React Native

Tabela de conteúdos

Introdução

O framework ReactNative é a última palavra em desenvolvimento multiplataforma de aplicativos mobile. Dentre os tópicos do curso, 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 curso é pra você.

Pré-requisitos

Espera-se que o aluno possua prévio conhecimento nos tópicos abaixo para que possa tirar o máximo de aproveitamento do curso:

- ES6 - https://javascript.info/
- ReactJS - https://reactjs.org/tutorial/tutorial.html
- CSS - https://developer.mozilla.org/pt-BR/docs/Aprender/CSS/Introduction_to_CSS

Visão geral do Curso

- React: Uma biblioteca JavaScript para construir interfaces com o usuário;
- Expo: Expo é um conjunto de ferramentas gratuito e de código aberto criado em torno do React Native para ajudá-lo a criar projetos iOS e Android nativos usando JavaScript e React.
- Create React Native Apps: Permite a criação de aplicativos React Native em qualquer sistema operacional;
- React Navigation: Roteamento e navegação para aplicativos React Native;
- Redux: Contêiner de estado previsível para aplicativos JavaScript;
- 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;
- Superagent: O SuperAgent é uma pequena biblioteca de solicitação HTTP progressiva do lado do cliente e um módulo Node.js com a mesma API, com muitos recursos de cliente HTTP de alto nível;
- 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

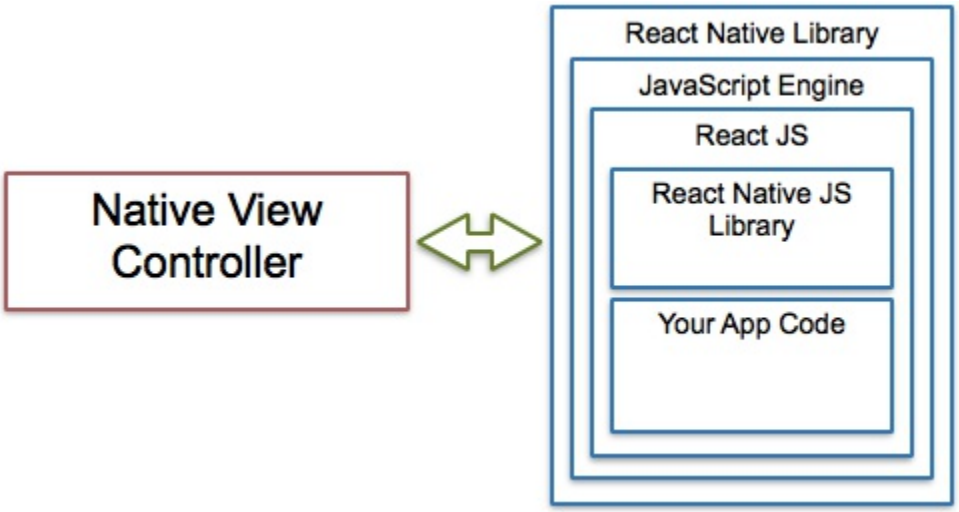
O React Native permite a criação de aplicativos móveis usando apenas JavaScript. Ele usa o mesmo design que o React, permitindo compor uma rica interface de usuário a partir de componentes declarativos.

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:

Ferramentas instaladas no computador:

- Android Studio: https://developer.android.com/studio/
- NodeJS e NPM: https://nodejs.org/en/download/
- Expo Cli: `npm install -g expo-cli`
- Reactotron: https://github.com/infinitered/reactotron/releases
- VSCode - https://code.visualstudio.com/

São recomendados os seguintes plugins para o VSCode:

- EditorConfig
- ESLint
- Prettier
- vscode-icons: File -> Preferences -> File Icon Theme para ativar

Ferramentas instaladas no dispositivo mobile:

- Expo Client (No Mobile)

Testando instalação

Podemos testar a instalação de algumas das ferramentas:

```
PS C:\Users\usuario\> --version
v10.4.1

PS C:\Users\usuario\> expo --version
2.3.8
...
```

Começando com o React Native

Recaptulando sobre o 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 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 e . Mas também pode compor e renderizar componentes React personalizados. Por exemplo, podemos nos referir a toda a lista de compras escrevendo . 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/

Recapitulando sobre o 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 visualizações. 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.

Para mudar algo no estado, você precisa despachar uma ação. Uma ação é um objeto JavaScript simples (observe como não introduzimos nenhuma mágica?) Que descreva 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 são 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". Novamente, nada de mágico - 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 idéia 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.

```
console.log(store.getState())

/* Prints
{
  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 exibições nem os retornos de chamada 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
      })
    default:
      return state
  }
}

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

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
  }
});
```

Introdução ao 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.

Introdução ao Expo Platform

Expo é um conjunto de ferramentas gratuitas e de código aberto criadas em torno do React Native para ajudá-lo a criar projetos iOS e Android nativos usando JavaScript e React.

Os aplicativos Expo são aplicativos React Native que contêm o Expo SDK. O SDK é uma biblioteca nativa e JS que fornece acesso à funcionalidade do sistema do dispositivo (como câmera, contatos, armazenamento local e outros hardwares). Isso significa que você não precisa usar o Xcode ou o Android Studio ou escrever qualquer código nativo, além de tornar seu projeto puro JavaScript, ou seja, muito portável, pois pode ser executado em qualquer ambiente nativo que contenha o Expo SDK.

O Expo também fornece componentes de interface do usuário para lidar com uma variedade de casos de uso que quase todos os aplicativos cobrirão, mas não serão integrados ao núcleo React Native, por exemplo, ícones, desfocar visões e muito mais.

Finalmente, o Expo SDK fornece acesso a serviços que normalmente são difíceis de gerenciar, mas são exigidos por quase todos os aplicativos. O mais popular entre estes: o Expo pode gerenciar os assets, pode cuidar de notificações push, e pode construir binários nativos que estão prontos para implantar na loja de aplicativos.

