas (Android e iOS). Esses componentes nativos incluem botões, textos, listas, etc.

Exemplo de alguns componentes

> Virtual DOM e reconciliação

O react native utiliza o conceito de ***Virtual DOM***, o mesmo conceito utilizado no ReactJS, sendo uma representação do ***Document Object Model*** em memória, mais rápido de manipular e renderizar. Toda vez que ocorre alguma atualização no estado da aplicação, o react native faz uma comparação entre o ***Virtual DOM*** e o ***DOM*** real, e atualiza apenas o que foi alterado, esse processo é chamado ***reconciliação***.

> Arquitetura baseada em componentes

O react native é baseado em **componentes**, onde as telas (UI) são construídas a partir de componentes menores, que podem ser reutilizados em outras telas, ou até mesmo em outras aplicações.

Os componentes são unidades independentes que encapsulam lógica e apresentação (UI), promovendo o reuso de código, modularidade e manutenibilidade.

O React native possui duas categorias de componentes, os componentes ***funcionais*** e componentes de ***classe***. Atualmente é recomendado pelos próprios criadores que se utilize **componentes funcionais**.

Utilizando um componente

> Controle de estado

Conceito essencial para o desenvolvimento, o react native possui várias opções para controle de estado, sendo os dois principais conceitos: ***state*** e as ***props***.

- ***State*** é um objeto que contém dados que podem ser alterados durante a vida útil do componente. Quando o estado é alterado, o componente é renderizado novamente.

- ***Props*** são propriedades passadas dos componentes pai para componentes filho, e não podem ser alteradas durante a vida útil do componente.

Exemplo de controle de estado com props

Configurando o ambiente de desenvolvimento

> NodeJS

Para instalar o node, bata baixar a versão LTS, disponível no Site oficial. Após a instalação, é possível verificar a versão do node e do npm, através do comando:

```
node -v
```

Com a instalação do node, o ***npm*** também é instalado, para verificar a versão do npm, basta rodar o comando:

```
npm -v
```

> VSCode

O VSCode é um editor de código fonte, gratuito e open-source, desenvolvido pela Microsoft. Para instalar o VSCode, basta baixar a versão compatível com o seu sistema operacional, disponível no Site oficial.

> Expo

O Expo é uma ferramenta que facilita o desenvolvimento de aplicações React Native, pois possui uma série de ferramentas e bibliotecas já integradas, bem como a possibilidade de rodar o projeto em um emulador ou no ***próprio celular***. React Native Expo Go Quick Start

`(Opcional) - Baixar o aplicativo Expo Go no celular, para rodar o projeto no próprio celular.`

Criando um projeto React Native com Expo

Agora que temos o editor de texto e todas as ferramentas necessárias instaladas, podemos criar o nosso primeiro projeto react native. Para isso, basta abrir o terminal, escolher um diretório e rodar o comando:

```
npx create-expo-app <NomeDoProjeto>
```

Após a instalação, basta entrar no diretório do projeto e rodar o comando:

```
cd <NomeDoProjeto>;
npx expo start
```

Após startar o projeto, aparecerá um QRCode no terminal, basta escanear o QRCode com o aplicativo Expo Go que será possível visualizar o projeto no próprio celular.

Também é possível rodar o projeto em um emulador, para isso, basta ter o Android Studio com um Android Virtual Device instalado e rodando, e pressionar `a` no terminal, que o projeto será aberto no emulador.

`Todas as instruções para rodar o projeto no emulador ou no próprio celular, estarão disponíveis no terminal após rodar o comando npx expo start.`

Caso deseje rodar o projeto na web, também é possível, porém é necessário instalar alguns pacotes adicionais, com os seguintes comandos:

```
npx expo install react-dom react-native-web @expo/webpack-config
```

O comando irá instalar as dependências react-dom, react-native-web e @expo/webpack-config.

Após a instalação, basta rodar novamente o comando:

```
npx expo start
```

depois, pressionar `w` no terminal, que o projeto será aberto no navegador.

Estrutura do projeto e pastas

> Pastas e arquivos gerados automaticamente pelo `npx create-expo-app`

```
├── App.js # Arquivo de entrada. Primeiro componente renderizado
├── app.json # Configurações do projeto expo
├── assets # Imagens, fontes e etc
│   └── [...]
├── babel.config.js # Configurações do bundler para React Native
├── node_modules # Onde dependências do projeto são instaladas
│   └── [...]
├── package.json # Dependências, scripts e detalhe do projeto
├── package-lock.json # Versões das dependências
├── README.md # Documentação do projeto
```

node_modules é a pasta onde ficam as dependências do projeto e as bibliotecas terceiras para o projeto, todas são geradas automaticamente e gerenciadas pelo npm.

