



Apostila de Treinamento

Parte 1

4Quality
tecnologia

Tabela de conteúdos

Introdução	1.1
React Native	1.2
Montando o Ambiente	1.3
Introdução ao React Native	1.4
Iniciando o Projeto	1.5
Mais componentes	1.6
Atualizando o estado do componente	1.7
Uma lista de usuários	1.8
Melhorando o Layout	1.9
Um componente para favoritar	1.10
Centralizando o Estado	1.11
Redux	1.12
Conectando às APIs	1.13
Login	1.14
Navegando entre telas	1.15

Introdução

O framework ReactNative é a última palavra em desenvolvimento multiplataforma de aplicativos mobile. Dentro os tópicos do treinamento, abordaremos os conceitos fundamentais do framework, além de ferramentas e bibliotecas que auxiliam o dia a dia de um desenvolvedor ReactNative. Para você profissional de TI que gosta de se manter atualizado ao que há de mais moderno em termos de desenvolvimento mobile, este treinamento é pra você!

Conhecimentos desejáveis

É interessante que o aluno possua boa fluência com a linguagem JavaScript versão ES6 para um melhor rendimento no treinamento. Há um material de altíssima qualidade disponibilizado de forma gratuita no link <https://javascript.info/> para quem desejar aprofundar os conhecimentos.

Visão geral do treinamento

O treinamento tem foco prático na construção de uma aplicação padrão React Native com as seguintes funcionalidades:

- Listagem de animais;
- Favoritar um animal;
- Login/Logout;
- Navegação entre telas;
- Demais operações de CRUD (Create/Retrieve/Update/Delete) de animais;

Para isso, utilizaremos as seguintes tecnologias/bibliotecas além do React Native:

- *React*: Uma biblioteca JavaScript para construir interfaces com o usuário;
- *NativeBase*: Componentes *cross platform* para ReactNative;
- *React Navigation*: Roteamento e navegação para aplicativos React Native;
- *Redux*: Contêiner de estado previsível para aplicativos JavaScript;
- *Axios*: Cliente HTTP baseado em promessa para o navegador e node.js;
- *Redux Thunk*: Permite que se escreva criadores de ações que retornam uma função ao invés de uma ação;
- *Redux Form*: A melhor maneira de gerenciar estado de formulários no Redux;
- *React DevTools*: Permite inspecionar a hierarquia do componente React, incluindo o componente props e state;
- *Reactotron*: Um aplicativo macOS, Windows e Linux para inspecionar seus aplicativos React JS e React Native;

React Native

Aplicativos móveis nativos usando JavaScript e React

O React Native permite a criação de aplicativos móveis usando apenas JavaScript. Ele usa o mesmo design que o React, permitindo compor uma rica interface de usuário a partir de componentes declarativos.

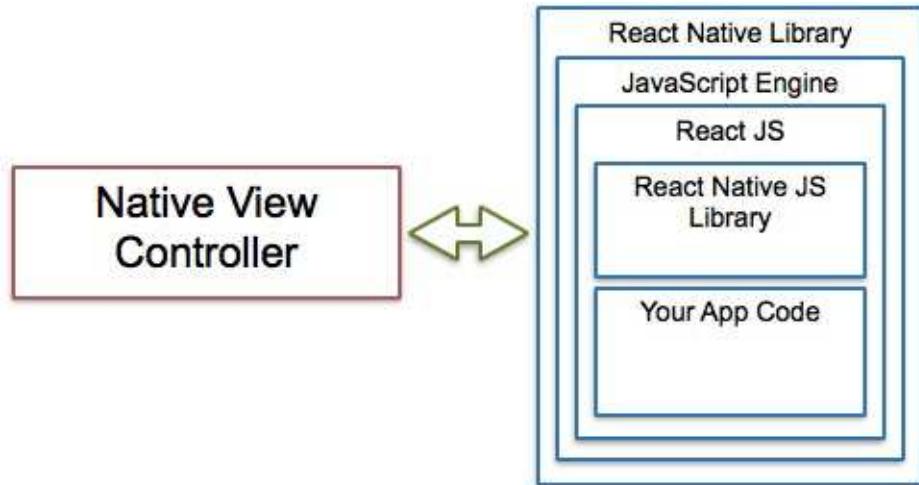
Um aplicativo React Native é um aplicativo móvel nativo

Com o React Native, não é criado um "aplicativo da Web para dispositivos móveis", um "aplicativo HTML5" ou um "aplicativo híbrido". Será criado um aplicativo móvel real que é indistinguível de um aplicativo criado usando o Objective-C ou o Java. O React Native usa os mesmos blocos de construção fundamentais de interface com o usuário dos aplicativos iOS e Android comuns. Esses blocos de construção são colocados juntos usando JavaScript e React.

```
import React, { Component } from "react";
import { Text, View } from "react-native";

class MeuComponente extends Component {
  render() {
    return (
      <View>
        <Text>React Native é o React aplicado ao desenvolvimento mobile.</Text>
        <Text>
          São utilizados componentes nativos como 'View' e 'Text', ao invés dos
          componentes web como 'div' e 'span'.
        </Text>
      </View>
    );
  }
}
```

Diagrama



Recompilação mais rápida

O React Native permite que se construa um aplicativo mais rapidamente. Em vez de recompilar, pode-se recarregar o aplicativo instantaneamente. Com o Hot Reloading, pode-se até mesmo executar um novo código, mantendo o estado do aplicativo.

Código nativo quando for necessário

React Native pode ser combinado com componentes escritos em Objective-C, Java ou Swift. É possível utilizar código nativo se for necessário otimizar alguns aspectos do aplicativo. Também é possível criar parte do aplicativo no React Native e parte usando o código nativo diretamente.

Montando o Ambiente

Para começarmos nosso desenvolvimento, necessitaremos das seguintes ferramentas instaladas no computador:

- Android Studio: <https://developer.android.com/studio/>
- Java JDK: <https://www.oracle.com/java/technologies/jdk8-downloads.html>
- NodeJS (8.3+) e NPM: <https://nodejs.org/en/download/>
- React Native CLI: `npm install -g react-native-cli`
- VSCode - <https://code.visualstudio.com/>

São recomendados os seguintes plugins para o VSCode:

- EditorConfig;
- ESLint;
- Prettier;
- vscode-icons;

Algumas variáveis de ambiente devem ser configuradas nos arquivos `$HOME/.bash_profile` ou `$HOME/.bashrc` no caso do Linux ou via Variáveis de Ambiente do Windows:

```
export ANDROID_HOME=$HOME/Android/Sdk
export PATH=$PATH:$ANDROID_HOME/emulator
export PATH=$PATH:$ANDROID_HOME/tools
export PATH=$PATH:$ANDROID_HOME/tools/bin
export PATH=$PATH:$ANDROID_HOME/platform-tools
```

Testando instalação

Podemos testar a instalação de algumas das ferramentas:

```
> node --version
v10.16.0
> npm --version
6.9.0
> react-native --version
react-native-cli: 2.0.1
react-native: n/a - not inside a React Native project directory
...
```

Introdução ao React Native

React

O React é uma biblioteca JavaScript declarativa, eficiente e flexível para criar interfaces com o usuário. Ele permite compor interfaces de usuário complexas a partir de pequenos e isolados códigos chamados "componentes":

```
class ListaCompras extends React.Component {
  render() {
    return (
      <div className="lista-compras">
        <h1>Lista de Compras para: {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}
```

Utiliza-se componentes para dizer ao React o que será exibido na tela. Quando os dados forem alterados, o React atualizará e renderizará novamente com eficiência os componentes.

Aqui, o `ListaCompras` é uma classe de componente React ou o tipo de componente React. Um componente recebe parâmetros, chamados `props` (abreviação de "propriedades"), e retorna uma hierarquia de componentes visuais para exibir através do método `render`.

O método `render` retorna uma descrição do que se deseja ver na tela. React pega a descrição e exibe o resultado. Em particular, `render` retorna um elemento React, que é uma descrição simples do que renderizar. A maioria dos desenvolvedores do React usa uma sintaxe especial chamada *JSX*, que facilita a gravação dessas estruturas. A sintaxe `<div/>` é transformada no momento da criação para `React.createElement('div')`. O exemplo acima é equivalente a:

```
return React.createElement(
  'div',
  {className: 'lista-compras'},
  React.createElement('h1' /* ... filhos de h1 ... */),
  React.createElement('ul' /* ... filhos de ul ... */),
);
```

O JSX possui todo o poder do JavaScript. Coloca-se qualquer expressão JavaScript dentro de chaves dentro do JSX. Cada elemento React é um objeto JavaScript que se pode armazenar em uma variável ou passar ao programa.

O componente `ListaCompras` acima apenas renderiza componentes DOM internos, como `<div>` e ``. Mas também pode compor e renderizar componentes React personalizados. Por exemplo, podemos nos referir a toda a lista de compras escrevendo `<ListaCompras/>`. Cada componente React é encapsulado e pode operar de forma independente; Isso permite que se construa interfaces com o usuário complexas a partir de componentes simples.

Podemos testar a compilação de código JSX de forma online através do link: <https://babeljs.io/repl/>. Insira o código abaixo:

```
<div class="foo">
  <span>Teste</span>
</div>
```

Virtual DOM

Primeiramente, DOM significa "Document Object Model". O DOM em palavras simples representa a interface do usuário do seu aplicativo. Sempre que há uma alteração no estado da interface do usuário do aplicativo, o DOM é atualizado para representar essa alteração. Agora, a dificuldade é que manipular frequentemente o DOM afeta o desempenho, tornando-o lento.

O que torna a manipulação do DOM lenta?

O DOM é representado como uma estrutura de dados em árvore. Por esse motivo, as alterações e atualizações no DOM são rápidas. Mas após a alteração, o elemento atualizado e seus filhos precisam ser renderizados novamente para atualizar a interface do usuário do aplicativo. A nova renderização ou nova pintura da interface do usuário é o que a torna lenta. Portanto, quanto mais componentes de interface do usuário você tiver, mais caras serão as atualizações do DOM, pois elas precisarão ser renderizadas novamente para cada atualização do DOM.

DOM virtual

É aí que o conceito de DOM virtual entra e apresenta um desempenho significativamente melhor que o DOM real. O DOM virtual é apenas uma representação virtual do DOM. Sempre que o estado de nosso aplicativo é alterado, o DOM virtual é atualizado em vez do DOM real.

Bem, você pode perguntar "O DOM virtual não está fazendo a mesma coisa que o DOM real, isso soa como um trabalho duplo? Como isso pode ser mais rápido do que apenas atualizar o DOM real?"

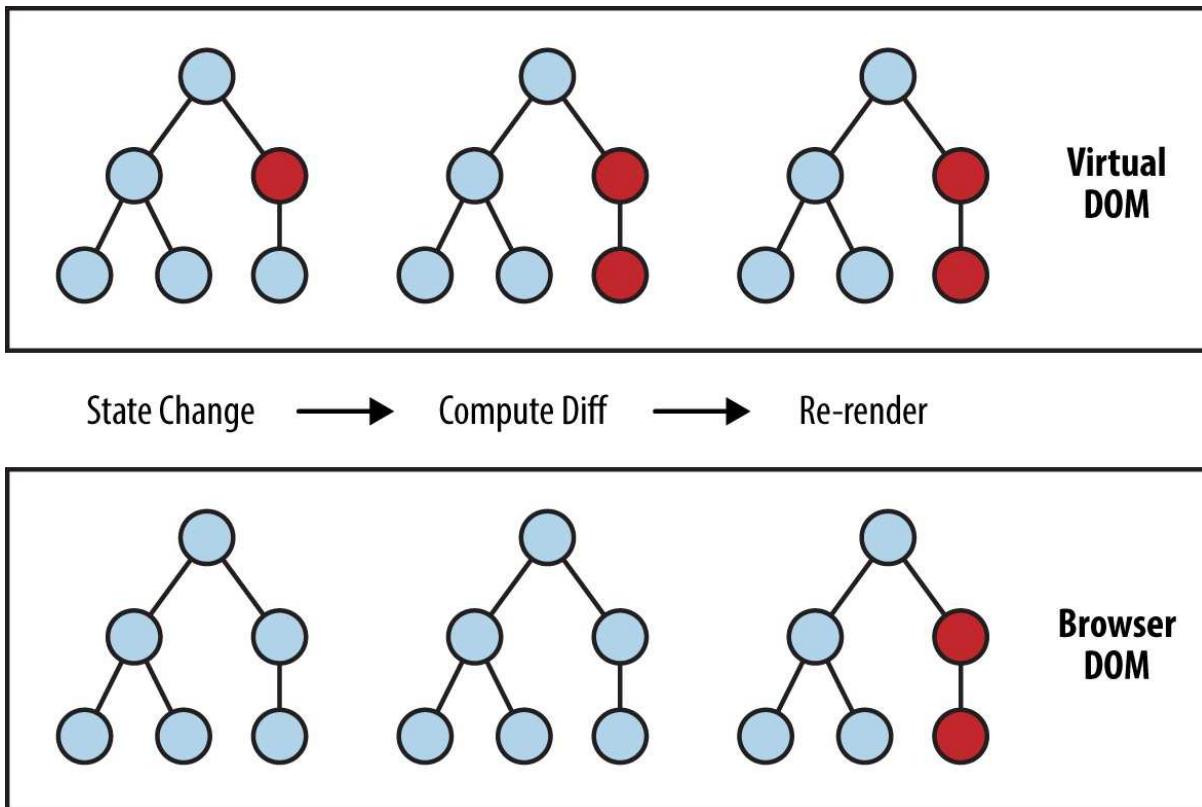
A resposta é que o DOM virtual é muito mais rápido e eficiente, eis o porquê.

Como o DOM virtual é mais rápido?

Quando novos elementos são adicionados à interface do usuário, um DOM virtual, representado como uma árvore, é criado. Cada elemento é um nó nesta árvore. Se o estado de qualquer um desses elementos for alterado, uma nova árvore DOM virtual será criada. Essa árvore é então comparada ou "diferenciada" com a árvore DOM virtual anterior.

Feito isso, o DOM virtual calcula o melhor método possível para fazer essas alterações no DOM real. Isso garante que haja operações mínimas no DOM real. Portanto, reduzindo o custo de desempenho da atualização do DOM real.

A imagem abaixo mostra a árvore DOM virtual e o processo de diferenciação.



(Adaptado de: <https://programmingwithmosh.com/react/react-virtual-dom-explained/>)

React Native

O React Native é uma solução multi-plataforma para escrever aplicativos móveis nativos. O Facebook abriu o código fonte do React Native em março de 2015. Eles o construíram porque, como muitas empresas, o Facebook precisava disponibilizar seus produtos na Web, bem como em várias plataformas móveis, e é difícil manter equipes especializadas necessárias para construir um mesmo app em diferentes plataformas. Depois de experimentar várias técnicas diferentes, a React Native foi a solução do Facebook para o problema.