Laboratório: Criando o projeto React Native: Sandbox

Baixar a aplicação Expo para o seu celular (Android ou iOs)

Acessar o endereço https://snack.expo.io/ e inserir o código abaixo:

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

export default class App extends React.Component {
  render() {
    return (
      <View style={{ marginTop: 50 }}>
        <Text>Olá Mundo!</Text>
      </View>
    );
  }
}
```

Clicar em "Run", selecionar QR Code e escanear o QR Code com o App Expo do Celular.

Criando um Belo Hello World

Primeiramente vamos instalar o `expo-cli` globalmente e iniciar um projeto de exemplo, para isso, abra o prompt de comandos (cmd) e execute os comandos a seguir:

```
C:\Users\nome> npm install -g expo-cli
...
C:\Users\nome> expo --version
2.2.5
C:\Users\nome>mkdir projects
C:\Users\nome> cd projects
C:\Users\nome\projects> expo init HelloWorld
? Choose a template: blank
[19:30:41] Extracting project files...
[19:31:48] Customizing project...

Your project is ready at C:\Users\nome\projects\HelloWorld
To get started, you can type:

  cd HelloWorld
  expo start

C:\Users\nome\projects> cd HelloWorld
```

Opcionalmente, ao invés de utilizar o próprio celular, pode ser utilizado o emulador do Android.

Para isso, abra o AndroidStudio e clique em:

- Tools -> AVD Manager -> Actions | Play

Ativando a depuração USB - Somente para celular Android

- Primeiro, abra "Configurações"
- Role a tela até encontrar "Sistema"
- Role a tela até encontrar "Sobre o dispositivo"
- Role para baixo novamente e encontre a entrada com o número da versão ou compilação (build number).
- Comece a tocar na seção "Número da versçai", e agora o Android exibirá uma mensagem informando que, em x cliques, você se tornará um desenvolvedor. Continue tocando até que o processo esteja concluído.
- Volte até a tela "Sistema"
- Role a tela até encontrar "Programador"

- Role a tela até encontrar "Depuração USB" e deixe a opção ativa

Ajustando o Hello World

Abra o VS Code e clique em: File -> Open Folder e abra a pasta do projeto criado.

Ative o terminal do VS Code: Terminal -> New Terminal

Agora, vamos executar a aplicação recém criada:

```
PS C:\Users\nome\projects\HelloWorld> expo start
[18:09:08] Starting project at C:\Users\nome\projects\HelloWorld
[18:09:09] Expo DevTools is running at http://localhost:19002
[18:09:09] Opening DevTools in the browser... (press shift-d to disable)
[18:09:20] Starting Metro Bundler on port 19001.
[18:09:26] Successfully ran `adb reverse`. Localhost URLs should work on the connected
Android device.
[18:09:27] Tunnel ready.

  exp://169.254.186.144:19000

  To run the app with live reloading, choose one of:
  • Sign in as @fulano in Expo Client on Android or iOS. Your projects will automatical
ly appear in the "Projects" tab.
  • Scan the QR code above with the Expo app (Android) or the Camera app (iOS).
  • Press a for Android emulator.
  • Press e to send a link to your phone with email/SMS.

Press ? to show a list of all available commands.
Logs for your project will appear below. Press Ctrl+C to exit.
```

Deve ser aberta uma página da Web com a interface do Expo, escolha a opção "Local", use a aplicação expo para escanear o QR Code que é exibido.

Agora, vamos customizar a tela da aplicação que está sendo exibida:

```
// App.js
import * as React from "react";
import { Text, View, StyleSheet } from "react-native";
import { Constants } from "expo";

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.paragraph}>Olá Mundo!</Text>
      </View>
```

```
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    paddingTop: Constants.statusBarHeight,
    backgroundColor: "#ecf0f1",
    padding: 8
  },
  paragraph: {
    margin: 24,
    fontSize: 18,
    fontWeight: "bold",
    textAlign: "center"
  }
});
```

Conhecendo o React Native

Criando componentes

Vamos colocar uma segunda frase em nossa tela:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.paragraph}>Olá Mundo!</Text>
        <Text style={styles.paragraph}>
          Vamos começar a aprender React Native?
        </Text>
      </View>
    );
  }
}
// Código posterior omitido
```

E se quiséssemos utilizar o mesmo estilo em outras partes de nosso sistema? Para que isso fique mais fácil podemos criar um componente.

Vamos criar uma pasta chamada components e dentro dela um arquivo chamado Mensagem.js:

```
// components/Mensagem.js
import * as React from "react";
import { StyleSheet, Text } from "react-native";

export class Mensagem extends React.Component {
  render() {
    return (
      <Text style={styles.paragraph}>
        Nosso primeiro componente React Native!
      </Text>
    );
  }
}

const styles = StyleSheet.create({
  paragraph: {
    margin: 24,
    fontSize: 18,
    fontWeight: "bold",
    textAlign: "center"
  }
});
```

Agora, no arquivo App.js podemos importar o componente que foi criado anteriormente:

```
// App.js
// Código anterior omitido

// Novidade aqui!
import { Mensagem } from "./components/Mensagem";

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.paragraph}>Olá Mundo!</Text>

        {/* Novidade aqui! */}
        <Mensagem />
      </View>
    );
  }
}
// Código posterior omitido
```

Passando props

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

Nosso componente está com o texto "Fixo", o ideal é que o texto da mensagem seja parametrizado, vamos fazer isso agora:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.paragraph}>Olá Mundo!</Text>

        {/* Novidade aqui! */}
        <Mensagem texto="Passando propriedades dinamicamente!" />
      </View>
    );
  }
}
// Código posterior omitido
```

```
// components/Mensagem.js
// Código anterior omitido
export class Mensagem extends React.Component {
  render() {
    return (
      <Text style={styles.paragraph}>
        {/* Novidade aqui! */}
        {this.props.texto}
      </Text>
    );
  }
}
// Código posterior omitido
```

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.

Será então que podemos tornar nosso código mais enxuto? Vamos utilizar mais nosso componente de mensagens:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
```

```
  render() {
    return (
      <View style={styles.container}>
        {/* Novidade aqui! */}
        <Mensagem texto="Olá Mundo via props!" />
        <Mensagem texto="Passando propriedades dinamicamente!" />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    paddingTop: Constants.statusBarHeight,
    backgroundColor: "#ecf0f1",
    padding: 8
  }
});
// Código posterior omitido
```

Setando state

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 no construtor e, em seguida, chamar `setState` quando quiser alterá-lo.