Pré-build, Build, APK e Publicação

> Pre-build

Antes de realizar o build (gerar arquivos para publicar nas lojas), é necessário realizar o ***pre-build***, que é o processo que gera o código nativo (iOS e Android). Mais informações podem ser encontradas na Documentação oficial. Comando para realizar o pre-build:

```
npx expo prebuild
```

Após executado, serão geradas duas pastas, uma para iOS e outra para Android, com os arquivos nativos. que podem ser executados no XCode e no Android Studio. E serão utilizados para o build. Esse processo é interessante principalmente em casos que se deseja alterar arquivos nativos antes do build.

> Build

O Expo recomenda utilizar o EAS para realizar o build, pois já gera o binário pronto para publicação nas lojas (Google Play Store ou Apple App Store), o único detalhe é que ele exige que você tenha uma conta no Expo, e que o projeto esteja publicado no Expo.

Para instalar o EAS CLI, basta rodar o comando e em seguida autenticar-se na conta do Expo:

```
npm install -g eas-cli;
eas login
```

Agora podemos criar um arquivo de configuração para o build, onde podemos definir para qual plataforma ele será buildado (Android, iOS ou ambas), para isso precisamos rodar o comando:

```
eas build:configure
```

Após escolher entre Android, iOS ou ambos, será gerado um arquivo de configuração, que pode ser encontrado em `./eas.json`, e pode ser alterado manualmente, caso necessário. Para mais informações sobre o arquivo de configuração, basta acessar a Configuração do EAS Build com o eas.json.

Agora podemos rodar o comando para realizar o build:

```
eas build
```

Esse comando irá perguntar novamente qual plataforma desejamos buildar, caso escolha **iOS** ou **ambas**, será necessário autenticar-se no Apple Developer. Para evitar isso podemos selecionar para buildar somente Android.

Essa etapa de build é feito na **Nuvem do Expo**, por isso é necessário esperar alguns minutos até que o build seja finalizado, pois, existe uma fila no plano gratuito. O comando irá mostrar no terminal um link com os detalhes do build, algo parecido com:

```
https://expo.dev/accounts/{conta}/projects/{projeto}/builds/{buildId}
```

Para fazer a publicação na loja, basta seguir o passo a passo disponível em Publicar o build

Todo o processo de build pode ser encontrado na documentação oficial Processo de build EAS

> Build de produção localmente

Também é possível gerar o build localmente, porém é mais trabalhoso e requer um macOS caso desejamos buildar para iOS. O processo pode ser encontrado em Build de produção local.

> APK

Também é possível gerar APKs para Android, porém é necessário algumas configurações extras no arquivo `eas.json`. Para mais informações, basta acessar Configurações para gerar APKs.

Tabela com comandos utilizados

Comando	Ação
`npx create-expo-app <NomeDoProjeto>`	Inicia um projeto expo, `<NomeDoProjeto>` deve ser substituído
`npx expo start`	Inicia o projeto expo, mostra o QRCode e demais funções
`npx expo install`	Instala as dependências do projeto
`npx expo prebuild`	Gera o código nativo iOS e Androxid

Comando	Ação
`eas login`	Comando para autenticar na conta Expo
`eas whoami`	Verifica se está autenticado
`eas build`	Inicia processo de build para iOS, Android ou ambos
`eas build:list`	Lista de builds da sua conta

Principais hooks do React Native

O React possui uma série de ***hooks*** que podem ser utilizados para controlar o estado, ciclo de vida e efeitos colaterais dos componentes.

- ***useState***: Hook que permite adicionar estado a um componente.

Retorna um array com dois elementos, o primeiro é o estado atual e o segundo é uma função que permite atualizar esse estado.

Toda vez que esse estado for atualizado, o componente é renderizado novamente.

```
const [count, setCount] = React.useState(0)
```

Exemplo de uso do useState
Documentação do React sobre useState

- ***useEffect***: Hook que permite realizar efeitos colaterais em componentes.