O que faz o React Native Diferente?

Já existem soluções para criar aplicativos para dispositivos móveis: desde escrever código nativo em linguagens proprietárias até escrever *aplicativos da Web para dispositivos móveis* ou soluções híbridas. Então, por que os desenvolvedores precisam de outra solução? Por que eles deveriam dar uma chance ao React Native?

Ao contrário de outras opções disponíveis, o React Native permite que os desenvolvedores escrevam aplicativos nativos no iOS e no Android usando JavaScript com o React em uma única base de código. Ele usa os mesmos princípios de design usados pelo React na Web e permite criar interfaces usando o modelo de componente que já é familiar aos desenvolvedores. Além disso, ao contrário de outras opções que permitem usar tecnologias da Web para criar aplicativos híbridos, o React Native é executado no dispositivo usando os mesmos blocos de construção fundamentais usados pelas soluções específicas da plataforma, tornando-a uma experiência mais natural para os usuários.

(Adaptado de: <http://engineering.monsanto.com/2018/01/11/react-native-intro/>)

Comparando React com React Native

Configuração e Construção

React-Native é um framework e o ReactJS é uma biblioteca de JavaScript. Quando se inicia um novo projeto com o ReactJS, deverá ser escolhido um bundler como o Webpack que tentará descobrir quais módulos de empacotamento são necessários para o seu projeto. O React-Native vem com tudo o que se precisa. Quando se inicia um novo projeto React Native é fácil de configurar: leva apenas uma linha de comando para ser executada no terminal e está pronto para começar. Pode-se começar a codificar o primeiro aplicativo nativo imediatamente usando o ES6, alguns recursos do ES7 e até mesmo alguns polyfills.

Para executar o aplicativo, precisará ter o Xcode (para iOS, somente no Mac) ou o Android Studio (para Android) instalado em seu computador. Pode-se optar por executá-lo em um simulador/emulador da plataforma que deseja usar ou diretamente em seus próprios dispositivos.

DOM e Estilização

O React-Native não usa HTML para renderizar o aplicativo, mas fornece componentes alternativos que funcionam de maneira semelhante. Esses componentes React-Native mapeiam os componentes reais reais da interface iOS ou Android que são renderizados no aplicativo.

A maioria dos componentes fornecidos pode ser traduzida para algo semelhante em HTML, onde, por exemplo, um componente View é semelhante a uma tag div e um componente Text é semelhante a uma tag p.

```
import React, {Component} from 'react';
import {View, Text} from 'react-native';

export default class App extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.intro}>Hello world!</Text>
      </View>
    );
  }
}
```

Para estilizar os componentes React-Native, é necessário criar folhas de estilo em JavaScript.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  content: {
    backgroundColor: '#fff',
    padding: 30,
  },
  button: {
    alignSelf: 'center',
    marginTop: 20,
    width: 100,
  },
});
```

(Adaptado de: <https://medium.com/@alexmngn/from-reactjs-to-react-native-what-are-the-main-differences-between-both-d6e8e88ebf24>)

Expo vs React Native CLI

Existem dois métodos para inicializar e desenvolver seu aplicativo: Expo e a CLI do React Native. Até recentemente, havia um terceiro método distinto, usando o Create React Native App (CRNA). Desde então, a CRNA foi incorporada ao projeto Expo e só continua a existir como uma entidade separada para fornecer compatibilidade com versões anteriores.

Para usuários com pouca experiência de desenvolvimento, a maneira mais fácil de começar é com o Expo CLI. O Expo é um conjunto de ferramentas criadas em torno do React Native e, embora tenha muitos recursos, o recurso mais relevante para nós no momento é que você pode escrever um aplicativo React Native em poucos minutos. Você precisará apenas de uma versão recente do Node.js e de um telefone ou emulador.

Desenvolvedores mais experientes podem usar o React Native CLI. Requer o Xcode ou o Android Studio para começar. Se você já possui uma dessas ferramentas instaladas, poderá criar uma aplicação em alguns minutos.

A Expo se enquadra na categoria de ferramentas, fornecendo um fluxo de trabalho de desenvolvimento mais robusto e amigável ao desenvolvedor, ao custo de alguma flexibilidade. Os aplicativos iniciados com a Expo também têm acesso a vários recursos úteis fornecidos pelo Expo SDK, como o BarcodeScanner, o MapView, o ImagePicker e muito mais.

A inicialização de um aplicativo com a CLI do React Native, por meio do comando react-native init fornece flexibilidade ao custo da facilidade de desenvolvimento. Diz-se que um aplicativo React Native criado com o comando react-native init é um aplicativo React Native puro, pois nenhum código nativo está oculto ao desenvolvedor.

Em nosso caso utilizaremos o React Native CLI durante nosso treinamento.

Iniciando o Projeto

Podemos inicializar e executar nosso projeto com muita facilidade utilizando o `react-native-cli` :

```
> react-native init CoZooMob
```

Inicie o Android Studio e execute uma instância do emulador.

Abra a pasta do projeto no VS Code, em seguida ative o terminal: `Terminal -> New Terminal`

Divida o terminal em dois: *Split Terminals*

Em um terminal entre com o comando:

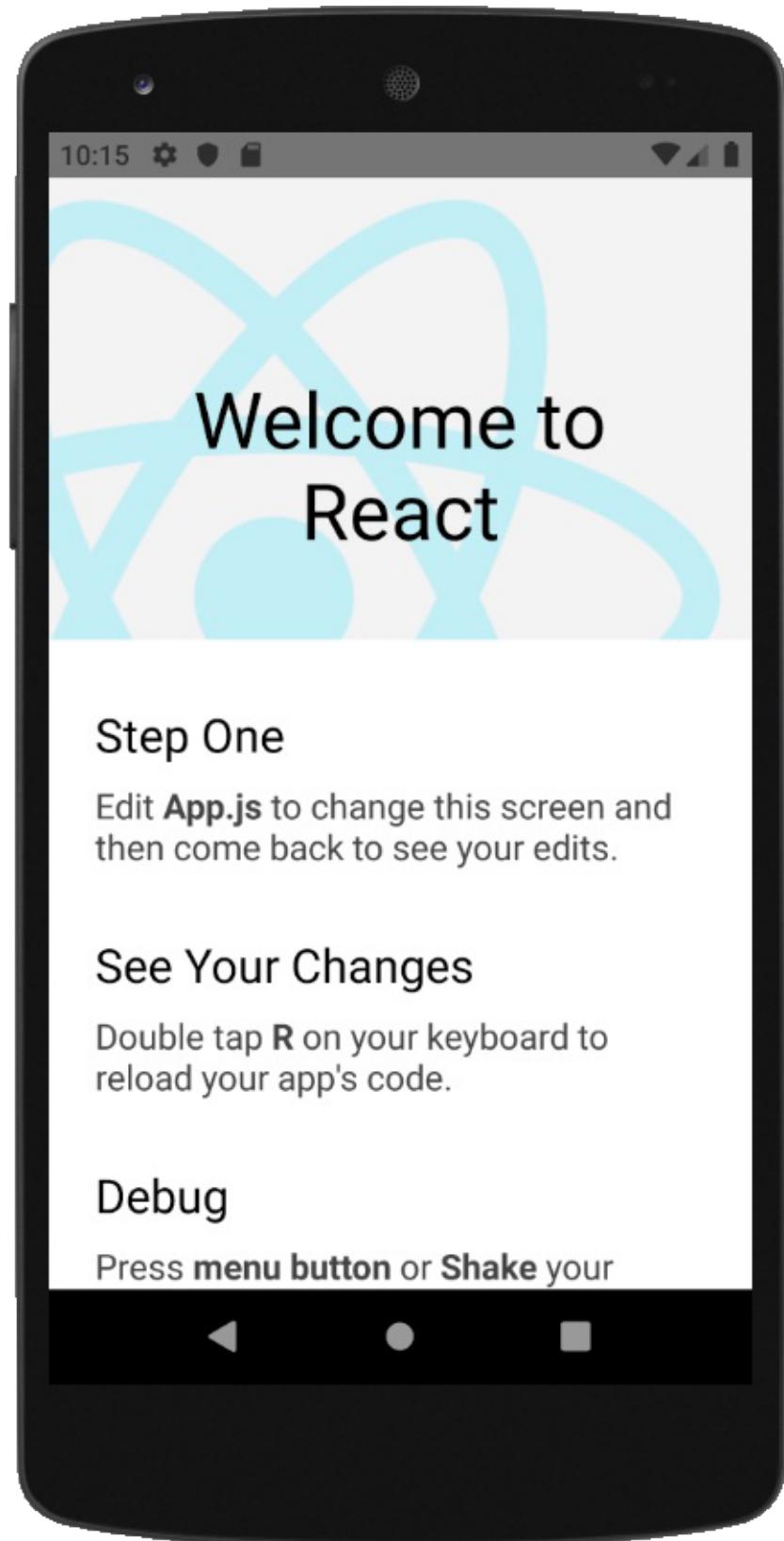
```
> react-native start
```

E no outro:

```
> react-native run-android
```

Obs.: O comando `react-native run-android` deve ser executado após o emulador do Android estar em execução.

A aplicação deverá ser executada no emulador conforme o print abaixo:



Conhecendo a estrutura básica de um projeto React Native

- `__tests__` : Pasta de testes do projeto;

- `android` : Pasta do projeto nativo Android;
- `ios` : Pasta do projeto nativo iOS;
- `node_modules` : Pasta dependências JavaScript;
- `.buckconfig` : Configuração do sistema de build Buck;
- `.eslintrc.js` : Configuração do linter de código;
- `.flowconfig` : Configuração do checador de tipagem estática Flow;
- `.gitattributes` : Configurações do git;
- `.gitignore` : Configuração de arquivos ignorados pelo git;
- `.prettierrc.js` : Configuração do formatador de código Prettier;
- `.watchmanconfig` : Configuração da ferramenta de monitoração Watchman;
- `App.js` : A primeira tela (e componente) de nosso aplicativo!
- `app.json` : Algumas propriedades de nosso aplicativo;
- `babel.config.js` : Configuração do transpilador babel;
- `index.js` : Ponto de partida inicial de nossa aplicação;
- `metro.config.js` : Configuração do bundler metro;
- `package-lock.json` : Descritor da árvore de dependências completa do projeto;
- `package.json` : Descritor do projeto NPM;

Ajustando a tela inicial

Vamos alterar o arquivo `App.js` para que tenha o seguinte conteúdo:

- `App.js`

```
import React, {Component} from 'react';
import {Text, View} from 'react-native';

export default class App extends Component {
  render() {
    return (
      <View style={{flex: 1, justifyContent: 'center', alignItems: 'center'}}>
        <Text>Controle de Zoológico Mobile!</Text>
      </View>
    );
  }
}
```

Obs.: Caso ocorra um erro do tipo:

```
Error: ESLint configuration in .eslintrc.js » @react-native-community/eslint-config
is invalid:
  - Property "overrides" is the wrong type (expected array but got `{"files":["**/*_tests_/**/*.*js","**/*.*spec.js","**/*.*test.js"],"env":{"jest":true,"jest/globals":true}}`).
```

Isto ocorre devido a uma incompatibilidade entre a versão mais nova do ESLint e a configuração do react (<https://github.com/facebook/create-react-app/issues/7284>), e pode ser resolvida através do *downgrade* da versão do ESLint do projeto, alterando a seguinte linha no `package.json` :

```
...
  "devDependencies": {
    "eslint": "5.16.0"
  }
...
```

Em seguida, devemos remover e reinstalar as dependências:

```
> rm -rf node_modules
> npm install
```

Auto-formatação do Código

Para facilitar com que nosso time utilize o mesmo padrão de codificação, iremos configurar para que o VS Code formate automaticamente nosso código: File -> Preferences -> Settings -> User -> Format on Save: True

É necessário que os plugins ESLint e Prettier estejam instalados no VS Code.

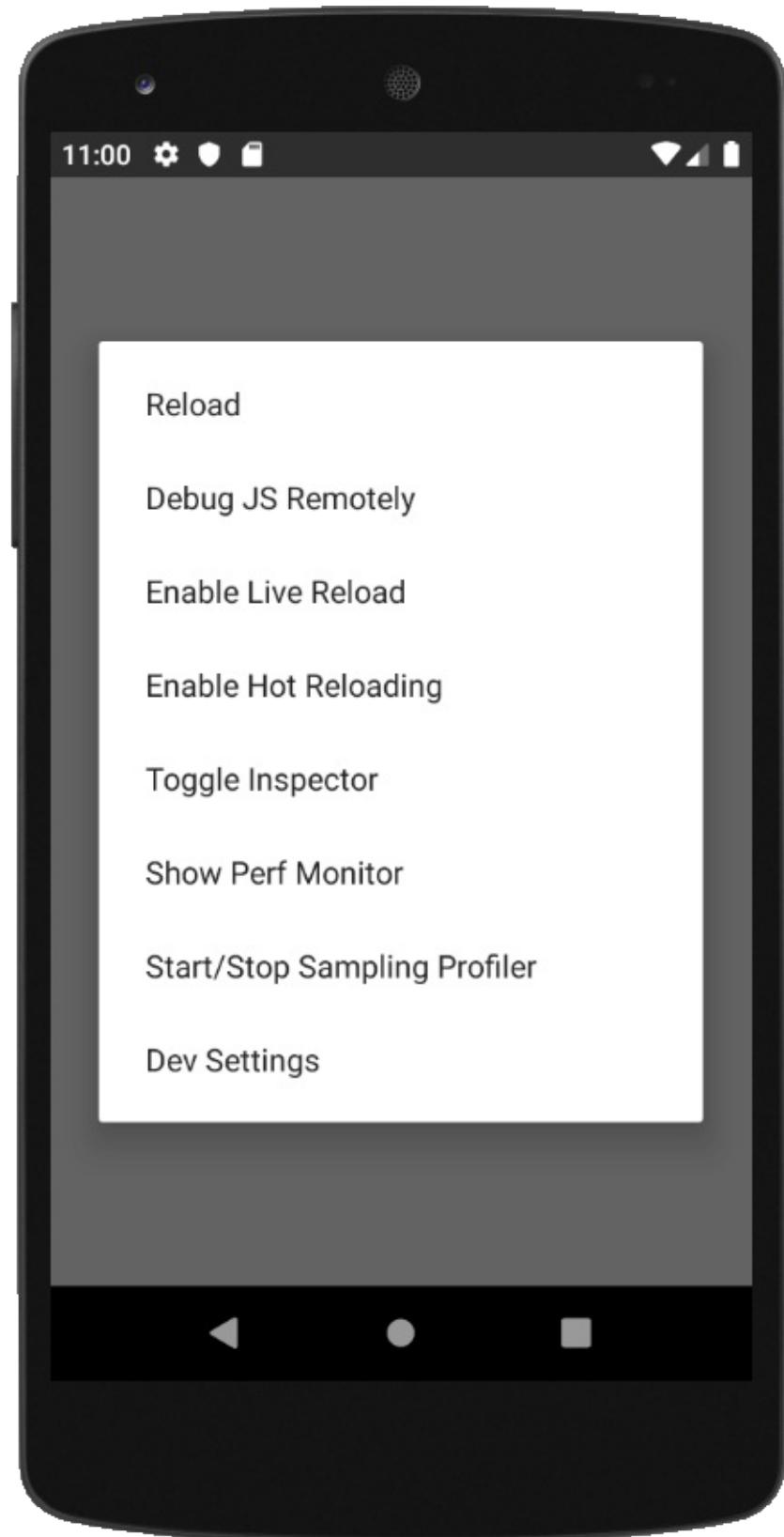
Agora, sempre que algum arquivo for salvo, ele será automaticamente formatado.

Conhecendo melhor o VS Code

The screenshot shows the Visual Studio Code interface with the following details:

- Title Bar:** App.js - CoZooMob - Visual Studio Code
- File Menu:** File Edit Selection View Go Debug Terminal Help
- Explorer:** Shows the project structure under COZOOLOB, including files like App.js, _tests_, android, ios, node_modules, .buckconfig, .eslintrc.js, .flowconfig, .gitattributes, .gitignore, .prettierrc.js, and .watchmanconfig.
- Editor:** The main editor window displays the content of App.js, which contains a React Native component named HelloWorldApp. The code includes imports for React and Component from 'react' and Text, View from 'react-native'. The component's render method returns a View with a central Text element containing the text "Controle de Zoológico Mobile!".
- Bottom Status Bar:** Shows the current branch as master*, file count (0), and other status indicators.
- Bottom Icons:** Includes icons for GitHub, Prettier, and other extensions.

Conhecendo as Ferramentas de Desenvolvedor



ES2015 (ES6) e JSX

Dois pontos importantes de serem notados é o uso de ES2015 e JSX. O ES2015 nos permite utilizar diversos recursos modernos da linguagem JavaScript que veremos no decorrer do treinamento. Já o JSX é uma extensão do JS que nos permite descrever elementos de forma similar ao XML diretamente no código de nossa aplicação, tornando a escrita de componentes muito mais simples.

Componentes

Componentes são a estrutura básica do React, basicamente tudo que vemos em tela são Componentes ou uma composição de diversos componentes. Criaremos muitos componentes no decorrer de nosso treinamento.

Um componente é uma classe simples, único método obrigatório de um componente é a função `render`, que define a exibição do mesmo.

Melhorando a organização do código

Uma boa prática é já focar desde o começo de nosso projeto em sua organização. Para isso vamos mover nosso componente para uma subpasta que criaremos `src/components/App.js`.

Após a movimentação do arquivo, é necessário atualizar sua referência no arquivo `index.js`.

Exibindo algo mais interessante

Vamos começar exibindo os detalhes de um animal:

- `App.js`