Vamos fazer um simples contator de cliques em nossa aplicação:

```
// App.js
// Código anterior omitido
// Novidade aqui!
import { Button, StyleSheet, View } from "react-native";

export default class App extends React.Component {
  // Novidade aqui!
  constructor(props) {
    super(props);
    this.state = { clicks: 0 };
  }

  // Novidade aqui!
  handleClick() {
```

```
    this.setState({
      clicks: this.state.clicks + 1
    });
  }

  render() {
    return (
      <View style={styles.container}>
        {/* Novidade aqui! */}
        <Button
          title={`Clicou ${this.state.clicks} vezes`}
          onPress={this.handleClick.bind(this)}
        />
      </View>
    );
  }
}
// Código posterior omitido
```

Outra forma de fazer o bind:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { clicks: 0 };

    // Novidade aqui!
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({
      clicks: this.state.clicks + 1
    });
  }

  render() {
    return (
      <View style={styles.container}>
        {/* Novidade aqui! */}
        <Button
          title={`Clicou ${this.state.clicks} vezes`}
          onPress={this.handleClick}
        />
      </View>
    );
```

```
  }
}
// Código posterior omitido
```

E mais uma forma de fazer o bind:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { clicks: 0 };
  }

  // Novidade aqui!
  handleClick = () => {
    this.setState({
      clicks: this.state.clicks + 1
    });
  };

  render() {
    return (
      <View style={styles.container}>
        <Button
          title={`Clicou ${this.state.clicks} vezes`}
          onPress={this.handleClick}
        />
      </View>
    );
  }
}
// Código posterior omitido
```

setState também possui uma segunda forma onde é passada uma função como argumento:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { clicks: 0 };
  }

  // Novidade aqui!
  handleClick = () => {
    this.setState(prevState => ({ clicks: prevState.clicks + 1 }));
```

```
  };

  render() {
    return (
      <View style={styles.container}>
        <Button
          title={`Clicou ${this.state.clicks} vezes`}
          onPress={this.handleClick}
        />
      </View>
    );
  }
}
// Código posterior omitido
```

Ciclo de vida

Cada componente React vem com vários métodos que permitem aos desenvolvedores atualizar o estado do aplicativo e refletir a alteração na interface do usuário. Existem três fases principais de um componente, incluindo mounting (montagem), updating (atualização) e unmounting (desmontagem).

Mounting

Esses métodos serão chamados quando uma instância de um componente React for criada e montada no DOM.

constuctor()

Esse método é chamado antes de um componente React ser montado. É essencial chamar `super(props)` antes de qualquer declaração no construtor. Isto ocorre pois nos permitirá chamar o construtor da classe pai e inicializar a si mesmo caso nossa classe estenda qualquer outra classe que tenha o próprio construtor.

O construtor é perfeito para inicializar o estado ou vincular os manipuladores de eventos à instância da classe. Por exemplo:

```
constructor(props) {
  super(props);
  this.state = {
    count: 0,
    value: 'Hey There!',
  };
  this.handleClick = this.handleClick.bind(this);
};
```

O construtor não deve causar nenhum efeito colateral.

componentWillMount()

`componentWillMount` será chamado uma vez antes do componente ser montado e será executado antes da função de renderização.

componentDidMount()

Depois que um componente é montado, esse método é chamado. Este é o local certo para carregar qualquer dado do *endpoint*.

Chamar aqui `setState` irá disparar re-render, então use este método com cuidado.

Updating

componentDidUpdate(prevProps, prevState, snapshot)

Este método será chamado após cada renderização ocorrer. Como esse método é chamado apenas uma vez após a atualização, é um local adequado para implementar quaisquer operações de efeitos colaterais.

Unmounting

componentWillUnmount()

Quando um componente é desmontado ou destruído, este método será chamado. Este é um lugar para fazer alguma limpeza como:

- Invalidando temporizadores
- Cancelar qualquer pedido de rede
- Remover manipuladores de eventos
- Limpar todas as assinaturas

JSX

A sintaxe de tags curiosa que estamos utilizando não é JavaScript nem HTML.

É chamado de JSX e é uma extensão de sintaxe para JavaScript. Recomenda-se usá-lo com o React para descrever como deve ser a interface do usuário. O JSX pode lembrá-lo de uma linguagem de marcação, mas ela vem com todo o poder do JavaScript.

O JSX produz React "elements".

Por que o JSX?

O React adota o fato de que a lógica de renderização é inerentemente associada a outra lógica da interface do usuário: como os eventos são tratados, como o estado muda com o tempo e como os dados são preparados para exibição.

Em vez de separar artificialmente as tecnologias, colocando marcação e lógica em arquivos separados, o React separa as preocupações com unidades fracamente acopladas chamadas “componentes” que contêm ambos.

O React não exige o uso de JSX, mas a maioria das pessoas considera útil como um auxílio visual ao trabalhar com a interface do usuário dentro do código JavaScript.

Incorporando Expressões no JSX

No exemplo abaixo, nós declaramos uma variável chamada *nome* e a usamos dentro do JSX, colocando-a entre chaves:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    const nome = "José";

    return (
      <View style={styles.container}>
        <Text>Olá {nome}</Text>
      </View>
    );
  }
}
// Código posterior omitido
```

Você pode colocar qualquer expressão JavaScript válida dentro das chaves no JSX. Por exemplo, 2 + 2, usuario.nome ou formatMessage(message) são todas expressões JavaScript válidas.