Aceita uma função que será executada toda vez que o componente for renderizado, e um array de dependências, que determina quando a função será executada novamente. Pode ser utilizado para buscas em APIs, atualizar a DOM em resposta a eventos e também adicionar ou remover eventos.

```
React.useEffect(() => {
  console.log('Componente renderizado')
}, [])
```

Exemplo de uso do useEffect

Documentação do React sobre useEffect

- ***useContext***: Hook que permite acessar um contexto, dentro de componentes. Permite que se subscreva a mudanças e atualize o contexto.

```
// Cria context
const Context = React.createContext(null)

// Engloba a context em um componente pai
<Context.Provider value={value}>
  <Componente />
</Context.Provider>

// Todo componente filho pode acessar o contexto
const value = React.useContext(Context)
```

Exemplo de uso do useContext

Documentação do React sobre useContext

Post do Kent C. Dodds sobre boas práticas com useContext

- ***useReducer***: Hook que permite adicionar e controlar um estado complexo a um componente funcional.

Centraliza a lógica em uma função específica e pura, que recebe o estado atual e uma ação, e retorna o novo estado.

```
const initialState = { count: 0 }
const reducer = (state, action) => {
  // Lógica para atualizar o estado
}

// Utiliza o useReducer em um componente
const [state, dispatch] = React.useReducer(reducer, initialState)
```

Exemplo de uso do useReducer

Documentação do React sobre useReducer

- ***useCallback***: Hook que permite memorizar funções.

Retorna uma versão memorizada da função, que só muda se uma das dependências mudar.

```
const memoizedCallback = React.useCallback(() => {
  doSomething(a, b)
}, [a, b])
```

Exemplo de uso do useCallback

Documentação do React sobre useCallback

- ***useMemo***: Hook que permite memorizar valores.

Muito semelhante ao useCallback. Retorna um valor memorizado, que só muda se uma das dependências mudar.

```
const memoizedValue = React.useMemo(() => computeExpensiveValue(a, b), [a, b])
```

Exemplo de uso do useMemo

Documentação do React sobre useMemo

Estilização

O React Native utiliza um subconjunto de propriedades CSS para estilização.

Os estilos são criados utilizando JavaScript, através de objetos que contém propriedades e valores.

Os nomes das propriedades são escritos em **camelCase** e os valores são escritos como strings.

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

export default App() {
  return (
    <View style={styles.container}>
      <Text style={{ color: 'red' }}>Hello World!</Text>
    </View>
  )
}
```

Documentação do React Native sobre estilização

> Modelo de layout flexbox

Flexbox é um modelo de layout que permite o design de layout responsivo e previsível, sem depender de tamanhos específicos de tela.

O React Native implementa o layout Flexbox de uma maneira mais simplificada.

Algumas das principais propriedades são:

1. `flex`:

A propriedade flex determina como o componente vai crescer ou encolher para preencher o espaço disponível do container.
Aceita um valor numérico que representa o fator flex.

Por exemplo, um componente com **flex: 2** vai ocupar o dobro do espaço de um componente com **flex: 1**.

2. `flexDirection`:

A propriedade **flexDirection** determina a direção principal do layout, se é uma coluna ou uma linha. Aceita os valores **row**, **row-reverse**, **column** e **column-reverse**.

Por exemplo, **row** coloca os itens da esquerda para a direita e **column** coloca os itens de cima para baixo.

3. `justifyContent`:

A propriedade **justifyContent** alinha os itens no eixo principal. Aceita os valores **flex-start**, **center**, **flex-end**, **space-around**, **space-between** e **space-evenly**.

Por exemplo, **justifyContent: flex-start** alinha os itens no início do container e **flex-end** alinha os itens no final do container.

4. `alignItems`:

A propriedade **alignItems** alinha os itens no eixo secundário. Aceita os valores **flex-start**, **center**, **flex-end**, **stretch** e **baseline**.

Por exemplo, **alignItems: center** alinha os itens no contro do eixo secundário do container.

5. `alignSelf`:

A propriedade **alignSelf** sobrescreve a propriedade **alignItems** do container para um único item. Aceita os valores **auto**, **flex-start**, **center**, **flex-end**, **stretch** e **baseline**.

Por exemplo, **alignSelf: flex-end** alinha um item no final do container.

Documentaçao do React Native sobre Flexbox
CSS Tricks Guia completo de Flexbox

Conectando com APIS (Network)

A maneira mais simples de realizar requisições HTTP em React Native é utilizando a API ***fetch***.

A função ***fetch*** é nativa do JavaScript, e é utilizada para realizar requisições HTTP, como GET, POST, PUT, DELETE, etc.

Por ser uma operação assíncrona, a função ***fetch*** retorna uma ***Promise***, que pode ser tratada com ***then*** e ***catch***.