```
import React, {Component} from 'react';
import {Image, Text, View, Dimensions} from 'react-native';

export default class App extends Component {
  render() {
    const {width} = Dimensions.get('screen');

    return (
      <View>
        <Text style={{fontSize: 16}}>Leão</Text>
        <Image
          source={{
            uri:
              'https://upload.wikimedia.org/wikipedia/commons/4/40/Just_one_lion.jpg
            ,
            {}
          }}
          style={{width, height: width}}
        />
      </View>
    );
  }
}
```



Repare na sintaxe de `const {width} = Dimensions.get('screen')`, esta é a chamada atribuição via desestruturação. A sintaxe de atribuição via desestruturação (destructuring assignment) é uma expressão JavaScript que possibilita extrair dados de arrays ou objetos em variáveis distintas (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Atribuicao_via_desestruturacao)

Pensando na melhoria de código, seria mais eficiente que tivéssemos tanto as propriedades do CSS quanto os dados dos animais separados em variáveis:

- App.js

```
const {width} = Dimensions.get('screen');

export default class App extends Component {
  render() {
    const animal = {
      nome: 'Leão',
      urlImagen:
        'https://upload.wikimedia.org/wikipedia/commons/4/40/Just_one_lion.jpg',
    };

    return (
      <View>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        <Image
          source={{
            uri: animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  nomeAnimal: {fontSize: 16},
  imagemAnimal: {width, height: width},
});
```

Vários Animais

Agora que já temos um animal sendo exibido, nosso próximo passo é passar a exibir uma lista de animais, para isso, utilizaremos um array de animais e a função `map`:

- App.js

```

export default class App extends Component {
  render() {
    const animais = [
      {
        nome: 'Leão',
        urlImagen:
          'https://upload.wikimedia.org/wikipedia/commons/4/40/Just_one_lion.jpg',
      },
      {
        nome: 'Girafa',
        urlImagen:
          'https://upload.wikimedia.org/wikipedia/commons/9/97/Namibia_Etosha_Girafe_02.jpg',
      },
      {
        nome: 'Gato',
        urlImagen:
          'https://upload.wikimedia.org/wikipedia/commons/b/b2/WhiteCat.jpg',
      },
    ];
    return (
      <View>
        {animais.map(animal => (
          <View>
            <Text style={styles.nomeAnimal}>{animal.nome}</Text>
            <Image
              source={{uri: animal.urlImagen}}
              style={styles.imagemAnimal}
            />
          </View>
        )));
      </View>
    );
  }
}

```

Alguns pontos merecem destaque aqui:

- Utilizamos o método `map` do objeto array do JavaScript para iterar sobre a lista de animais, perceba que no JSX não há uma sintaxe especial de template, utilizamos as construções do próprio JavaScript;
- É necessário envolver a resposta do `map` em um elemento raiz `<View>`, tente remover este elemento e veja o que ocorre;

Mas ainda faltam acertar alguns pontos, pois é exibido um *warning* referente a *unique key prop*. Para este problema, basta incluirmos a propriedade `key` com valores distintos para cada item da lista:

- App.js

```
return (
  <View>
    {animais.map(animal => (
      <View key={animal.nome}>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        <Image
          source={{
            uri: animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />
      </View>
    )));
  </View>
);
```

Você percebeu como são colocados os comentários no JSX? Utiliza-se `{/* Comentários de Bloco */}`.

Pronto, agora o erro não deve mais ser exibido!

Rolando a Lista

Mas há outro problema, nossa lista não está "rolando" com o gesto de scroll, e não é um erro do emulador. Para que nossa lista possa rolar e exibir os itens mais abaixo devemos trocar o elemento raiz pelo FlatList (<https://facebook.github.io/react-native/docs/flatlist.html>).

- App.js

```
return (
  <View>
    <FlatList
      data={animais}
      renderItem={({item}) => (
        <View>
          <Text style={styles.nomeAnimal}>{item.nome}</Text>
          <Image
            source={{
              uri: item.urlImagen,
            }}
            style={styles.imagemAnimal}
          />
        </View>
      )})
      keyExtractor={item => item.nome}
    />
  </View>
);
```

Agora nossa lista deve estar "rolando" com os gestos do usuário.

Mais componentes

Já temos um componente funcional em nossa tela, mas vamos componentizar ainda mais.

Nosso primeiro passo é nomear corretamente o nosso componente inicial, daremos o nome de

`ListaAnimais`, para isso, iremos renomear o arquivo `App.js` para `ListaAnimais.js` e a classe para `ListaAnimais`:

- `ListaAnimais.js`

```
export default class ListaAnimais extends Component {  
    // Código omitido  
}
```

- `index.js`

```
AppRegistry.registerComponent(appName, () => ListaAnimais);
```

Em seguida iremos decompor nosso componente, criando um componente específico para representar cada animal na listagem. Criaremos o arquivo `Animal.js` dentro da pasta `components`:

- `Animal.js`

```
import React, {Component} from 'react';  
import {Dimensions, Image, StyleSheet, Text, View} from 'react-native';  
  
const {width} = Dimensions.get('screen');  
  
export default class Animal extends Component {  
    render() {  
        const animal = {};  
        return (  
            <View>  
                <Text style={styles.nomeAnimal}>{animal.nome}</Text>  
                <Image  
                    source={{  
                        uri: animal.urlImagen,  
                    }}  
                    style={styles.imagemAnimal}  
                />  
            </View>  
        );  
    }  
}  
  
const styles = StyleSheet.create({  
    nomeAnimal: {fontSize: 16},
```

```
    imagemAnimal: {width, height: width},
});
```

Nosso próximo passo é utilizar este componente para compor a lista de animais:

- `ListaAnimais.js`

```
import React, {Component} from 'react';
import {FlatList, View} from 'react-native';
import Animal from './Animal';

export default class ListaAnimais extends Component {
  render() {
    const animais = [
      {
        nome: 'Leão',
        urlImagen:
          'https://upload.wikimedia.org/wikipedia/commons/4/40/Just_one_lion.jpg',
      },
      {
        nome: 'Girafa',
        urlImagen:
          'https://upload.wikimedia.org/wikipedia/commons/9/97/Namibie_Etosha_Girafe_02.jpg',
      },
      {
        nome: 'Gato',
        urlImagen:
          'https://upload.wikimedia.org/wikipedia/commons/b/b2/WhiteCat.jpg',
      },
    ];

    return (
      <View>
        <FlatList
          data={animais}
          renderItem={({item}) => <Animal />}
          keyExtractor={item => item.nome}
        />
      </View>
    );
  }
}
```

Mas agora temos um problema, o array `animais` está no componente `ListaAnimais`, como podemos torná-lo disponível no componente `Animal`?

Props

A maioria dos componentes pode ser personalizada quando eles são criados, com diferentes parâmetros. Esses parâmetros de criação são chamados de *props*.

As *props* permitem criar um único componente que é usado em muitos lugares diferentes do aplicativo, com propriedades ligeiramente diferentes em cada lugar. Basta se referir a `this.props` em sua função de renderização.

- `Animal.js`

```
export default class Animal extends Component {
  render() {
    return (
      <View>
        <Text style={styles.nomeAnimal}>{this.props.animal.nome}</Text>
        <Image
          source={{
            uri: this.props.animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />
      </View>
    );
  }
}
```

Para que este código funcione, devemos passar a propriedade `animal` a partir do componente superior, neste caso `ListaAnimais`:

- `ListaAnimais.js`

```
return (
  <View>
    <FlatList
      data={animais}
      renderItem={({item}) => <Animal animal={animal} />}
      keyExtractor={item => item.nome}
    />
  </View>
);
```

Mas perceba que estamos repetindo várias vezes o trecho `this.props`, para tornar nosso código menos verboso, podemos nos valer da sintaxe de desestruturação de objetos do ES6:

- `Animal.js`

```
export default class Animal extends Component {
  render() {
    const {animal} = this.props;
    return (
```

```

<View>
  <Text style={styles.nomeAnimal}>{animal.nome}</Text>
  <Image
    source={{
      uri: animal.urlImagen,
    }}
    style={styles.imagemAnimal}
  />
</View>
);
}
}

```

Um pouco de interatividade

Vamos permitir que o usuário possa "favoritar" um animal da listagem, para isso, vamos precisar de um ícone clicável.

Existem várias alternativas para incluir ícones em nossa aplicação, uma das mais populares é utilizar a biblioteca *NativeBase* (<https://nativebase.io/>) que, além de ícones, nos trará uma série de componentes amigáveis que poderemos utilizar futuramente.

A instalação do *NativeBase* está descrita em <https://docs.nativebase.io/docs/GetStarted.html> e pode ser resumida nos seguintes passos:

- Encerre a execução do script `react-native start` (`CTRL+C`) e em seguida:

```

> npm install native-base
> react-native link

```

- Desinstale a aplicação do emulador (clique e segure na aplicação e arraste para lixeira);
- Reinstale novamente a aplicação no emulador através dos seguintes comandos:

```

> react-native start
> react-native run-android

```

Agora vamos incluir um ícone de "favoritar" em nossa aplicação:

- `Animal.js`

```

import React, {Component} from 'react';
import {Dimensions, Image, StyleSheet, Text, View} from 'react-native';

// Novidade aqui
import {Icon} from 'native-base';

const {width} = Dimensions.get('screen');

```

```
export default class Animal extends Component {
  render() {
    const {animal} = this.props;
    return (
      <View>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        <Image
          source={{
            uri: animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />

        {/* Novidade aqui */}
        <Icon name="star" />
      </View>
    );
  }
}
```

Tornando o ícone clicável

Para tornar o ícone clicável, podemos utilizar o componente *TouchableOpacity* (<https://facebook.github.io/react-native/docs/touchableopacity>), que irá associar uma área da tela envolvendo componentes a uma função que será executada ao se clicar nesta área:

- `Animal.js`

```
export default class Animal extends Component {
  render() {
    const {animal} = this.props;
    return (
      <View>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        <Image
          source={{
            uri: animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />
        <TouchableOpacity onPress={console.warn('Favoritado!')}>
          <Icon name="star" />
        </TouchableOpacity>
      </View>
    );
  }
}
```

`console.warn` irá exibir uma caixa amarela de texto no rodapé da tela da aplicação, e é uma forma bem simples de conseguirmos uma depuração básica.

Mas o resultado obtido é diferente do que esperávamos, pois as mensagens de console são exibidas imediatamente após o carregamento da tela. Isso ocorre pois quando ocorre a renderização do componente, todas as expressões entre chaves são executadas. A propriedade `onPress` irá associar o resultado desta execução ao evento de `clique`, neste modo, o resultado da expressão deve ser uma referência a uma função.

Para corrigir isso, vamos alterar nossa função `onPress` :

```
export default class Animal extends Component {
  render() {
    const {animal} = this.props;
    return (
      <View>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        <Image
          source={{
            uri: animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />
        <TouchableOpacity onPress={() => console.warn('Favoritado!')}>
          <Icon name="star" />
        </TouchableOpacity>
      </View>
    );
  }
}
```

Atualizando o estado do componente

Para que um animal seja favoritado, o "estado" deverá ser atualizado (valor das variáveis).

Existem dois tipos de dados que controlam um componente: *props* e *state*. *props* são definidas pelo componente pai e são fixas durante todo o tempo de vida de um componente. Para os dados que vão mudar, temos que usar o *state*.

Em geral, você deve inicializar o estado, e chamar `setState` quando quiser alterá-lo.

Para começar vamos fazer um simples contador de cliques em nossa aplicação:

- `Animal.js`

```
export default class Animal extends Component {
  state = {
    contador: 0,
  };

  render() {
    const {animal} = this.props;
    return (
      <View>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        <Image
          source={{
            uri: animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />
        <TouchableOpacity
          onPress={() => this.setState({contador: this.state.contador + 1})}
        >
          <Icon name="star" />
        </TouchableOpacity>
        <Text>Foi favoritado {this.state.contador} vezes</Text>
      </View>
    );
  }
}
```

Desafio: Se substituirmos o código `this.state.contador + 1` por `this.state.contador++` ele continua funcionando como deveria? Porquê?

Constructor

Um padrão comum de codificação é inicializarmos o `state` diretamente no construtor de nosso componente:

```
export default class Animal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      contador: 0,
    };
  }

  // Código omitido
}
```

É muito importante chamarmos `super(props)` na primeira linha do construtor para garantir que a classe `Component` do React initialize corretamente as propriedades de nosso componente.

Outra melhoria que também podemos fazer é utilizarmos a desestruturação de objetos para o `state` de forma a simplificar o acesso a suas propriedades.

- `Animal.js`

```
render() {
  const {animal} = this.props;
  const {contador} = this.state;

  return (
    <View>
      <Text style={styles.nomeAnimal}>{animal.nome}</Text>
      <Image
        source={{
          uri: animal.urlImagen,
        }}
        style={styles.imagemAnimal}
      />

      {/* Novidades aqui */}
      <TouchableOpacity
        onPress={() => this.setState({contador: contador + 1})}
        >
        <Icon name="star" />
      </TouchableOpacity>
      <Text>Foi favoritado {contador} vezes</Text>

    </View>
  );
}
```

Desfavoritando

Tudo certo até agora, mas se paramos para pensar, um usuário só pode favoritar um animal uma única vez, ou seja, esta é uma propriedade booleana e o animal será favoritado e desfavoritado se o usuário pressionar mais de uma vez o ícone. Vamos fazer esta implementação ajustando também o texto abaixo do ícone pois agora temos um valor booleano:

- Animal.js

```
export default class Animal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      favoritado: false,
    };
  }

  render() {
    const {animal} = this.props;
    const {favoritado} = this.state;

    return (
      <View>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        <Image
          source={{
            uri: animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />
        <TouchableOpacity
          onPress={() => this.setState({favoritado: !favoritado})}>
          <Icon name="star" />
        </TouchableOpacity>
        <Text>Foi favoritado: {favoritado + ''}</Text>
      </View>
    );
  }
}
```

Desafio: Qual a finalidade de fazer `favoritado + ''`? O que acontece se não fizermos esta concatenação? Existe outra maneira de conseguir o mesmo efeito?

O padrão de UX comum quando um item está favoritado ou não é o ícone estar preenchido ou vazio, para isso, podemos utilizar uma técnica de renderização condicional baseada em variáveis de elementos:

- Animal.js

```
render() {
  const {animal} = this.props;
  const {favoritado} = this.state;
```

```
// Novidade aqui
let iconeFavoritado;

if (favoritado) {
  iconeFavoritado = <Icon name="star" />;
} else {
  iconeFavoritado = <Icon name="star-outline" />;
}

return (
  <View>
    <Text style={styles.nomeAnimal}>{animal.nome}</Text>
    <Image
      source={{
        uri: animal.urlImagen,
      }}
      style={styles.imagemAnimal}
    />

    {/* Novidade aqui */}
    <TouchableOpacity
      onPress={() => this.setState({favoritado: !favoritado})}>
      {iconeFavoritado}
    </TouchableOpacity>

    <Text>Foi favoritado: {favoritado + ''}</Text>
  </View>
);
}
```

Passando props para um state

Se paramos para analisar, `favoritado` é uma propriedade do animal, e não do componente como implementamos, deste modo, devemos atualizar o objeto `animal` que nos é passado via `props`. Mas aí há um problema, não podemos atualizar os `props` recebidos, esta é uma premissa do React, para valores mutáveis, devemos atualizar nosso objeto `state`. A solução é copiarmos a referência do objeto `animal` recebido via `props` para nosso `state` no momento da construção do objeto. Teremos que alterar também o funcionamento do `onPress` do componente `TouchableOpacity`, como nossa função de atualização de estado do animal ficará um pouco mais complicada, vamos isolar em um novo método de nosso componente:

- `Animal.js`

```
export default class Animal extends Component {
  // Novidade aqui
  constructor(props) {
    super(props);
    this.state = {
      animal: this.props.animal,
```

```

    };

}

// Novidade aqui
favoritar() {
  const {animal} = this.state;
  let novoAnimal = {...animal};
  novoAnimal.favoritado = !novoAnimal.favoritado;
  this.setState({animal: novoAnimal});
}

render() {
  const {animal} = this.state;

  let iconeFavoritado;

  if (animal.favoritado) {
    iconeFavoritado = <Icon name="star" />;
  } else {
    iconeFavoritado = <Icon name="star-outline" />;
  }

  return (
    <View>
      <Text style={styles.nomeAnimal}>{animal.nome}</Text>
      <Image
        source={{
          uri: animal.urlImagen,
        }}
        style={styles.imagemAnimal}
      />
      <TouchableOpacity onPress={this.favoritar}>
        {iconeFavoritado}
      </TouchableOpacity>
      <Text>Foi favoritado: {animal.favoritado + ''}</Text>
    </View>
  );
}
}

```

Métodos e Funções

Mas há algo errado, pois ao tentar favoritar o animal ocorre um erro: `undefined is not an object: evaluating 'this.state.animal'`. O motivo deste erro é que o escopo léxico da palavra `this` é alterado dinamicamente dependendo do momento da execução da função que a encapsula.

Veremos alguns meios de resolver este problema.

Primeira forma bind

O método `bind()` cria uma nova função que, quando chamada, tem sua palavra-chave `this` definida com o valor fornecido, com uma sequência determinada de argumentos precedendo quaisquer outros que sejam fornecidos quando a nova função é chamada. (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Function/bind)

- `Animal.js`

```
render() {
  const {animal} = this.state;

  let iconeFavoritado;

  if (animal.favoritado) {
    iconeFavoritado = <Icon name="star" />;
  } else {
    iconeFavoritado = <Icon name="star-outline" />;
  }

  return (
    <View>
      <Text style={styles.nomeAnimal}>{animal.nome}</Text>
      <Image
        source={{
          uri: animal.urlImagen,
        }}
        style={styles.imagemAnimal}
      />

      {/* Novidade aqui */}
      <TouchableOpacity onPress={this.favoritar.bind(this)}>
        {iconeFavoritado}
      </TouchableOpacity>

      <Text>Foi favoritado: {animal.favoritado + ''}</Text>
    </View>
  );
}
```

Esta solução não é recomendada, pois há um overhead de performance, você consegue visualizar o motivo? A cada chamada do método `render` uma nova função é criada, pressionando o motor JavaScript.

Para resolver isso, podemos realizar o `bind` uma única vez no construtor do componente:

- `Animal.js`

```
export default class Animal extends Component {
  constructor(props) {
    super(props);
```

```
// Novidade aqui
this.favoritar = this.favoritar.bind(this);

this.state = {
  animal: this.props.animal,
};

favoritar() {
  const {animal} = this.state;
  let novoAnimal = {...animal};
  novoAnimal.favoritado = !novoAnimal.favoritado;
  this.setState({animal: novoAnimal});
}

render() {
  const {animal} = this.state;

  let iconeFavoritado;

  if (animal.favoritado) {
    iconeFavoritado = <Icon name="star" />;
  } else {
    iconeFavoritado = <Icon name="star-outline" />;
  }

  return (
    <View>
      <Text style={styles.nomeAnimal}>{animal.nome}</Text>
      <Image
        source={{
          uri: animal.urlImagen,
        }}
        style={styles.imagemAnimal}
      />

      {/* Novidade aqui */}
      <TouchableOpacity onPress={this.favoritar}>
        {iconeFavoritado}
      </TouchableOpacity>

      <Text>Foi favoritado: {animal.favoritado + ''}</Text>
    </View>
  );
}
}
```

O ponto negativo desta estratégia é que precisamos "lembrar" de fazer o `bind` no construtor de todos nossos métodos que acessam `this`.

Segunda forma arrow functions

Uma expressão arrow function possui uma sintaxe mais curta quando comparada a uma expressão de função (*function expression*) e não tem seu próprio `this` (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Functions/Arrow_functions), pois utiliza a referência `this` de seu contexto de invocação:

- `Animal.js`

```
export default class Animal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animal: this.props.animal,
    };
  }

  favoritar() {
    const {animal} = this.state;
    let novoAnimal = {...animal};
    novoAnimal.favoritado = !novoAnimal.favoritado;
    this.setState({animal: novoAnimal});
  }

  render() {
    const {animal} = this.state;

    let iconeFavoritado;

    if (animal.favoritado) {
      iconeFavoritado = <Icon name="star" />;
    } else {
      iconeFavoritado = <Icon name="star-outline" />;
    }

    return (
      <View>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        <Image
          source={{
            uri: animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />

        {/* Novidade aqui */}
        <TouchableOpacity onPress={() => this.favoritar()}>
          {iconeFavoritado}
        </TouchableOpacity>
    );
  }
}
```

```
        <Text>Foi favoritado: {animal.favoritado + ''}</Text>
    </View>
);
}
}
```

Há ainda uma forma alternativa de utilizarmos `arrow functions` alterando a declaração da função para um atributo que referencia uma arrow function:

- `Animal.js`

```
export default class Animal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animal: this.props.animal,
    };
  }

  // Novidade aqui
  favoritar = () => {
    const {animal} = this.state;
    let novoAnimal = {...animal};
    novoAnimal.favoritado = !novoAnimal.favoritado;
    this.setState({animal: novoAnimal});
  };

  render() {
    const {animal} = this.state;

    let iconeFavoritado;

    if (animal.favoritado) {
      iconeFavoritado = <Icon name="star" />;
    } else {
      iconeFavoritado = <Icon name="star-outline" />;
    }

    return (
      <View>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        <Image
          source={{
            uri: animal.urlImagen,
          }}
          style={styles.imagemAnimal}
        />

        {/* Novidade aqui */}
        <TouchableOpacity onPress={this.favoritar}>

```

```

        {iconeFavoritado}
      </TouchableOpacity>
      <Text>Foi favoritado: {animal.favoritado + ''}</Text>
    </View>
  );
}
}

```

Todas as formas apresentadas são válidas, porém a única de fato não recomendada é realizar o `bind` na função `render` devido ao custo computacional mais elevado.

Durante o treinamento daremos preferência a última forma por simplicidade.

Desafio: Você conseguiria transformar a função `favoritar` em uma função de duas linhas?

Reduzindo a complexidade do método `render`

Uma boa prática de código é manter a complexidade do método `render` a menor possível, isso irá facilitar um eventual refatoramento do componente.

Agora que já aprendemos a trabalhar com métodos em nosso componente, podemos ter um método que retorne o ícone de favorito isolando o código da função `render`:

- `Animal.js`

```

export default class Animal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animal: this.props.animal,
    };
  }

  favoritar = () => {
    const {animal} = this.state;
    let novoAnimal = {...animal};
    novoAnimal.favoritado = !novoAnimal.favoritado;
    this.setState({animal: novoAnimal});
  };

  // Novidade aqui
  botaoFavorito(animal) {
    let iconeFavorito;

    if (animal.favoritado) {
      iconeFavorito = <Icon name="star" />;
    } else {
      iconeFavorito = <Icon name="star-outline" />;
    }
  }
}

```

```
return (
  <TouchableOpacity onPress={this.favoritar}>
    {iconeFavorito}
  </TouchableOpacity>
);
}

render() {
  const {animal} = this.state;

  return (
    <View>
      <Text style={styles.nomeAnimal}>{animal.nome}</Text>
      <Image
        source={{
          uri: animal.urlImagen,
        }}
        style={styles.imagemAnimal}
      />

      {/* Novidade aqui */}
      {this.botaoFavorito(animal)}

      <Text>Foi favoritado: {animal.favoritado + ''}</Text>
    </View>
  );
}
}
```

Você deve ter notado que optamos por passar o objeto `animal` como um parâmetro para a função, isso foi feito para que a função dependa menos do estado do componente e será importante quando fizermos mais refatorações.

Desafio: Não utilizamos a notação de `arrow function` na declaração do método `botaoFavorito` nem no seu acionamento e há uma referência a `this`, porquê não ocorre um erro?

Uma lista de usuários

Se analisarmos com mais profundidade nossa estrutura de dados veremos que há uma falha pois nosso sistema será multiusuário, e como podemos saber se foi o usuário atual que de fato favoritou a foto? Para corrigir isso, devemos transformar a propriedade "favoritado" em uma lista de nomes de usuários que favoritaram a foto.

Esta alteração é bem impactante, a primeira alteração é ajustar nosso componente `ListaAnimais` para que os animais tenham uma nova propriedade e também haja o estado `usuarioLogado` que representa o usuário atualmente logado no sistema:

- `ListaAnimais.js`

```
export default class ListaAnimais extends Component {
  constructor(props) {
    super(props);

    // Novidades aqui
    this.state = {
      animais: [
        {
          nome: 'Leão',
          urlImagen:
            'https://upload.wikimedia.org/wikipedia/commons/4/40/Just_one_lion.jpg'

          favoritoUsuarios: [],
        },
        {
          nome: 'Girafa',
          urlImagen:
            'https://upload.wikimedia.org/wikipedia/commons/9/97/Namibie_Etosha_Girafe_02.jpg',
          favoritoUsuarios: ['maria'],
        },
        {
          nome: 'Gato',
          urlImagen:
            'https://upload.wikimedia.org/wikipedia/commons/b/b2/WhiteCat.jpg',
          favoritoUsuarios: ['maria', 'paulo'],
        },
      ],
      usuarioLogado: 'jose',
    };
  }

  render() {
    const {animais, usuarioLogado} = this.state;
    return (
      <View>
```

```

    /* Novidades aqui */
    <FlatList
      data={animais}
      renderItem={({item}) => (
        <Animal animal={item} usuarioLogado={usuarioLogado} />
      )}
      keyExtractor={item => item.nome}
    />
  </View>
);
}
}

```

Agora ajustaremos o componente `Animal` para as novas `props`. A lógica de identificar se o usuário atual já favoritou ou não um animal está mais complicada, por isso será isolada em uma função `isFavoritado`. Também teremos dois métodos distintos, um irá `favoritar` e outro irá `desfavoritar`:

- `Animal.js`