No exemplo abaixo, incorporamos o resultado de chamar uma função JavaScript, formatMessage(message).

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  constructor(props) {
    super(props);
```

```
  }

  formatMessage(message = "") {
    return message.toUpperCase();
  }

  render() {
    const nome = "José";

    return (
      <View style={styles.container}>
        <Text>Olá {this.formatMessage(nome)}</Text>
      </View>
    );
  }
}
// Código posterior omitido
```

JSX é uma expressão também

Após a compilação, as expressões JSX se tornam chamadas de função JavaScript regulares e são avaliadas como objetos JavaScript.

Isso significa que você pode usar o JSX dentro de instruções if e for loops, atribuí-lo a variáveis, aceitá-lo como argumentos e retorná-lo de funções:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  constructor(props) {
    super(props);
  }

  formatMessage(message = "") {
    let el;
    if (message.length > 2) {
      el = <Text>{message.toUpperCase()}</Text>;
    } else {
      el = <Text>{message}</Text>;
    }
    return el;
  }

  render() {
    const nome = "José";
    const saudacao = "oi";

    return (
```

```
      <View style={styles.container}>
        {this.formatMessage(saudacao)}
        {this.formatMessage(nome)}
      </View>
    );
  }
}
// Código posterior omitido
```

Especificando Atributos com JSX

Você pode usar aspas para especificar literais de string como atributos:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <View style={styles.container}>
        <Mensagem texto="Olá José" />
      </View>
    );
  }
}
// Código posterior omitido
```

Você também pode usar chaves para incorporar uma expressão JavaScript em um atributo:

```
// App.js
// Código anterior omitido
export default class App extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    const minhaMsg = "Olá José";
    return (
      <View style={styles.container}>
        <Mensagem texto={minhaMsg} />
      </View>
    );
  }
```

```
}
// Código posterior omitido
```

Componentes de Função e Classe

Já vimos como criar componentes a partir de uma classe (a classe Mensagem), porém, há uma forma simplificada de criar componentes através de funções:

```
// App.js
// Código anterior omitido
function MensagemSimples(props) {
  return <Text>Uma componente de função: {props.texto}</Text>;
}