```
fetch('https://api.github.com/users/satinp')
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error(error))
```

É importante tratar os erros, pois a função ***fetch*** pode retornar um erro caso a requisição falhe.

Também é possível utilizar a função ***async/await*** para realizar requisições HTTP, que é uma maneira mais limpa e legível de lidar com operações assíncronas.

```
const fetchData = async () => {
  try {
    const response = await fetch('https://api.github.com/users/satinp')
    const data = await response.json()
    console.log(data)
  } catch (error) {
    console.error(error)
  }
}
```

AsyncStorage

O ***AsyncStorage*** é uma API que permite armazenar dados de forma assíncrona na memória local do dispositivo do usuário.

Isso significa que você pode salvar dados como configurações do aplicativo, preferências do usuário e outros dados que precisam ser persistentes mesmo após o aplicativo ser fechado.

É semelhante ao ***localStorage*** do navegador, porém é assíncrono e não bloqueia a thread principal.

Situações comuns de uso do ***AsyncStorage*** incluem:

- Salvar e recuperar preferências do usuário (nome e-mail, etc);
- Salvar dados de configuração do aplicativo (tema, idioma, etc);
- Persistir dados que precisam ser acessados offline (histórico de pesquisa, etc).
- Armazenar tokens de autenticação.

Algumas Vantagens do *AsyncStorage*:

- ***Fácil de usar***: A API é simples e fácil de usar, com métodos para salvar, recuperar e remover dados.
- ***Assíncrono***: Não bloqueia a thread principal, permitindo que o aplicativo continue funcionando enquanto os dados são salvos ou recuperados.
- ***Persistente***: Os dados são armazenados localmente no dispositivo do usuário, e são persistentes mesmo após o aplicativo ser fechado.

Pacote Async Storage, ReactNative.dev AsyncStorage

Custom hooks

Os ***custom hooks*** são funções JavaScript que utilizam hooks do React, e podem ser reutilizadas em vários componentes funcionais. Suas principais características são:

1. Prefixo com ***use***, exemplo: `useFetch`, `useTheme`, etc;
2. Devem utilizar outros hooks do React, caso contrário podem ser funções normais;
3. Um custom hook pode gerenciar seu próprio estado, lógica e efeitos colaterais;
4. Não renderiza componentes, apenas encapsula lógica.

São, resumidamente, funções que utilizam hooks do React, e podem ser reutilizadas em vários componentes funcionais.

Comumente utilizados quando possuímos lógica compartilhada entre vários componentes, por exemplo, uma função que realiza uma requisição HTTP:

```
import React from 'react'

const useFetch = (url) => {
  const [data, setData] = React.useState(null)
  const [loading, setLoading] = React.useState(true)
  const [error, setError] = React.useState(null)

  React.useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url)
        const result = await response.json()
        setData(result)
      } catch (err) {
        setError(err)
      } finally {
        setIsLoading(false)
      }
    }
    fetchData()
  }, [url])

  return { data, loading, error }
}
```

```jsx
import React from 'react'
import { View, Text } from 'react-native'
import useFetch from './useFetch' // Importe o hook

const ReactComponent = () => {
  const { data, isLoading, error } = useFetch('https://api.example.com/data')

  if (isLoading) {
    return <Text>Carregando...</Text>
  }

  if (error) {
    return <Text>Houve um erro: {error.message}</Text>
  }

  return (
    <View>
      <Text>Retorno da chamada: {JSON.stringify(data)}</Text>
    </View>
  )
}

export default ReactComponent
```

Links úteis:

Reusando logica com custom hooks

Exemplos de chamadas fetch

Conforme mencionado anteriormente, a função ***fetch*** é utilizada para realizar requisições HTTP, e pode ser utilizada para diversos tipos de chamadas, como GET, POST, PUT, DELETE, etc.

```
// GET
try {
  const response = await fetch('www.get-url.com')
  const data = await response.json()
  console.log(data)
} catch (error) {
  console.error(error)
}
```

O método fetch aceita um segundo argumento, um objeto de opções, que pode ser utilizado para configurar a requisição, como método, headers, body, etc.