```

export default class Animal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animal: this.props.animal,
    };
  }

  isFavoritado(animal, usuarioLogado) {
    return !!animal.favoritoUsuarios.find(usuario => usuario === usuarioLogado);
  }

  favoritar = (animal, usuarioLogado) => {
    let novoAnimal = {...animal};

    novoAnimal.favoritoUsuarios = [
      ...novoAnimal.favoritoUsuarios,
      usuarioLogado,
    ];

    this.setState({animal: novoAnimal});
  };

  desfavoritar = (animal, usuarioLogado) => {
    let novoAnimal = {...animal};

    novoAnimal.favoritoUsuarios = novoAnimal.favoritoUsuarios.filter(
      usuario => usuario !== usuarioLogado,
    );

    this.setState({animal: novoAnimal});
  };
}

```

```

};

botaoFavorito(animal, usuarioLogado) {
  let favoritado = this.isFavoritado(animal, usuarioLogado);

  if (favoritado) {
    return (
      <TouchableOpacity
        onPress={() => this.desfavoritar(animal, usuarioLogado)}
      >
        <Icon name="star" />
      </TouchableOpacity>
    );
  } else {
    return (
      <TouchableOpacity onPress={() => this.favoritar(animal, usuarioLogado)}>
        <Icon name="star-outline" />
      </TouchableOpacity>
    );
  }
}

render() {
  const {animal} = this.state;
  const {usuarioLogado} = this.props;

  return (
    <View>
      <Text style={styles.nomeAnimal}>{animal.nome}</Text>
      <Image
        source={{
          uri: animal.urlImagen,
        }}
        style={styles.imagemAnimal}
      />
      {this.botaoFavorito(animal, usuarioLogado)}
      <Text>
        Foi favoritado: {this.isFavoritado(animal, usuarioLogado) + ''}
      </Text>
    </View>
  );
}
}

```

Vamos deixar o texto do rodapé da imagem um pouco mais interessante informando quantas vezes o animal já foi favoritado, caso ele não tenha sido favoritado nenhuma vez, devemos exibir "Este animal ainda não foi favoritado":

- Animal.js

```
render() {
```

```
const {animal} = this.state;
const {usuarioLogado} = this.props;

return (
  <View>
    <Text style={styles.nomeAnimal}>{animal.nome}</Text>
    <Image
      source={{
        uri: animal.urlImagen,
      }}
      style={styles.imagemAnimal}
    />
    {this.botaoFavorito(animal, usuarioLogado)}

    {/* Novidade aqui */}
    <Text>
      Este animal
      {animal.favoritoUsuarios.length > 0
        ? ` já foi favoritado por ${animal.favoritoUsuarios.length} usuário(s)`
        : ' ainda não foi favoritado'}
    </Text>

    </View>
);
}
```

Desafio 1: Será que podemos substituir `animal.favoritoUsuarios.length > 0` por `animal.favoritoUsuarios.length`? Qual o motivo?

Desafio 2: Você consegue ajustar o código que sensibilize corretamente a exibição da palavra usuário em singular/plural? Ex: `Este animal já foi favoritado por 1 usuário` e `Este animal já foi favoritado por 2 usuários`.

Melhorando o Layout

Nossa aplicação já está lidando bem com as `props` e o `state`, mas algo que falta melhorarmos é o visual. É o que faremos agora com o auxílio dos componentes da biblioteca *NativeBase*.

Anatomia básica da tela

O componente inicial de um layout utilizando o `NativeBase` é o container, dentro do container podemos ter mais três componentes, um *header*, um *content* e um *footer*. Inicialmente definiremos apenas um cabeçalho básico e o conteúdo:

- `ListaAnimais`

```
import {Container, Content, Header, Title} from 'native-base';
import React, {Component} from 'react';
import {FlatList, StyleSheet} from 'react-native';
import Animal from './Animal';

export default class ListaAnimais extends Component {
  //...

  render() {
    const {animais, usuarioLogado} = this.state;
    return (
      <Container>
        <Header style={styles.header}>
          <Title>Controle de Animais</Title>
        </Header>
        <Content padder>
          <FlatList
            data={animais}
            renderItem={({item}) => (
              <Animal animal={item} usuarioLogado={usuarioLogado} />
            )}
            keyExtractor={item => item.nome}
          />
        </Content>
      </Container>
    );
  }

  const styles = StyleSheet.create({
    header: {height: 30},
  });
}
```

Já que estamos melhorando o componente de `ListaAnimais`, podemos refatorá-lo para consumir os dados fixos a partir de um json de dados que será fornecido. Para isso, crie um arquivo `data.json` na raiz do projeto:

- `data.json`

```
{  
  "animais": [  
    {  
      "nome": "Leão",  
      "urlImagem": "https://upload.wikimedia.org/wikipedia/commons/4/40/Just_one_lion.jpg",  
      "favoritoUsuarios": []  
    },  
    {  
      "nome": "Girafa",  
      "urlImagem": "https://upload.wikimedia.org/wikipedia/commons/9/97/Namibie_Etosha_Girafe_02.jpg",  
      "favoritoUsuarios": ["maria"]  
    },  
    {  
      "nome": "Gato",  
      "urlImagem": "https://upload.wikimedia.org/wikipedia/commons/b/b2/WhiteCat.jpg",  
      "favoritoUsuarios": ["maria", "paulo"]  
    }  
  "usuarioLogado": "jose"  
}
```

Agora, vamos ajustar o componente `ListaAnimais` para consumir este json:

- `ListaAnimais.js`

```
import {Container, Content, Header, Title} from 'native-base';  
import React, {Component} from 'react';  
import {FlatList, StyleSheet} from 'react-native';  
import Animal from './Animal';  
import {  

```

```
    };
}

// ...

}

const styles = StyleSheet.create({
  header: {height: 30},
});
```

Perceba como nosso componente ficou bem mais limpo agora!

Melhorando a exibição do Animal

Passaremos agora para o componente Animal, para um layout melhor, utilizaremos o componente *Card* da biblioteca NativeBase: (<https://docs.nativebase.io/Components.html#card-def-headref>)

- `Animal.js`

```
import {Body, Card, CardItem, Icon} from 'native-base';
import React, {Component} from 'react';
import {
  Dimensions,
  Image,
  StyleSheet,
  Text,
  TouchableOpacity,
} from 'react-native';

const {width} = Dimensions.get('screen');

export default class Animal extends Component {
  //...

  render() {
    const {animal} = this.state;
    const {usuarioLogado} = this.props;

    return (
      <Card>
        <CardItem header bordered>
          <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        </CardItem>
        <CardItem bordered>
          <Body style={styles.imageContainer}>
            <Image
              source={{
                uri: animal.urlImagen,
              }}>
```

```
        style={styles.imagemAnimal}
      />
    </Body>
  </CardItem>
<CardItem footer bordered>
  {this.botaoFavorito(animal, usuarioLogado)}
  <Text>
    Este animal
    {animal.favoritoUsuarios.length > 0
     ? ` já foi favoritado por ${animal.favoritoUsuarios.length} usuário(s)`
     : ' ainda não foi favoritado'}
  </Text>
</CardItem>
</Card>
);
}
}

const styles = StyleSheet.create({
  nomeAnimal: {fontSize: 18, fontWeight: 'bold'},
  imagemAnimal: {width: width * 0.7, height: width * 0.7},
  imageContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
});
```

Pronto, agora já temos um visual mais apresentável!

Um componente para favoritar

Uma das principais características do React é o foco na componentização. Até o momento, nossa aplicação possui apenas dois componentes, e o componente `Animal` possui uma lógica muito grande relacionada a funcionalidade de favoritar. Vamos melhorar isso, criando um componente exclusivo para o Botão Favoritar, que receberá toda esta lógica:

- `src\components\BotaoFavoritar.js`

```
import {Icon} from 'native-base';
import React, {Component} from 'react';
import {TouchableOpacity} from 'react-native';

export default class BotaoFavoritar extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animal: this.props.animal,
      usuarioLogado: this.props.usuarioLogado,
    };
  }

  isFavoritado(animal, usuarioLogado) {
    return !!animal.favoritoUsuarios.find(usuario => usuario === usuarioLogado);
  }

  favoritar = (animal, usuarioLogado) => {
    let novoAnimal = {...animal};

    novoAnimal.favoritoUsuarios = [
      ...novoAnimal.favoritoUsuarios,
      usuarioLogado,
    ];

    this.setState({animal: novoAnimal});
  };

  desfavoritar = (animal, usuarioLogado) => {
    let novoAnimal = {...animal};

    novoAnimal.favoritoUsuarios = novoAnimal.favoritoUsuarios.filter(
      usuario => usuario !== usuarioLogado,
    );

    this.setState({animal: novoAnimal});
  };

  render() {
```

```
const {animal, usuarioLogado} = this.state;

let favoritado = this.isFavoritado(animal, usuarioLogado);

if (favoritado) {
  return (
    <TouchableOpacity
      onPress={() => this.desfavoritar(animal, usuarioLogado)}>
      <Icon name="star" />
    </TouchableOpacity>
  );
} else {
  return (
    <TouchableOpacity onPress={() => this.favoritar(animal, usuarioLogado)}>
      <Icon name="star-outline" />
    </TouchableOpacity>
  );
}
}
```

Com este botão criado, podemos otimizar o componente `Animal` para que passe a utilizar o componente `BotaoFavoritar`:

- `Animal.js`

```
export default class Animal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animal: this.props.animal,
    };
  }

  render() {
    const {animal} = this.state;
    const {usuarioLogado} = this.props;

    return (
      <Card>
        <CardItem header bordered>
          <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        </CardItem>
        <CardItem bordered>
          <Body style={styles.imageContainer}>
            <Image
              source={{
                uri: animal.urlImagen,
              }}
              style={styles.imageAnimal}
            </Image>
          </Body>
        </CardItem>
      </Card>
    );
  }
}
```

```

        />
      </Body>
    </CardItem>
    <CardItem footer bordered>
      <BotaoFavoritar animal={animal} usuarioLogado={usuarioLogado} />
      <Text>
        Este animal
        {animal.favoritoUsuarios.length > 0
         ? ` já foi favoritado por ${animal.favoritoUsuarios.length} usuário(s)`
         : ' ainda não foi favoritado'}
      </Text>
    </CardItem>
  </Card>
);
}
}
}

```

Agora o componente `Animal` ficou muito mais simples. A aplicação deve estar funcionando, porém, quando se favorita um animal a contagem de favoritos não está mais sendo atualizada, o que pode ter ocorrido?

O problema é que estamos "espalhando" o estado de nossa aplicação através de diversos componentes. Os objetos presentes na lista de animais que estão no componente `ListaAnimais` não são os mesmos objetos presentes em `Animal` e `BotaoFavoritar`.

Para resolver este problema existem diversas opções, a primeira é utilizar uma estratégia chamada de `Container Components` e `Presentational Components`. Esta estratégia baseia-se em manter o estado em um componente pai, os componentes filhos, quando desejam atualizar seu estado, devem fazer isso acionando eventos do componente pai.

Vamos começar a implantar esta estratégia entre o componente `Animal` e o `BotaoFavoritar`. O primeiro passo é manter o estado as funções que alteram o estado no componente `Animal`, passando somente referências ao `BotaoFavoritar`:

- `Animal.js`

```

import {Body, Card, CardItem} from 'native-base';
import React, {Component} from 'react';
import {Dimensions, Image, StyleSheet, Text} from 'react-native';
import BotaoFavoritar from './BotaoFavoritar';

const {width} = Dimensions.get('screen');

export default class Animal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animal: this.props.animal,
    };
  }
}

```

```
isFavoritado(animal, usuarioLogado) {
  return !animal.favoritoUsuarios.find(usuario => usuario === usuarioLogado);
}

favoritar = () => {
  const {animal} = this.state;
  const {usuarioLogado} = this.props;

  let novoAnimal = {...animal};

  novoAnimal.favoritoUsuarios = [
    ...novoAnimal.favoritoUsuarios,
    usuarioLogado,
  ];

  this.setState({animal: novoAnimal});
};

desfavoritar = () => {
  const {animal} = this.state;
  const {usuarioLogado} = this.props;

  let novoAnimal = {...animal};

  novoAnimal.favoritoUsuarios = novoAnimal.favoritoUsuarios.filter(
    usuario => usuario !== usuarioLogado,
  );

  this.setState({animal: novoAnimal});
};

render() {
  const {animal} = this.state;
  const {usuarioLogado} = this.props;

  return (
    <Card>
      <CardItem header bordered>
        <Text style={styles.nomeAnimal}>{animal.nome}</Text>
      </CardItem>
      <CardItem bordered>
        <Body style={styles.imageContainer}>
          <Image
            source={{
              uri: animal.urlImagem,
            }}
            style={styles.imagemAnimal}
          />
        </Body>
      </CardItem>
    
```

```
<CardItem footer bordered>
  <BotaoFavoritar
    favoritado={this.isFavoritado(animal, usuarioLogado)}
    favoritarCallback={this.favoritar}
    desfavoritarCallback={this.desfavoritar}>
  />
  <Text>
    Este animal
    {animal.favoritoUsuarios.length > 0
      ? ` já foi favoritado por ${animal.favoritoUsuarios.length} usuário(s)`
      : ' ainda não foi favoritado'}
  </Text>
</CardItem>
</Card>
);
}
}

const styles = StyleSheet.create({
  nomeAnimal: {fontSize: 18, fontWeight: 'bold'},
  imagemAnimal: {width: width * 0.7, height: width * 0.7},
  imageContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
});
```

Agora atualizamos o `BotaoFavoritar` para que não tenha mais um estado próprio e acione as funções que lhe foram passadas como callback:

- `BotaoFavoritar.js`

```
import {Icon} from 'native-base';
import React, {Component} from 'react';
import {TouchableOpacity} from 'react-native';

export default class BotaoFavoritar extends Component {
  render() {
    const {favoritado, favoritarCallback, desfavoritarCallback} = this.props;

    if (favoritado) {
      return (
        <TouchableOpacity onPress={desfavoritarCallback}>
          <Icon name="star" />
        </TouchableOpacity>
      );
    } else {
      return (
```

```
<TouchableOpacity onPress={favoritarCallback}>
  <Icon name="star-outline" />
</TouchableOpacity>
);
}
}
}
```

Nossa aplicação deve estar funcionando normalmente, transformamos com sucesso o `BotaoFavoritar` em um Presentational Component e o `Animal` em um Container Component.

Centralizando o Estado

Podemos melhorar ainda mais a aplicação deixando também o componente `Animal` sem um estado próprio, e mantendo o componente `ListaAnimais` como nosso único Container Component e única fonte da verdade.

Primeiramente vamos adicionar uma nova propriedade aos nossos dados, um `_id` para cada animal, essa propriedade facilitará a identificação de cada um dos animais:

- `data.json`

```
{
  "animais": [
    {
      "_id": "1",
      "nome": "Leão",
      "urlImagem": "https://upload.wikimedia.org/wikipedia/commons/4/40/Just_one_lion.jpg",
      "favoritoUsuarios": []
    },
    {
      "_id": "2",
      "nome": "Girafa",
      "urlImagem": "https://upload.wikimedia.org/wikipedia/commons/9/97/Namibia_Etosha_Girafe_02.jpg",
      "favoritoUsuarios": ["maria"]
    },
    {
      "_id": "3",
      "nome": "Gato",
      "urlImagem": "https://upload.wikimedia.org/wikipedia/commons/b/b2/WhiteCat.jpg",
      "favoritoUsuarios": ["maria", "paulo"]
    }
  ],
  "usuarioLogado": "jose"
}
```

Agora vamos ajustar o componente `Animal` para ser um *Presentational Component*, a principal alteração é que ele agora irá receber as funções `favoritarCallback` e `desfavoritarCallback` como parâmetros e deverá repassá-las ao componente `BotaoFavoritar`:

- `Animal.js`