export default class App extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    const minhaMsg = "Olá José";
    return (
      <View style={styles.container}>
        <MensagemSimples texto={minhaMsg} />
      </View>
    );
  }
}
// Código posterior omitido
```

Essa função é um componente React válido porque aceita um único argumento de objeto "props" (que significa propriedades) com dados e retorna um elemento React. Chamamos esses componentes de "componentes de função" porque são literalmente funções JavaScript.

Instalando Expo Client para desenvolvimento e debug da aplicação

O Expo Client permite que você teste o código diretamente em um dispositivo físico, para isso, ele deve ser instalado através da loja de aplicativos.

Uma grande vantagem do Expo Client é que podemos compilar o código em um computador Windows/Linux e executar em um iPhone.

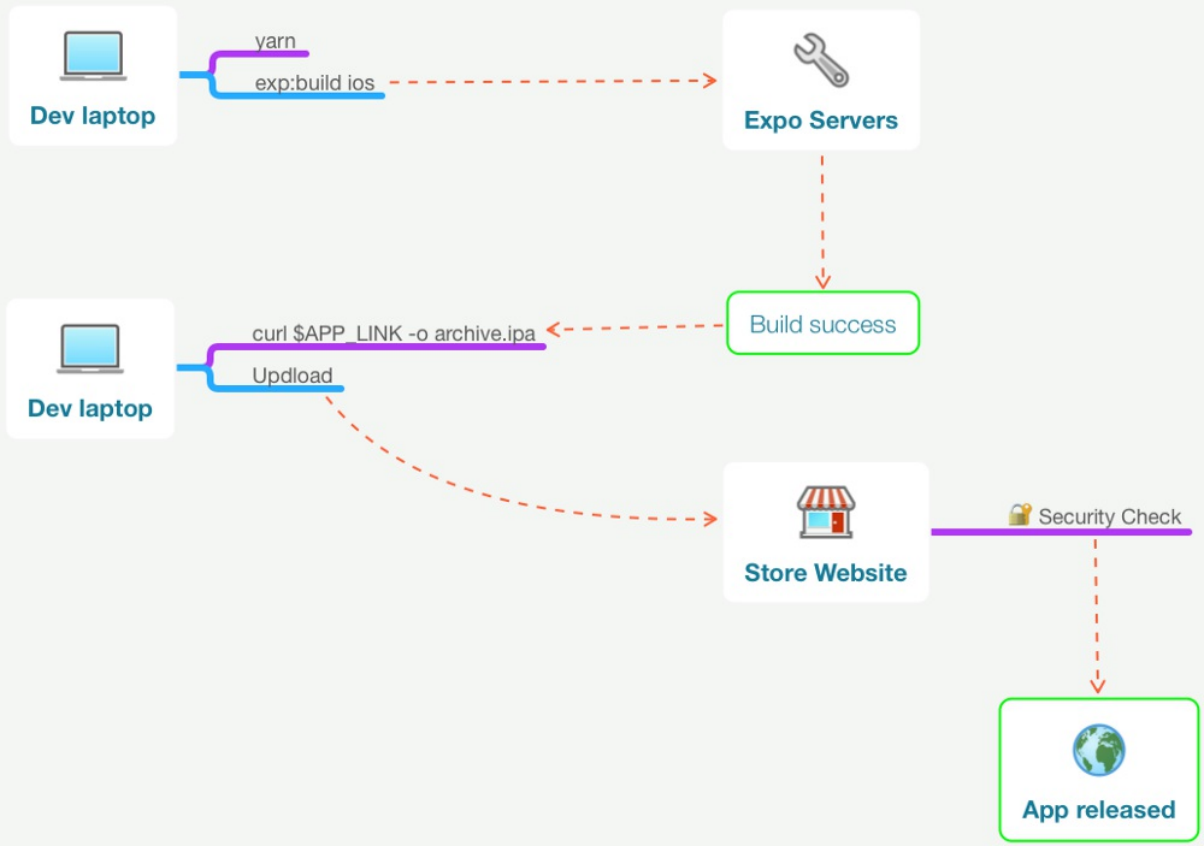
Analizando Expo Client & Dicas

Ao balançar o dispositivo, podemos acessar o menu do desenvolvedor do Expo Cli, onde podemos:

- Reload: Recarregar a aplicação
- Debug JS Remoately: Debugar a aplicação remotamente (Chrome remote debugging)
- Habilitar/Desabilitar o Live Reload
- Habilitar o Hot Reloading
- Ativar o "inspector"
- Exibir o monitor de performance
- Iniciar/Parar o profiler de JavaScript

Buildando um APK ou IPA via Expo CLi

A construção do binário, em sua versão final que será entregue para a loja de aplicativos é realizada no servidor do Expo:

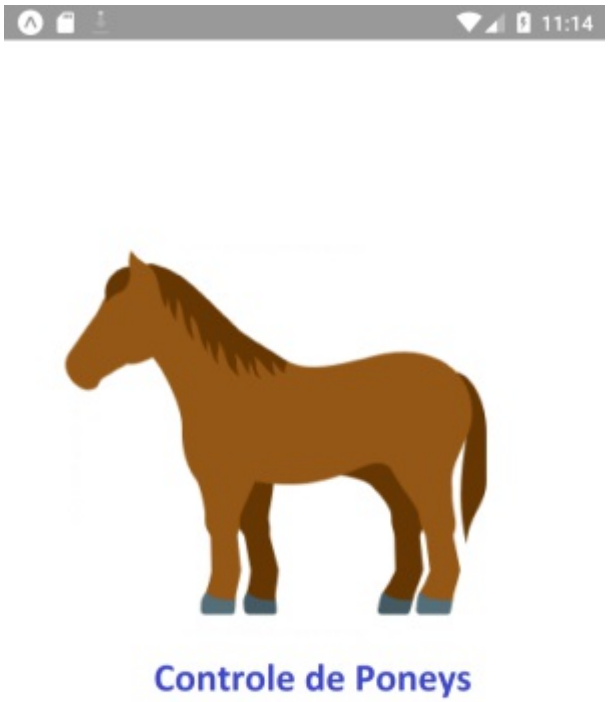


Deve ser impostado o seguinte commando:

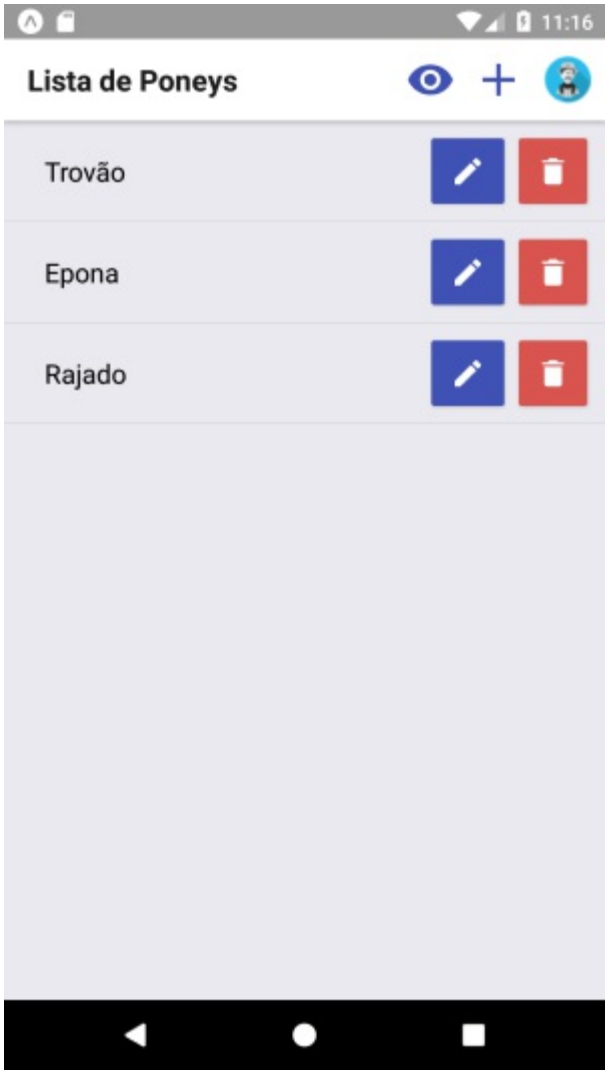
```
PS C:\Users\usuario\projects\HelloWorld> expo build:android
```

Desenvolvendo uma versão mobile para Controle de Poneys

Splash Screen



Tela Principal



Criando o Projeto CoponeyMob

Agora que já conhecemos as estruturas básicas do React Native, vamos criar nosso projeto que será um CRUD (Create/Retrieve/Update/Delete) de poneys.

O primeiro passo é a criação do projeto em si:

```
PS C:\Users\nome\projects> expo init CoponeyMob
? Choose a template: blank
[10:20:22] Extracting project files...
[10:22:13] Customizing project...