```
// POST
try {
  const response = await fetch('www.post-url.com', {
    method: 'POST',
    mode: 'cors',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ key: 'value' }),
  })
} catch (error) {
  console.error(error)
}
```

```
// DELETE
try {
  const id = 1
  const response = await fetch(`www.delete-url.com/${id}`, {
    method: 'DELETE',
    mode: 'cors',
    headers: {
      'Content-Type': 'application/json',
    },
  })
} catch (error) {
  console.error(error)
}
```

```
// PUT
try {
  const id = 1
  const response = await fetch(`www.put-url.com/${id}`, {
    method: 'PUT',
    mode: 'cors',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ key: 'value' }),
  })
} catch (error) {
  console.error(error)
}
```

Docker

O que é docker?

- Iniciado em 2013, ***Docker*** é uma plataforma de código aberto que permite que você crie, teste e implante aplicativos rapidamente.
- Permite que você empacote um aplicativo com todas as partes de que ele precisa, como bibliotecas e outras dependências, e o **envie** como um único pacote.
- Similares a máquinas virtuais, mas mais leves e rápidas, pois compartilham o kernel do sistema operacional.

Vantagens do Docker

- ***Portabilidade***: O Docker é executado em qualquer lugar, seja na nuvem, no seu laptop, em um servidor ou data center.
- ***Consistência***: O Docker garante que o ambiente de desenvolvimento seja o mesmo que o ambiente de produção.
- ***Eficiência***: O Docker permite que você execute mais aplicativos em um único servidor, economizando recursos.
- ***Escalabilidade***: O Docker facilita a escalabilidade de aplicativos pela sua leveza, adicionando ou removendo contêineres eficientemente.

Desvantagens do Docker

- ***Complexidade***: O Docker pode ser complexo para iniciantes, pois envolve a criação de imagens, contêineres, redes, volumes, etc.
- ***Desempenho***: O Docker pode ser mais lento do que a execução de aplicativos diretamente no sistema operacional, devido à virtualização.

O que é uma máquina virtual?

- Uma ***máquina virtual*** é um software que emula um computador físico, permitindo que você execute vários sistemas operacionais em um único hardware.
- Cada máquina virtual inclui um sistema operacional, aplicativos e bibliotecas, é isolada de outras máquinas virtuais.
- As máquinas virtuais são mais pesadas e lentas, pois incluem um sistema operacional completo.

O que é um container?

- Um ***container*** é uma instância de uma imagem Docker, um pacote de leitura somente com um conjunto de instruções para criar um contêiner.
- Um contêiner é uma instância isolada de um sistema operacional, que pode ser executado, parado, movido e deletado.
- Os contêineres são leves e rápidos, pois compartilham o kernel do sistema operacional com outros contêineres.

Casos de uso comuns do Docker:

- ***Desenvolvimento e testes***: O Docker é amplamente utilizado para desenvolvimento e testes de aplicativos, pois permite que você crie ambientes isolados e consistentes.
- ***Estudo e aprendizado***: Permite que você crie e inicie ambientes de desenvolvimento com facilidade, sem afetar o sistema operacional.
- ***Microserviços***: O Docker é uma escolha popular para arquiteturas de microsserviços, pois permite que você empacote e implante serviços independentes.
- ***CI/CD***: O Docker é frequentemente usado em pipelines de CI/CD para criar, testar e implantar aplicativos de forma automatizada.

Terminologia do Docker

- ***Docker Client***: A interface de linha de comando que permite que você interaja com o Docker.
- ***Docker Container***: Um pacote de software com todas as dependências necessárias para executar um aplicativo específico.
- ***Docker Image***: Um ou mais arquivos de leitura, que contém um conjunto de instruções para criar um contêiner. Pode ser compartilhado e reutilizado em diferentes ambientes.

- ***Docker Server***: Também conhecido como Docker daemon, é o comando `dockerd`, responsável por criar e gerenciar contêineres pelo client. Faz o trabalho de construir, executar e distribuir contêineres.
- ***Docker Registry***: Um serviço que armazena e distribui imagens do Docker. O Docker Hub é o registro público mais conhecido.

Glossário Docker

Utilizando containeres docker

Os containeres docker são executados a partir de imagens, que são arquivos de leitura somente, que contém um conjunto de instruções para criar um contêiner. Imagens podem ser encontradas no site do Docker Hub.

Ou também através do comando