```
const {width} = Dimensions.get('screen');

export default class Animal extends Component {
  isFavoritado(animal, usuarioLogado) {
    return !!animal.favoritoUsuarios.find(usuario => usuario === usuarioLogado);
```

```

        }

      render() {
        const {
          animal,
          usuarioLogado,
          favoritarCallback,
          desfavoritarCallback,
        } = this.props;

        return (
          <Card>
            <CardItem header bordered>
              <Text style={styles.nomeAnimal}>{animal.nome}</Text>
            </CardItem>
            <CardItem bordered>
              <Body style={styles.imageContainer}>
                <Image
                  source={{
                    uri: animal.urlImagem,
                  }}
                  style={styles.imagemAnimal}
                />
              </Body>
            </CardItem>
            <CardItem footer bordered>
              <BotaoFavoritar
                favoritado={this.isFavoritado(animal, usuarioLogado)}
                favoritarCallback={() => favoritarCallback(animal)}
                desfavoritarCallback={() => desfavoritarCallback(animal)}
              />
              <Text>
                Este animal
                {animal.favoritoUsuarios.length > 0
                  ? ` já foi favoritado por ${animal.favoritoUsuarios.length} usuário(s)`
                  : ' ainda não foi favoritado'}
              </Text>
            </CardItem>
          </Card>
        );
      }
    }
  }
}

```

Toda a lógica e estado agora ficará concentrada no componente `ListaAnimais` que deverá repassar referências das funções aos componentes inferiores:

- `ListaAnimais.js`

```
export default class ListaAnimais extends Component {
```

```
constructor(props) {
  super(props);

  this.state = {
    animais: animaisData,
    usuarioLogado: usuarioLogadoData,
  };
}

favoritar = animal => {
  const {usuarioLogado} = this.state;

  let novoAnimal = {...animal};

  novoAnimal.favoritoUsuarios = [
    ...novoAnimal.favoritoUsuarios,
    usuarioLogado,
  ];

  const novosAnimais = this.state.animais.map(a =>
    a._id === novoAnimal._id ? novoAnimal : a,
  );

  this.setState({animais: novosAnimais});
};

desfavoritar = animal => {
  const {usuarioLogado} = this.state;

  let novoAnimal = {...animal};

  novoAnimal.favoritoUsuarios = novoAnimal.favoritoUsuarios.filter(
    usuario => usuario !== usuarioLogado,
  );

  const novosAnimais = this.state.animais.map(a =>
    a._id === novoAnimal._id ? novoAnimal : a,
  );

  this.setState({animais: novosAnimais});
};

render() {
  const {animais, usuarioLogado} = this.state;
  return (
    <Container>
      <Header style={styles.header}>
        <Title>Controle de Animais</Title>
      </Header>
      <Content padder>
        <FlatList
```

```

        data={animais}
        renderItem={({item}) => (
          <Animal
            animal={item}
            usuarioLogado={usuarioLogado}
            favoritarCallback={this.favoritar}
            desfavoritarCallback={this.desfavoritar}
          />
        )}
        keyExtractor={item => item.nome}
      />
    </Content>
  </Container>
);
}
}

```

Desafio: Você consegue centralizar o código que atualiza os animais em uma única função? Deste modo teríamos algo como:

```

favoritar = animal => {
  const {usuarioLogado} = this.state;

  let novoAnimal = {...animal};

  novoAnimal.favoritoUsuarios = [
    ...novoAnimal.favoritoUsuarios,
    usuarioLogado,
  ];

  this.atualizarAnimal(novoAnimal);
};

desfavoritar = animal => {
  const {usuarioLogado} = this.state;

  let novoAnimal = {...animal};

  novoAnimal.favoritoUsuarios = novoAnimal.favoritoUsuarios.filter(
    usuario => usuario !== usuarioLogado,
  );

  this.atualizarAnimal(novoAnimal);
};

atualizarAnimal(animal) {
  // TODO
}

```

Com isso conseguimos transformar `ListaAnimais` em nosso único Container Component, e os demais componentes são Presentational Components.

Esta estratégia funciona para aplicações não muito complexas, mas se você parar para analisar, caso haja um número grande de componentes e funcionalidades, há uma tendência do nosso Container Component ficar muito grande e ter que passar um número elevado de funções para os componentes filhos, além do quê, qualquer mudança na hieranquia da árvore de componentes necessitará de ajustes nas funções que são transitadas.

Redux

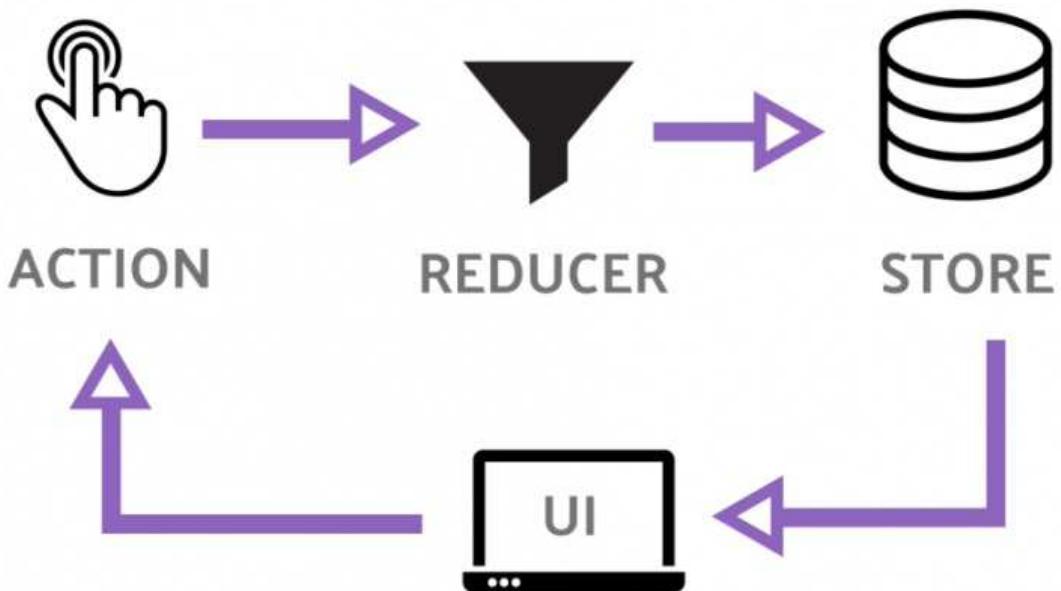
Quando uma aplicação torna-se muito complexa, o gerenciamento de estado pode se tornar algo extremamente problemático utilizando-se estratégias convencionais, é para resolver este problema que foi criada uma biblioteca chamada Redux.

O Redux é um contêiner de estado previsível para aplicativos JavaScript.

Ele ajuda escrever aplicativos que se comportam de maneira consistente, executados em diferentes ambientes (cliente, servidor e nativo) e são fáceis de testar. Além disso, proporciona uma ótima experiência de desenvolvedor, como edição de código ao vivo combinada com um depurador de viagem no tempo.

O Redux pode ser utilizado junto com o React ou com qualquer outra biblioteca de view. E é uma biblioteca pequena (2kB, incluindo dependências).

Conceitos centrais



Imagine que o estado de um aplicativo é descrito como um objeto simples. Por exemplo, o estado de um aplicativo todo pode ter esta aparência:

```
{
  todos: [
    {
      text: 'Eat food',
      completed: true
    },
    {
      text: 'Exercise',
      completed: false
    }
  ],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

```
}
```

Este objeto é como um "modelo", exceto que não há "setters". Isso acontece para que diferentes partes do código não alterem o estado arbitrariamente, causando bugs difíceis de reproduzir e corrigir.

Para mudar algo no estado, você precisa despachar uma ação. Uma ação é um objeto JavaScript simples que descreve o que aconteceu. Aqui estão algumas ações de exemplo:

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

Obrigando que todas as alterações sejam descritas como uma ação nos permite ter uma compreensão clara do que está acontecendo no aplicativo. Se algo mudou, sabemos porque mudou. Ações são como rastros do que aconteceu. Finalmente, para amarrar estados e ações juntos, escrevemos uma função chamada "reducer". É apenas uma função que toma o estado e a ação como argumentos e retorna o próximo estado do aplicativo. Seria difícil escrever uma função desse tipo para um aplicativo grande, por isso escrevemos funções menores gerenciando partes do estado:

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  if (action.type === 'SET_VISIBILITY_FILTER') {
    return action.filter
  } else {
    return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map(
        (todo, index) =>
          action.index === index
            ? { text: todo.text, completed: !todo.completed }
            : todo
      )
    default:
      return state
  }
}
```

E escrevemos outro redutor que gerencia o estado completo do nosso aplicativo chamando esses dois redutores para as chaves de estado correspondentes:

```
function todoApp(state = {}, action) {
```

```

return {
  todos: todos(state.todos, action),
  visibilityFilter: visibilityFilter(state.visibilityFilter, action),
};
}

```

Esta é basicamente a ideia do Redux. Observe que não usamos nenhuma API do Redux. Ele vem com alguns utilitários para facilitar esse padrão, mas a ideia principal é que se descreva como o estado é atualizado ao longo do tempo em resposta a objetos de ação, e 90% do código que você escreve é simplesmente JavaScript, sem uso do Redux em si, suas APIs ou qualquer "mágica".

Os três princípios

Única fonte de verdade

"O estado de todo o aplicativo é armazenado em uma árvore de objetos em um único local de armazenamento."

Isso facilita a criação de aplicativos universais, já que o estado do servidor pode ser serializado e implantado no cliente sem nenhum esforço extra de codificação. Uma única árvore de estado também facilita a depuração ou inspeção de um aplicativo; Ele também permite que você persista o estado do seu aplicativo em desenvolvimento, para um ciclo de desenvolvimento mais rápido. Algumas funcionalidades que têm sido tradicionalmente difíceis de implementar - Desfazer/Refazer, por exemplo - podem subitamente tornar-se triviais de implementar, se todo o seu estado estiver armazenado em uma única árvore.

```

{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}

```

Estado é somente leitura

"A única maneira de mudar o estado é emitir uma ação, um objeto descrevendo o que aconteceu."

Isso garante que nem as views nem os callbacks da rede jamais serão gravados diretamente no estado. Em vez disso, eles expressam a intenção de transformar o estado. Como todas as mudanças são centralizadas e acontecem uma a uma em uma ordem estrita, não há condições corrida a serem observadas. Como as ações são

apenas objetos simples, elas podem ser registradas, serializadas, armazenadas e, posteriormente, reproduzidas para fins de depuração ou teste.

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
})
```

As alterações são feitas com funções puras

"Para especificar como a árvore de estados é transformada por ações, você escreve redutores puros."

Redutores são apenas funções puras que tomam o estado anterior e uma ação, e retornam o próximo estado. Lembre-se de retornar novos objetos de estado, em vez de alterar o estado anterior. Você pode começar com um único redutor e, à medida que seu aplicativo cresce, divida-o em redutores menores que gerenciam partes específicas da árvore de estados. Como os redutores são apenas funções, você pode controlar a ordem na qual eles são chamados, passar dados adicionais ou até mesmo fazer reduções reutilizáveis para tarefas comuns, como paginação.

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case 'COMPLETE_TODO':
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
  }
}
```

```

        })
    }
    return todo
})
default:
    return state
}
}

import { combineReducers, createStore } from 'redux'
const reducer = combineReducers({ visibilityFilter, todos })
const store = createStore(reducer)

```

Passando o controle de estado para o Redux

Para iniciarmos com a utilização do Redux, primeiro, precisamos instalar as bibliotecas necessárias:

```
> npm install redux react-redux
```

Agora vamos criar nossos primeiros *reducers*.

Um reducer irá controlar a listagem de animais:

- `src\reducers\animais.js`

```

import {animais} from '../../../../../data.json';

const initialState = animais;

export default function animaisReducer(state = initialState, action) {
    switch (action.type) {
        default:
            return state;
    }
}

```

O outro reducer irá controlar o usuário logado:

- `src\reducers\usuarioLogado.js`

```

import {usuarioLogado} from '../../../../../data.json';

const initialState = usuarioLogado;

export default function usuarioLogadoReducer(state = initialState, action) {
    switch (action.type) {
        default:
            return state;
    }
}

```

```

    }
}

```

Agora criaremos um arquivo `index.js` na pasta `reducers` que terá a função de combinar os vários `reducers` que criamos:

- `src\reducers\index.js`

```

import {combineReducers} from 'redux';
import animaisReducer from './animais';
import usuarioLogadoReducer from './usuarioLogado';

const rootReducer = combineReducers({
  animais: animaisReducer,
  usuarioLogado: usuarioLogadoReducer,
});

export default rootReducer;

```

Em seguida, criaremos um arquivo JavaScript que tem a função criar uma `store` a partir dos `reducers`.

A `store` contém a árvore de estados completa do seu aplicativo. Só deve haver uma única loja no seu aplicativo.

- `src\configureStore.js`

```

import {createStore} from 'redux';
import rootReducer from './reducers';

export default function configureStore() {
  let store = createStore(rootReducer);
  return store;
}

```

Um passo adicional que faremos é a divisão do componente `ListaAnimais` em dois, teremos um componente de ordem superior que será responsável pelas configurações gerais da aplicação, esta divisão será útil mais a frente quando começarmos a realizar navegações em nossa aplicação.

- `src\App.js`

```

import {Container, Content, Header, Title} from 'native-base';
import React, {Component} from 'react';
import {StyleSheet} from 'react-native';
import {Provider} from 'react-redux';
import ListaAnimais from './components/ListaAnimais';
import configureStore from './configureStore';

const store = configureStore();

export default class App extends Component {

```

```

    render() {
      return (
        <Provider store={store}>
          <Container>
            <Header style={styles.header}>
              <Title>Controle de Animais</Title>
            </Header>
            <Content padder>
              <ListaAnimais />
            </Content>
          </Container>
        </Provider>
      );
    }
  }

const styles = StyleSheet.create({
  header: {height: 30},
  separator: {
    height: 1,
    backgroundColor: '#CED0CE',
    marginBottom: 10,
  },
});

```

O último passo é a atualização de `ListaAnimais` para que utilize o Redux. Primeiramente mapearemos os estados através da função `mapStateToProps`, que mapeará um estado do Redux para uma prop homônima do componente. Sempre que o estado do Redux for atualizado, a prop também será e o componente terá sua view atualizada:

- `ListaAnimais`