Your project is ready at C:\Users\nome\projects\CoponeyMob
To get started, you can type:

  cd CoponeyMob
  expo start

PS C:\Users\nome\projects> cd .\CoponeyMob\
```

```
PS C:\Users\nome\projects\CoponeyMob> expo start
[10:43:15] Starting project at C:\Users\nome\projects\CoponeyMob
[10:43:16] Expo DevTools is running at http://localhost:19002
[10:43:16] Opening DevTools in the browser... (press shift-d to disable)
...
```

Colocando os binários do Android no PATH

Caso os comandos `adb` e `emulator` não estejam disponíveis, verifique a configuração das variáveis de ambiente do Windows:

- Tecla Windows -> Editar as variáveis de ambiente do sistema -> Clique em Path e 'Editar'
- Certifique-se que os caminhos abaixo estão configurados:
  - %USERPROFILE%\AppData\Local\Android\Sdk\platform-tools
  - %USERPROFILE%\AppData\Local\Android\Sdk\emulator

Executando o emulador a partir da linha de comandos:

Pode-se iniciar o emulador direto pela linha de comando para evitar a necessidade de abrir o Android Studio:

```
> emulator -list-avds
Nexus_5X_API_25

> emulator -avd Nexus_5X_API_25
```

O comando 'adb devices' lista os dispositivos conectados, tando emulador quanto físicos:

```
> adb devices
```

Em caso de dificuldade de conexão com o Expo local, pode ser feita tentativa de novo 'adb reverse':

```
> adb -s 0040483203 reverse tcp:19001 tcp:19001
```

É possível também ver o log do dispositivo pela linha de comandos

```
> adb logcat
```

Adicionando Libs ao projeto

Editorconfig

Criar na raiz do projeto o arquivo .editorconfig com o seguinte conteúdo:

```
[*]
end_of_line = lf
insert_final_newline = true
charset = utf-8
```

ESlint

Instalar as seguintes dependências:

npm install --save-dev babel-eslint prettier-eslint eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-react

Criar na raiz do projeto o arquivo .eslintrc com o seguinte conteúdo:

```
{
  "parser": "babel-eslint",
  "env": {
    "es6": true,
    "browser": true,
    "node": true
  },
  "plugins": ["react", "import", "jsx-a11y"],
  "extends": ["eslint:recommended", "plugin:react/recommended"],
  "globals": {
    "Expo": true
  }
}
```

Agora vamos fazer com que haja auto-formatação do código sempre que os arquivos do projeto forem salvos:

No VSCode acessar: File -> Preferences -> Settings -> Workspace Settings -> ... -> Open Settings.json e deixar o arquivo conforme abaixo:

```
{
  // Format a file on save. A formatter must be available, the file must not be auto-sa
ved, and editor must not be shutting down.
  "editor.formatOnSave": true,
  // Enable/disable default JavaScript formatter (For Prettier)
  "javascript.format.enable": false,
  // Use 'prettier-eslint' instead of 'prettier'. Other settings will only be fallbacks
 in case they could not be inferred from eslint rules.
```

```
  "prettier.eslintIntegration": true
}
```

Em seguida, reiniciar o VS Code.

Reactotron

Primeiro, vamos instalar o software no desktop: https://github.com/infinitered/reactotron/releases.

Agora, precisamos incluir a dependência em nosso projeto e fazer o adb reverse da porta 9090 que será utilizada para comunicação:

```
PS C:\Users\nome\projects\CoponeyMob> npm install --save-dev reactotron-react-native
PS C:\Users\nome\projects\CoponeyMob> adb reverse tcp:9090 tcp:9090
```

Criar o arquivo de configuração do Reactotron na raiz do projeto:

```
// ReactotronConfig.js
import Reactotron from "reactotron-react-native";

Reactotron.configure({ port: 9090 }) // controls connection & communication settings
  .useReactNative() // add all built-in react native plugins
  .connect(); // let's connect!
```

Ajustar o arquivo App.js:

```
// App.js
import React from "react";
import { StyleSheet, Text, View } from "react-native";

// Novidade aqui
import "./ReactotronConfig";
import Reactotron from "reactotron-react-native";

export default class App extends React.Component {
  render() {
    // Novidade aqui
    Reactotron.log("Testando a conexão com o Reactotron.");

    return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
      </View>
    );
  }
}
```

Para saber mais:

- https://github.com/infinitered/reactotron/blob/master/docs/quick-start-react-native.md

Sobre a questão de configuração da porta do Reactotron:

- https://github.com/infinitered/reactotron/issues/690

NativeBase

A biblioteca NativeBase apresenta uma série de componentes visuais multiplataforma que nos auxilia no desenvolvimento de nossa aplicação.

Precisamos incluir a dependência em nosso projeto:

```
PS C:\Users\nome\projects\CoponeyMob> npm install --save native-base @expo/vector-icons
```

Demais bibliotecas

As demais bibliotecas que utilizaremos em nosso projeto, serão instaladas durante o desenvolvimento das funcionalidades.

Analizando package.json

Vamos atualizar o nome do projeto no arquivo package.json:

```
{
  "name": "coponeymob",
  "main": "node_modules/expo/AppEntry.js",
  "private": true,
  "scripts": {
    "start": "expo start",
    "android": "expo start --android",
    "ios": "expo start --ios",
    "eject": "expo eject"
  },
  "dependencies": {
    "@expo/vector-icons": "^8.0.0",
    "expo": "^31.0.2",
    "native-base": "^2.8.1",
    "react": "16.5.0",
    "react-native": "https://github.com/expo/react-native/archive/sdk-31.0.0.tar.gz"
  },
  "devDependencies": {
    "babel-eslint": "^10.0.1",
```

```
    "babel-preset-expo": "^5.0.0",
    "eslint-plugin-import": "^2.14.0",
    "eslint-plugin-jsx-a11y": "^6.1.2",
    "eslint-plugin-react": "^7.11.1",
    "prettier-eslint": "^8.8.2",
    "reactotron-react-native": "^2.1.0"
  }
}
```

Configurando a Estrutura do Projeto

Os arquivos relativos ao código fonte de nosso projeto ficarão em uma pasta 'src', e, dentro dela, teremos as pastas 'api', 'assets', 'components' e 'reducers'.

O React Native não nos obriga a seguir uma estrutura pré-definida de pastas no projeto, a excessão é o arquivo `App.js` que é o ponto de início de nossa aplicação.

Vamos baixar e descompactar dentro da pasta src os assets da aplicação: http://bit.ly/cpmassets

Agora, vamos apagar a pasta assets da raiz do projeto e atualizar o arquivo app.json com o novo caminho da splash screen e do ícone da aplicação:

app.json