```
docker search {nome da imagem}
```

que busca imagens do Docker Hub.

Rodando containers

Alguns comandos úteis do Docker:

```
# Lista comandos da CLI do Docker
docker --help
```

```
# Verifica se a imagem do hello-world está disponível no daemon.
# Caso não esteja, baixa a imagem do Registry - Docker Hub
docker container run hello-world
# ou
docker run hello-world
```

```
# Lista todos os contêineres em execução
# Pode retornar containers finalizados se passado o argumento -a
docker container ls
# ou
docker ps
```

```
# Atribui um nome ao contêiner
docker container run --name c-hello-world hello-world
```

```
# Inicia um contêiner ubuntu no modo interativo
# O comando /bash abre um shell no contêiner
docker container run -it ubuntu /bash
```

```
# Detalhes do contêiner, baseado no ID.
docker container inspect {Id do container}
```

```
# Remove um contêiner baseado no ID
docker container rm {Id do container}
```

```
# -d: Executa o contêiner em segundo plano (modo daemon)
docker container run -it -d ubuntu
```

```
# Para um contêiner baseado no ID
docker container stop {I do container}
```

```
# Acessa um contêiner em execução, no modo interativo
docker container exec -it {Id do container}
```

```
# -p 8080:80: Mapeia a porta 8080 do host para a porta 80 do contêiner
docker container run -d -p 8080:80 nginx
```

Utilizando imagens docker

Imagens docker são criadas a partir de um arquivo chamado ***Dockerfile***, que contém instruções para criar a imagem.
Alguns exemplos de instruções comuns em um ***Dockerfile*** são:

- ***FROM***: Especifica a imagem base.
- ***ENV***: Define variáveis de ambiente.
- ***WORKDIR***: Define o diretório de trabalho.
- ***RUN***: Executa um comando no contêiner.
- ***COPY***: Copia arquivos do host para o contêiner.
- ***CMD***: Especifica o comando a ser executado quando o contêiner é iniciado.

Resumo Dockerfile

Criando uma imagem ubuntu

```
# Busca uma imagem do ubuntu na sua versão mais recente
FROM ubuntu:latest
# Roda os comandos update e install do apt-get
RUN apt-get update && \
  apt-get install curl -y
```

```
# -t: Tag, nome da imagem a ser criada, satin-ubuntu
# e o diretório onde está o Dockerfile
docker image build -t satin-ubuntu .
```

```
# Exibe todas as baixadas localmente
docker image ls
```

```
# Exibe o histórico de criação da imagem
docker history satin-ubuntu
```

```
# Exibe detalhes da imagem
docker image inspect {nome da imagem}
```

Publicando uma imagem no Docker Hub

```
# Loga no Docker Hub
docker login
# Será solicitado usuário e senha
```

```
# Cria uma nova tag para a imagem
docker tag satin-ubuntu psatin/ubuntu
# psatin é o nome do usuário no Docker hub
# ubuntu é o nome da imagem
```

```
# Publica a imagem no Docker Hub
docker push psatin/satin-ubuntu
```

Docker Compose

O ***Docker Compose*** é uma ferramenta que permite definir e executar aplicativos Docker multi-contêineres. Ele utiliza um arquivo chamado ***docker-compose.yml*** para configurar os serviços do aplicativo.

Ele é útil para desenvolvimento, teste e produção, pois permite que você defina e execute vários contêineres com um único comando.

Exemplo de arquivo docker-compose.yml

```yaml
version: '3'
services:
  web:
    image: nginx:latest
    ports:
      - '8080:80'
  db:
    image: mysql:latest
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: mydb
```

No exemplo acima temos dois serviços, ***web*** e ***db***. O serviço ***web*** utiliza a imagem ***nginx:latest*** e mapeia a porta ***8080*** do host para a porta ***80*** do contêiner. O serviço ***db*** utiliza a imagem ***mysql:latest*** e define variáveis de ambiente para o MySQL.

Para iniciar os serviços definidos no arquivo ***docker-compose.yml***, basta rodar o comando:

```
docker-compose up
```

Para parar os serviços, basta rodar o comando:

```
docker-compose down
```

Documentação Docker Compose

Referências:

Livros:

- Learn all about React Native - Innoware PJP
- Docker: Up & Running

Sites/Documentações/Bibliotecas:

- React Native Doc, Expo doc, React navigation,Async Storage,
  Docker Docs