```

import React, {Component} from 'react';
import {FlatList} from 'react-native';
import {connect} from 'react-redux';
import Animal from './Animal';

class ListaAnimais extends Component {
  favoritar = animal => {
    const {usuarioLogado} = this.state;

    let novoAnimal = {...animal};

    novoAnimal.favoritoUsuarios = [
      ...novoAnimal.favoritoUsuarios,
      usuarioLogado,
    ];

    this.atualizarAnimal(novoAnimal);
  };
}

```

```

desfavoritar = animal => {
  const {usuarioLogado} = this.state;

  let novoAnimal = {...animal};

  novoAnimal.favoritoUsuarios = novoAnimal.favoritoUsuarios.filter(
    usuario => usuario !== usuarioLogado,
  );

  this.atualizarAnimal(novoAnimal);
};

atualizarAnimal = novoAnimal => {
  const novosAnimais = this.state.animais.map(a =>
    a._id === novoAnimal._id ? novoAnimal : a,
  );

  this.setState({animais: novosAnimais});
};

render() {
  const {animais, usuarioLogado} = this.props;
  return (
    <FlatList
      data={animais}
      renderItem={({item}) => (
        <Animal
          animal={item}
          usuarioLogado={usuarioLogado}
          favoritarCallback={this.favoritar}
          desfavoritarCallback={this.desfavoritar}
        />
      )}
      keyExtractor={item => item.nome}
    />
  );
}

const mapStateToProps = state => {
  return {
    animais: state.animais,
    usuarioLogado: state.usuarioLogado,
  };
};

const mapDispatchToProps = {};

export default connect(
  mapStateToProps,

```

```
    mapDispatchToProps,
})(ListaAnimais);
```

No entanto, agora se você tentar favoritar um animal ocorrerá um erro. Temos que ajustar as funcionalidades de nossa aplicação para que utilizem a lógica do Redux.

Em primeiro lugar, criaremos um arquivo chamado `constants.js` que conterá as constantes relativas às ações:

- `src\constants.js`

```
const FAVORITAR = 'FAVORITAR';
const DESFAVORITAR = 'DESFAVORITAR';

export {FAVORITAR, DESFAVORITAR};
```

Agora, vamos criar um arquivo `actions.js` que irá criar as ações do Redux a partir de funções simples:

- `src\actions.js`

```
import {FAVORITAR, DESFAVORITAR} from './constants';

export function favoritar(animal, usuario) {
  return {
    type: FAVORITAR,
    data: {
      animal,
      usuario,
    },
  };
}

export function desfavoritar(animal, usuario) {
  return {
    type: DESFAVORITAR,
    data: {
      animal,
      usuario,
    },
  };
}
```

Um outro passo é atualizar o reducer de animais, para suportar estas novas ações:

- `src\reducers\animais.js`

```
import {animais} from '../../data.json';
import {FAVORITAR, DESFAVORITAR} from '../constants.js';
```

```

const initialState = animais;

function atualizaAnimal(listaAnimais, animal) {
  return listaAnimais.map(a => (a._id === animal._id ? animal : a));
}

export default function animaisReducer(state = initialState, action) {
  switch (action.type) {
    case FAVORITAR: {
      const {animal, usuario} = action.data;
      const novoAnimal = {...animal};
      novoAnimal.favoritoUsuarios = [...novoAnimal.favoritoUsuarios, usuario];
      return atualizaAnimal(state, novoAnimal);
    }

    case DESFAVORITAR: {
      const {animal, usuario} = action.data;
      const novoAnimal = {...animal};
      novoAnimal.favoritoUsuarios = novoAnimal.favoritoUsuarios.filter(
        u => u !== usuario,
      );
      return atualizaAnimal(state, novoAnimal);
    }

    default:
      return state;
  }
}

```

Simplificaremos agora o componente de `ListaAnimais` pois não será mais necessário passar tantas props para o componente `Animal` :

- `ListaAnimais`

```

import React, {Component} from 'react';
import {FlatList} from 'react-native';
import {connect} from 'react-redux';
import Animal from './Animal';

class ListaAnimais extends Component {
  render() {
    const {animais} = this.props;
    return (
      <FlatList
        data={animais}
        renderItem={({item}) => <Animal animal={item} />}
        keyExtractor={item => item.nome}
      />
    );
  }
}

```

```

}

const mapStateToProps = state => {
  return {
    animais: state.animais,
    usuarioLogado: state.usuarioLogado,
  };
};

const mapDispatchToProps = {};

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(ListaAnimais);

```

O componente `Animal` será todo refatorado para disparar as ações do Redux:

- `Animal.js`

```

import {Body, Card, CardItem} from 'native-base';
import React, {Component} from 'react';
import {Dimensions, Image, StyleSheet, Text} from 'react-native';
import {connect} from 'react-redux';
import BotaoFavoritar from './BotaoFavoritar';
import {favoritar, desfavoritar} from '../actions';
import {bindActionCreators} from 'redux';

const {width} = Dimensions.get('screen');

class Animal extends Component {
  isFavoritado(animal, usuarioLogado) {
    return !!animal.favoritoUsuarios.find(usuario => usuario === usuarioLogado);
  }

  render() {
    const {animal} = this.props;

    return (
      <Card>
        <CardItem header bordered>
          <Text style={styles.nomeAnimal}>{animal.nome}</Text>
        </CardItem>
        <CardItem bordered>
          <Body style={styles.imageContainer}>
            <Image
              source={{
                uri: animal.urlImagem,
              }}
              style={styles.imagemAnimal}

```

```

        />
      </Body>
    </CardItem>
    <CardItem footer bordered>
      <BotaoFavoritar
        favoritado={this.isFavoritado(animal, this.props.usuarioLogado)}
        favoritarCallback={() =>
          this.props.favoritar(animal, this.props.usuarioLogado)
        }
        desfavoritarCallback={() =>
          this.props.desfavoritar(animal, this.props.usuarioLogado)
        }
      />
      <Text>
        Este animal
        {animal.favoritoUsuarios.length > 0
          ? ` já foi favoritado por ${animal.favoritoUsuarios.length} usuário(s)`
        : ' ainda não foi favoritado'}
      </Text>
    </CardItem>
  </Card>
);
}
}

const mapStateToProps = state => {
  return {
    animais: state.animais,
    usuarioLogado: state.usuarioLogado,
  };
};

const mapDispatchToProps = dispatch =>
  bindActionCreators({favoritar, desfavoritar}, dispatch);

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(Animal);

const styles = StyleSheet.create({
  nomeAnimal: {fontSize: 18, fontWeight: 'bold'},
  imagemAnimal: {width: width * 0.7, height: width * 0.7},
  imageContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
});

```

Nossa aplicação voltou a funcionar e já está com todo o estado gerenciado pelo Redux, o que a tornará mais facilmente escalável.

Conectando às APIs

Chegou a hora de deixarmos de utilizar dados fixos começarmos a consumir as APIs de backend. Já temos um servidor levantado com as seguintes funcionalidades:

Funcionalidade	Método HTTP	URL
Listar Animais	GET	https://cozooapi.herokuapp.com/v1/animais
Cadastrar Animal	POST	https://cozooapi.herokuapp.com/v1/animais
Detalhar Animal	GET	https://cozooapi.herokuapp.com/v1/animais/{id}
Atualizar Animal	PUT	https://cozooapi.herokuapp.com/v1/animais/{id}
Excluir Animal	DELETE	https://cozooapi.herokuapp.com/v1/animais/{id}

Caso queira dar uma olhada no código fonte deste serviço, ele pode ser acessado em <https://github.com/tiagolpadua/cozooapi>.

O código de acionamento de nossa API ficará isolado em uma pasta `api` dentro de `src`, mas para realizar os requests usaremos a biblioteca `axios` (<https://github.com/axios/axios>), vamos instalá-la em nosso projeto:

```
> npm install axios
```

Agora criaremos o arquivo que faz as chamadas para as APIs:

- `src/api/index.js`

```
import axios from 'axios';

const api = axios.create({
  baseURL: 'https://cozooapi.herokuapp.com/v1/',
});

export function listaAnimais() {
  return api.get('/animais');
}

export function excluirAnimal(id) {
  return api.delete(`/animais/${id}`);
}

export function incluirAnimal(animal) {
  return api.post('/animais', animal);
}

export function atualizarAnimal(animal) {
  return api.put(`/animais/${animal._id}`, animal);
}
```

Devemos também ajustar o arquivo de constantes para incluir a nova constante que representará a ação

`CARREGAR_ANIMAIS` :

- `src/constants.js`

```
const FAVORITAR = 'FAVORITAR';
const DESFAVORITAR = 'DESFAVORITAR';
const CARREGAR_ANIMAIS = 'CARREGAR_ANIMAIS';

export {FAVORITAR, DESFAVORITAR, CARREGAR_ANIMAIS};
```

O próximo passo é chamar esta API a partir do arquivo que define as ações:

- `src/actions.js`

```
import {carregarAnimaisAPI} from './api';
import {CARREGAR_ANIMAIS, DESFAVORITAR, FAVORITAR} from './constants';

export function carregarAnimais() {
  carregarAnimaisAPI()
    .then(res => ({
      type: CARREGAR_ANIMAIS,
      data: res.data,
    }))
    .catch(error => {
      console.warn(error.message);
    });
}

// ...
```

Agora, vamos ajustar nosso reducer de animais para aceitar a ação `CARREGAR_ANIMAIS` :

- `src/reducers/animais.js`

```
import {animais} from '../../../../../data.json';
import {FAVORITAR, DESFAVORITAR, CARREGAR_ANIMAIS} from '../constants.js';

const initialState = animais;

function atualizaAnimal(listaAnimais, animal) {
  return listaAnimais.map(a => (a._id === animal._id ? animal : a));
}

export default function animaisReducer(state = initialState, action) {
  switch (action.type) {
    case CARREGAR_ANIMAIS:
      return action.data;
```

```
// ...
}
}
```

O último passo é fazer com que, no momento de carregamento da tela de listagem, haja a solicitação de carregamento da lista de animais na aplicação:

- `src/components/ListaAnimais.js`

```
import React, {Component} from 'react';
import {FlatList} from 'react-native';
import {connect} from 'react-redux';
import Animal from './Animal';
import {carregarAnimais} from '../actions';
import {bindActionCreators} from 'redux';

class ListaAnimais extends Component {
  // Novidade aqui
  componentDidMount() {
    this.props.carregarAnimais();
  }

  render() {
    const {animais} = this.props;
    return (
      <FlatList
        data={animais}
        renderItem={({item}) => <Animal animal={item} />}
        keyExtractor={item => item.nome}
      />
    );
  }
}

const mapStateToProps = state => {
  return {
    animais: state.animais,
    usuarioLogado: state.usuarioLogado,
  };
};

// Novidade aqui
const mapDispatchToProps = dispatch =>
  bindActionCreators({carregarAnimais}, dispatch);

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(ListaAnimais);
```

Mas se tentarmos executar nossa app vamos receber um erro:



Ou seja, uma ação deve retornar um "objeto plano".

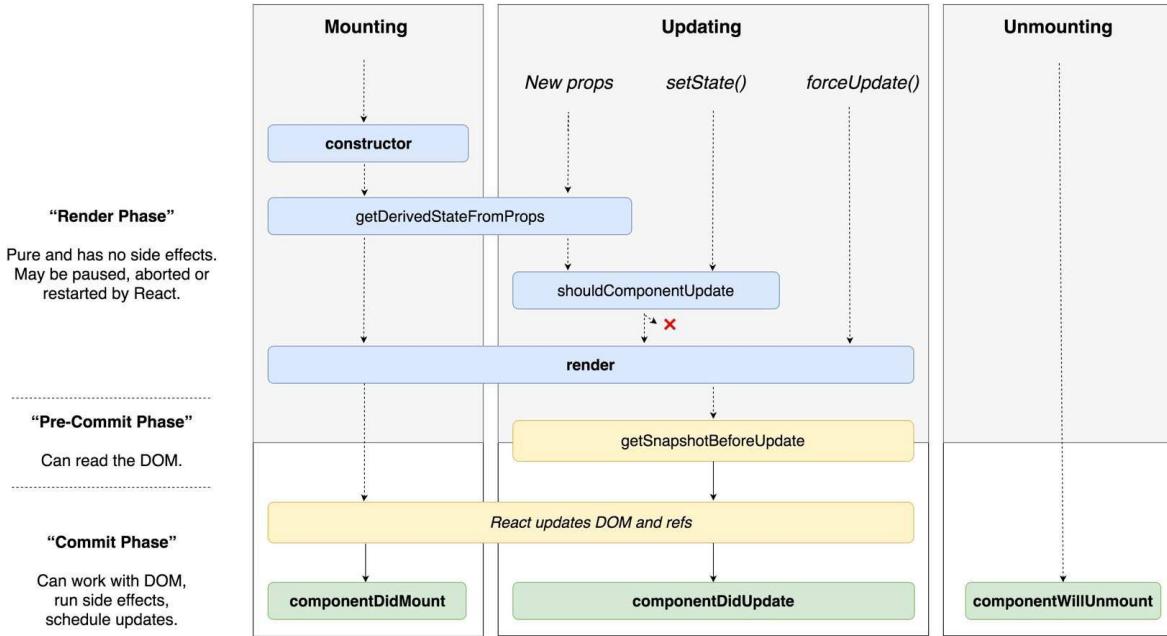
Ciclo de vida do React

Antes de corrigirmos este problema, é interessante entender um pouco do ciclo de vida do React.

Existem várias funções fornecidas pelo React Native para instanciar, montar, renderizar e, eventualmente, atualizar, desmontar e destruir componentes. Essa API nos ajuda a inicializar, atualizar os dados da maneira correta.

A API do ciclo de vida do componente

No React Native, a API contém três fases principais: fase de montagem, fase de atualização e fase de desmonte:



Agora vamos voltar ao nosso problema principal!

Redux Thunk

Com uma store Redux básica e simples, você só pode fazer atualizações síncronas simples despachando uma ação. O middleware Redux Thunk amplia as habilidades da store e permite escrever uma lógica assíncrona que interage com a store.

Os thunks são o middleware recomendado para a lógica básica de efeitos colaterais do Redux, incluindo lógica síncrona complexa que precisa de acesso ao armazenamento e lógica assíncrona simples como solicitações AJAX. (<https://github.com/reduxjs/redux-thunk>)

O middleware Redux Thunk permite escrever criadores de ação que retornam uma função em vez de uma ação. O thunk pode ser usado para atrasar o envio de uma ação ou para enviar somente se uma determinada condição for atendida. A função interna recebe os métodos de armazenamento expedidos e `getState` como parâmetros.

O primeiro passo é instalá-lo:

```
> npm install redux-thunk
```

Agora, vamos ajustar nossa função em `configureStore` para utilizar o *redux-thunk*:

- `configureStore.js`

```
import {applyMiddleware, createStore} from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

export default function configureStore() {
```

```
let store = createStore(rootReducer, applyMiddleware(thunk));
return store;
}
```

Nossa action também deve ser ajustada para utilizar uma função `dispatch` :

- `actions.js`