```
{
  "expo": {
    "name": "CoponeyMob",
    "description": "This project is really great.",
    "slug": "CoponeyMob",
    "privacy": "public",
    "sdkVersion": "31.0.0",
    "platforms": ["ios", "android"],
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./src/assets/icon.png",
    "splash": {
      "image": "./src/assets/splash.png",
      "resizeMode": "contain",
      "backgroundColor": "#ffffff"
    },
    "updates": {
      "fallbackToCacheTimeout": 0
    },
    "assetBundlePatterns": ["**/*"],
    "ios": {
      "supportsTablet": true
    }
  }
```

```
}
```

Para saber mais

How To Structure a React Native App For Scale - https://medium.com/the-andela-way/how-to-structure-a-react-native-app-for-scale-a29194cd33fc

Criando a tela Home

Listando os Poneis

Nossa primeira tarefa mover nossa base de código para dentro da pasta `src`, para isso, iremos alterar o arquivo `App.js` para apontar para um componente que criaremos dentro da pasta `src`:

```
// App.js
import ListaPoneysScreen from "./src/components/ListaPoneysScreen";
export default ListaPoneysScreen;
```

Agora, dentro da pasta `src\components`, vamo criar a tela inicial de nosso sistema, que será uma listagem de poneis:

```
// src/components/ListaPoneysScreen.js
import { Left, ListItem, Text } from "native-base";
import React from "react";
import { FlatList, StyleSheet, View } from "react-native";

class ListarPoneysScreen extends React.Component {
  poneys = [
    { nome: "Tremor" },
    { nome: "Tzar" },
    { nome: "Pégaso" },
    { nome: "Epona" },
    { nome: "Macedonio" },
    { nome: "Vicário" },
    { nome: "Tro" },
    { nome: "Nicanor" },
    { nome: "Niceto" },
    { nome: "Odón" },
    { nome: "Relâmpago" },
    { nome: "Pio" },
    { nome: "Elegante" },
    { nome: "Pompeu" }
  ];

  constructor(props) {
    super(props);
    this.poneys = this.poneys.map((p, idx) => ({ ...p, _id: idx + "" }));
  }

  render() {
```

```
    return (
      <View>
        <FlatList
          data={this.poneys}
          renderItem={({ item }) => (
            <ListItem noIndent>
              <Left>
                <Text style={styles.item}>{item.nome}</Text>
              </Left>
            </ListItem>
          )}
          keyExtractor={item => item._id}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  itemContainer: {
    flex: 1,
    flexDirection: "row",
    borderBottomWidth: 1
  },
  item: {
    padding: 10,
    fontSize: 18,
    height: 44,
    width: 120
  }
});

export default ListarPoneysScreen;
```

Observação

Em relação ao warning:

```
Require cycles are allowed, but can result in uninitialized values. Consider refactorin
g to remove the need for a cycle.
- node_modules\expo\build\environment\logging.js:25:23 in warn
- node_modules\metro\src\lib\polyfills\require.js:109:19 in metroRequire
- node_modules\native-base\dist\src\basic\DatePicker.js:1:774 in <unknown>
...
```

Ele pode ser ignorado com segurança e já existe uma issue aberta na comunidade para corrigir: https://github.com/GeekyAnts/NativeBase/issues/2320

Iniciando o Native Base

A biblioteca do native base necessita que sejam carregados alguns arquivos de fonte e ícones antes da exibição da tela inicial de nossa aplicação (http://docs.nativebase.io/docs/GetStarted.html), vamos criar um componente chamado `CoponeyMob` na pasta `src` que terá esta funcionalidade, vamos também ativar o Reactotron para o Coponeymob:

```
// src/ReactotronConfig.js
import Reactotron from "reactotron-react-native";

Reactotron.configure({ port: 9090 }) // controls connection & communication settings
  .useReactNative() // add all built-in react native plugins
  .connect(); // let's connect!
```

```
// src/CoponeyMob.js
import React from "react";
import { StyleSheet, View } from "react-native";
import Reactotron from "reactotron-react-native";
import ListaPoneysScreen from "./components/ListaPoneysScreen";
import "./ReactotronConfig";
import { Spinner } from "native-base";

Reactotron.log("Testando a conexão com o Reactotron.");

export default class CoponeyMob extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isReady: false
    };
  }

  async componentDidMount() {
    await Expo.Font.loadAsync({
      Roboto: require("native-base/Fonts/Roboto.ttf"),
      Roboto_medium: require("native-base/Fonts/Roboto_medium.ttf"),
      Ionicons: require("native-base/Fonts/Ionicons.ttf")
    });
    setTimeout(() => this.setState({ isReady: true }), 0);
  }
```

```
  render() {
    if (!this.state.isReady) {
      return (
        <View style={styles.loadingContainer}>
          <Spinner />
        </View>
      );
    } else {
      return <ListaPoneysScreen />;
    }
  }
}

const styles = StyleSheet.create({
  loadingContainer: {
    flexDirection: "column",
    justifyContent: "center",
    alignItems: "center",
    height: "100%"
  }
});
```

Devemos ajustar novamente nosso `App.js` para nossa nova tela inicial:

```
// App.js
import CoponeyMob from "./src/CoponeyMob.js";
export default CoponeyMob;
```

Criando uma rota para a tela

Primeiramente vamos instalar a biblioteca que fornece o recurso de navegação do react:

```
> npm install --save react-navigation
```

Nossa aplicação possuirá várias telas, temos que criar o mecanismo que permitirá a navegação entre estas telas, para isso, primeiro criaremos um novo componente que irá centralizar nosso fluxo de navegação.