```
export function carregarAnimais() {
  return dispatch => {
    carregarAnimaisAPI()
      .then(res => {
        dispatch({
          type: CARREGAR_ANIMAIS,
          data: res.data,
        });
      })
      .catch(error => {
        console.warn(error.message);
      });
  };
}
```

Após a realização destes passos, nossa primeira chamada à API deve estar funcionando corretamente e os animais devem estar sendo listados na aplicação.

Login

Agora que já estamos conseguindo nos conectar com a API do sistema, é hora de criarmos uma tela de login para identificar o usuário atual.

Login.js

Primeiro, vamos fazer toda a parte visual utilizando diversos componentes do Native Base:

- `src/components/Login.js`

```
import {Button, Form, Input, Item as FormItem, Label, Text} from 'native-base';
import React, {Component} from 'react';
import {StyleSheet} from 'react-native';
import {connect} from 'react-redux';
import {bindActionCreators} from 'redux';
import {carregarAnimais} from '../actions';

class Login extends Component {
  render() {
    return (
      <Form>
        <FormItem floatingLabel>
          <Label>Usuario</Label>
          <Input />
        </FormItem>
        <FormItem floatingLabel last>
          <Label>Senha</Label>
          <Input secureTextEntry={true} />
        </FormItem>

        <Button full primary style={styles.botaoLogin}>
          <Text>Logar</Text>
        </Button>
      </Form>
    );
  }
}

const mapStateToProps = () => ({});

const mapDispatchToProps = {};

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(Login);
```

```
const styles = StyleSheet.create({
  botaoLogin: {marginTop: 10},
});
```

Em seguida, vamos trocar o componente principal de `App.js` para nossa tela de `Login`:

- `App.js`

```
export default class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Container>
          <Header style={styles.header}>
            <Title>Controle de Animais</Title>
          </Header>
          <Content padder>
            {/* Novidade aqui */}
            <Login />
          </Content>
        </Container>
      </Provider>
    );
  }
}
```

Vamos agora capturar o input do usuário ajustando o estado do componente e associar uma função à ação de login:

- `src/components/Login.js`

```
class Login extends Component {
  constructor(props) {
    super(props);

    // Novidade aqui
    this.state = {
      usuario: '',
      senha: '',
    };
  }

  // Novidade aqui
  login = () => {
    console.warn(this.state.usuario + ' ' + this.state.senha);
  };

  render() {
    return (

```

```

<Form>
  <FormItem floatingLabel>
    <Label>Usuario</Label>
    {/* Novidades aqui */}
    <Input
      onChangeText={text => this.setState({usuario: text})}
      value={this.state.text}
    />
  </FormItem>
  <FormItem floatingLabel last>
    <Label>Senha</Label>
    {/* Novidades aqui */}
    <Input
      secureTextEntry={true}
      onChangeText={text => this.setState({senha: text})}
      value={this.state.text}
    />
  </FormItem>

  <Button full primary style={styles.botaoLogin} onPress={this.login}>
    <Text>Logar</Text>
  </Button>
</Form>
);
}
}

```

Para melhorar a experiência do usuário, podemos desabilitar a capitalização automática da primeira letra no campo de input de texto:

```

<Input
  onChangeText={text => this.setState({usuario: text})}
  autoCapitalize="none"
  value={this.state.text}
/>

```

Agora que já temos todo o front-end pronto, temos que criar as ações do Redux necessárias para o Login:

- `constants.js`

```

const LOGIN = 'LOGIN';
const FAVORITAR = 'FAVORITAR';
const DESFAVORITAR = 'DESFAVORITAR';
const CARREGAR_ANIMAIS = 'CARREGAR_ANIMAIS';

export {LOGIN, FAVORITAR, DESFAVORITAR, CARREGAR_ANIMAIS};

```

Criar a ação de login:

- actions.js

```
import {carregarAnimaisAPI, loginAPI} from './api';
import {LOGIN, CARREGAR_ANIMAIS, DESFAVORITAR, FAVORITAR} from './constants';

export function login(usuario, senha) {
  return dispatch => {
    loginAPI(usuario, senha)
      .then(res => {
        console.warn(res);
        dispatch({
          type: LOGIN,
          data: {usuarioLogado: usuario, token: res.data.token},
        });
      })
      .catch(error => {
        console.warn(error.message);
      });
  };
}
```

A chamada de API correspondente:

- /api/index.js

```
import axios from 'axios';

// ...

export function loginAPI(usuario, senha) {
  return api.post('/login', {usuario, senha});
}
```

E por fim ajustar o reducer:

- usuarioLogado.js

```
import {LOGIN} from '../constants';

const initialState = null;

export default function usuarioLogadoReducer(state = initialState, action) {
  switch (action.type) {
    case LOGIN:
      return action.data;

    default:
      return state;
  }
}
```

```
}
```

Com todo o fluxo pronto, podemos ajustar nosso componente para utilizar a nova ação:

- `Login.js`

```
class Login extends Component {  
  // ...  
  
  // Novidade aqui  
  login = () => {  
    this.props.login(this.state.usuario, this.state.senha);  
  };  
  
  // ...  
}  
  
const mapStateToProps = () => ({});  
  
const mapDispatchToProps = dispatch => bindActionCreators({login}, dispatch);  
  
// ...
```

A operação foi realizada, porém após o login ainda não estamos navegando para a tela de listagem de animais. Esta é nossa próxima missão.

Navegando entre telas

Para efetuar a navegação utilizaremos a biblioteca *React Navigation*.

O React Navigation nasceu da necessidade da comunidade React Native de uma solução de navegação extensível e fácil de usar, escrita inteiramente em JavaScript (para que você possa ler e entender toda o fonte), com poderosas primitivas nativas. (<https://reactnavigation.org/docs/en/getting-started.html>)

Para começar a utilizá-la, devemos primeiramente instalar as dependências necessárias:

```
> npm install react-navigation react-native-gesture-handler react-navigation-stack
```

Agora vamos ajustar nosso componente principal da aplicação inicializando o React Navigation e descrevendo quais são as telas de nossa aplicação:

- App.js

```
import {Container, Content, Header, Title} from 'native-base';
import React, {Component} from 'react';
import {StyleSheet, Text} from 'react-native';
import {createAppContainer} from 'react-navigation';
import {createStackNavigator} from 'react-navigation-stack';
import {Provider} from 'react-redux';
import configureStore from './configureStore';
import Login from './components/Login';

const store = configureStore();

// Novidade aqui
const AppNavigator = createStackNavigator(
{
  Login,
},
{
  initialRouteName: 'Login',
},
);

const Navigation = createAppContainer(AppNavigator);

export default class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Container>
          <Header style={styles.header}>
            <Title>Controle de Animais</Title>
          </Header>
    
```

```
    /* Novidade aqui */
    <Navigation />
</Container>
</Provider>
);
}
}

// ...
```

O componente `Content` do Native Base apresenta uma incompatibilidade com o React Navigation, por isso foi necessário migrá-lo para o componente inferior:

- `Login.js`

```
render() {
  return (
    <Content padder>
      <Form>
        <FormItem floatingLabel>
          <Label>Usuario</Label>
          <Input
            onChangeText={text => this.setState({usuario: text})}
            autoCapitalize="none"
            value={this.state.usuario}
          />
        </FormItem>
        <FormItem floatingLabel last>
          <Label>Senha</Label>
          <Input
            secureTextEntry={true}
            onChangeText={text => this.setState({senha: text})}
            value={this.state.senha}
          />
        </FormItem>

        <Button full primary style={styles.botaoLogin} onPress={this.login}>
          <Text>Logar</Text>
        </Button>
      </Form>
    </Content>
  );
}
// ...
```

Você deve ter percebido que há uma barra de título "sobrando" na tela, isso ocorre pois o React Navigation também controla esta barra, vamos então ajustá-la:

- `App.js`

```

const AppNavigator = createStackNavigator(
{
  Login,
},
{
  initialRouteName: 'Login',
  defaultNavigationOptions: {
    header: () => (
      <Header style={styles.header}>
        <Title>Controle de Animais</Title>
      </Header>
    ),
  },
},
);
;

const Navigation = createAppContainer(AppNavigator);

export default class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Container>
          <Navigation />
        </Container>
      </Provider>
    );
  }
}

```

Nossa aplicação deve estar funcionando normalmente, porém está "parada" na tela de Login, mas agora chegou o momento de realizarmos a navegação "de fato". Primeiro, vamos incluir a tela `ListaAnimais` na lista de telas do sistema:

- `App.js`

```

const AppNavigator = createStackNavigator(
{
  Login,
  // Novidade aqui
  ListaAnimais,
},
{
  initialRouteName: 'Login',
  defaultNavigationOptions: {
    header: () => (
      <Header style={styles.header}>
        <Title>Controle de Animais</Title>
      </Header>
    ),
  },
}

```

```
        ),
    },
},
);
```

O próximo passo é ajustar a action de login:

- `actions.js`

```
export function login(usuario, senha) {
  return dispatch => {
    // Este return é necessário para que haja uma promessa na resposta do acionamento da action
    return loginAPI(usuario, senha).then(res => {
      dispatch({
        type: LOGIN,
        data: {usuarioLogado: usuario, token: res.data.token},
      });
    });
  };
}
```

Por fim, a tela de login deve realizar a navegação através de um objeto `navigation` que é fornecido diretamente nas props do componente:

- `Login.js`

```
login = () => {
  this.props
    .login(this.state.usuario, this.state.senha)
    .then(() => this.props.navigation.navigate('ListaAnimais'))
    .catch(error => console.warn(error));
};
```

Alertas

Neste momento podemos fazer uma melhoria de usabilidade, definindo uma forma de exibição de erros em nossa aplicação.

No entanto, o componente de exibição de mensagens de alerta nativo é diferente para o Android e para o IOS, assim, possuem implementações diferentes de exibição.

Para resolver este problema podemos utilizar uma técnica de criação de arquivos com a extenção `.android.js` e `.ios.js`. Arquivos com o nome da plataforma em sua extensão serão aplicados corretamente dependendo da plataforma de execução do aplicativo:

- `src/util/Alerta.android.js`

```
import {ToastAndroid} from 'react-native';

export default class Alerta {
  static mensagem(texto) {
    ToastAndroid.show(texto, ToastAndroid.LONG);
  }
}
```

- src/util/Alerta.ios.js

```
import {ToastAndroid} from 'react-native';

export default class Alerta {
  static mensagem(texto) {
    ToastAndroid.show(texto, ToastAndroid.LONG);
  }
}
```

Agora podemos utilizar os métodos estáticos para exibir a mensagem correta de acordo com a plataforma de execução:

- Login.js

```
login = () => {
  this.props
    .login(this.state.usuario, this.state.senha)
    .then(() => this.props.navigation.navigate('ListaAnimais'))

  // Novidade aqui
  .catch(() =>
    Alerta.mensagem('Verifique o usuário e senha e tente novamente.'))
};

};
```

Loading

A resposta visual de nossa aplicação ainda está deixando a desejar. A API de login pode demorar um pouco a responder e o usuário não possui nenhum feedback visual disso. Para resolver, vamos implementar um efeito de Loading nas telas. Faremos isso utilizando um estado do Redux:

- constants.js

```
const LOGIN = 'LOGIN';
const FAVORITAR = 'FAVORITAR';
const DESFAVORITAR = 'DESFAVORITAR';
const CARREGAR_ANIMAIS = 'CARREGAR_ANIMAIS';
```

```
// Novidade aqui
const SET_LOADING = 'SET_LOADING';

export {LOGIN, FAVORITAR, DESFAVORITAR, CARREGAR_ANIMAIS, SET_LOADING};
```

Criaremos agora um novo reducer para o estado de loading:

- `src/reducers/loading.js`

```
import {SET_LOADING} from '../constants';

const initialState = false;

export default function loadingReducer(state = initialState, action) {
  switch (action.type) {
    case SET_LOADING:
      return action.data;

    default:
      return state;
  }
}
```

O novo reducer deve ser incluído na lista de reducers:

- `src/reducers/index.js`

```
import {combineReducers} from 'redux';
import animaisReducer from './animais';
import usuarioLogadoReducer from './usuarioLogado';

// Novidade
import loadingReducer from './loading';

const rootReducer = combineReducers({
  animais: animaisReducer,
  usuarioLogado: usuarioLogadoReducer,

  // Novidade
  loading: loadingReducer,
});

export default rootReducer;
```

Vamos ajustar nossa action de Login para que também controle o estado de loading da tela:

- `actions.js`

```
export function login(usuario, senha) {
```

```

return dispatch => {
  // Novidade aqui
  dispatch({
    type: SET_LOADING,
    data: true,
  });

  return (
    loginAPI(usuario, senha)
      .then(res => {
        dispatch({
          type: LOGIN,
          data: {usuarioLogado: usuario, token: res.data.token},
        });
      })
    )

    // Novidade aqui
    .finally(() => {
      dispatch({
        type: SET_LOADING,
        data: false,
      });
    })
  );
}
}

```

Criaremos agora nosso componente `Carregando` que exibirá um spinner indicando que uma operação demorada está ocorrendo:

- `src/components/Carregando.js`

```

import React, {Component} from 'react';
import {ActivityIndicator, StyleSheet, View} from 'react-native';
import {connect} from 'react-redux';

class Carregando extends Component {
  render() {
    return (
      this.props.loading && (
        <View style={styles.loading}>
          <ActivityIndicator size="large" />
        </View>
      )
    );
  }
}

const mapStateToProps = state => {
  return {

```

```
        loading: state.loading,
    };
};

const mapDispatchToProps = {};

export default connect(
    mapStateToProps,
    mapDispatchToProps,
)(Carregando);

const styles = StyleSheet.create({
    loading: {
        position: 'absolute',
        left: 0,
        right: 0,
        top: 0,
        bottom: 0,
        alignItems: 'center',
        justifyContent: 'center',
    },
});
```

Agora basta inserir este novo componente em nosso componente principal:

- `App.js`

```
export default class App extends Component {
    render() {
        return (
            <Provider store={store}>
                <Container>
                    <Navigation />

                    {/* Novidade aqui */}
                    <Carregando />
                </Container>
            </Provider>
        );
    }
}
```

Restringindo o "Voltar"

Talvez você não tenha percebido, mas após o usuário navegar para a tela de listagem, se ele pressionar o botão "voltar" do dispositivo ele retornará para a tela de login, isso não faz muito sentido, vamos corrigir este comportamento emitindo uma comando especial para a navegação:

- `Login.js`

```
login = () => {
  this.props
    .login(this.state.usuario, this.state.senha)
    .then(() => {
      // Novidade aqui
      const resetAction = StackActions.reset({
        index: 0,
        actions: [NavigationActions.navigate({routeName: 'ListaAnimais'})],
      });
      this.props.navigation.dispatch(resetAction);
    })
    .catch(() =>
      Alerta.mensagem('Verifique o usuário e senha e tente novamente.'),
    );
};
```

Nossa aplicação já possui uma ótima estrutura básica e está pronta para a inclusão de novas funcionalidades!