```
// src/CoponeyMobNav.js
import React from "react";
import { StyleSheet, View } from "react-native";
import { createStackNavigator } from "react-navigation";
import ListarPoneysScreen from "./components/ListarPoneysScreen";

const RootStack = createStackNavigator(
```

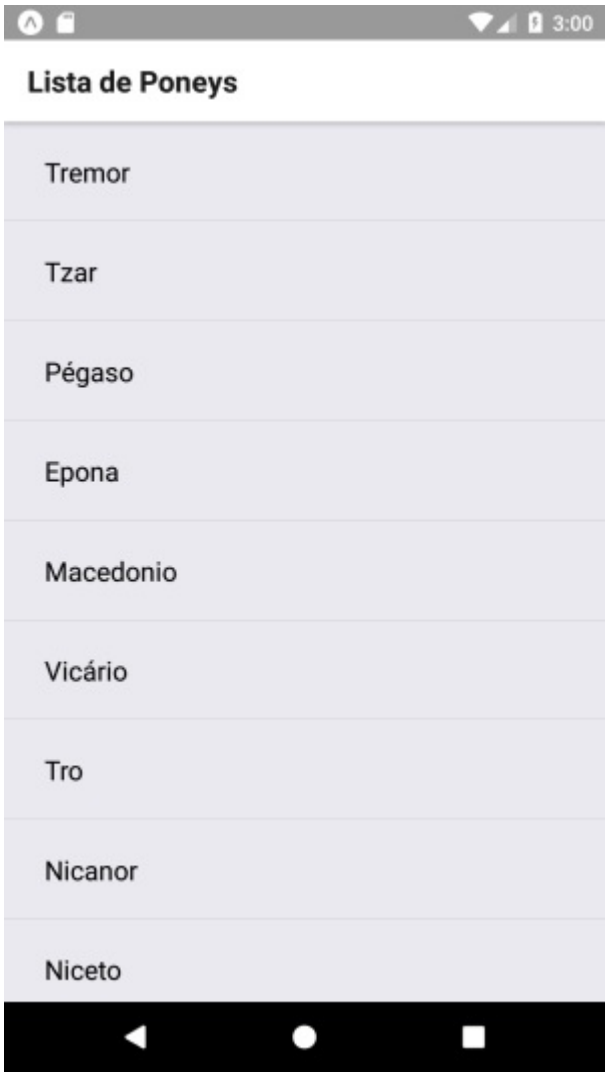
```
  {
    ListarPoneys: {
      screen: ListarPoneysScreen,
      navigationOptions: {
        title: "Lista de Poneys"
      }
    }
  },
  {
    initialRouteName: "ListarPoneys"
  }
);

class CoponeyMobNav extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <View style={styles.container}>
        <RootStack />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1
  }
});

export default CoponeyMobNav;
```

Em seguida, vamos alterar nosso componente principal para que exiba o componente de navegação:

```
// src/CoponeyMob.js
// Código anterior omitido
  render() {
    if (!this.state.isReady) {
      return (
        <View style={styles.loadingContainer}>
          <Spinner />
        </View>
      );
    } else {
```

```
      // Novidade aqui
      return <CoponeyMobNav />;
    }
  }
// Código posterior omitido
```

A tela principal da aplicação deve estar com esta aparência:



Passando o controle de estado para o Redux

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

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

Agora vamos criar nosso primeiro reducer:

```
// src/reducers/poneys.js
let poneys = [
  { nome: "Tremor" },
  { nome: "Tzar" },
  { nome: "Pégaso" },
  { nome: "Epona" },
  { nome: "Macedonio" },
  { nome: "Vicário" },
  { nome: "Tro" },
  { nome: "Nicanor" },
  { nome: "Niceto" },
  { nome: "Odón" },
  { nome: "Relâmpago" },
  { nome: "Pio" },
  { nome: "Elegante" },
  { nome: "Pompeu" }
];

poneys = poneys.map((p, idx) => ({ ...p, _id: idx + "" }));

const initialState = { list: poneys, viewDeleted: false };

export default function poneysReducer(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 criaremos:

```
// src/reducers/index.js
import { combineReducers } from "redux";
import poneysReducer from "./poneys";

const rootReducer = combineReducers({
  poneys: poneysReducer
});

export default rootReducer;
```

Em seguira, criaremos um arquivo js que tem a função criar uma 'store' a partir dos 'reducuers':

```
// src/configureStore.js.js
import { createStore } from "redux";
```

```
import rootReducer from "./reducers";

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

O próximo passo é configurar nosso componente principal para disponibilizar a 'store' em nossa árvore de componentes:

```
// src/CoponeyMob.js
import { Spinner } from "native-base";
import React from "react";
import { StyleSheet, View } from "react-native";

// Novidades aqui
import { Provider } from "react-redux";
import configureStore from "./configureStore";

import Reactotron from "reactotron-react-native";
import CoponeyMobNav from "./CoponeyMobNav";
import "./ReactotronConfig";

Reactotron.log("Testando a conexão com o Reactotron.");

// Novidades aqui
const store = configureStore();

export default class CoponeyMob extends React.Component {
  // Código atual omitido

  render() {
    if (!this.state.isReady) {
      return (
        <View style={styles.loadingContainer}>
          <Spinner />
        </View>
      );
    } else {
      return (
        // Novidades aqui
        <Provider store={store}>
          <CoponeyMobNav />
        </Provider>
      );
    }
  }
}
```

```
// Código posterior omitido
```

Nossa tela de listagem de poneys deverá então obter a lista de poneys a partir do redux:

```
// src/CoponeyMob.js
import { Left, ListItem, Text } from "native-base";

// Novidade aqui
import PropTypes from "prop-types";

import React from "react";
import { FlatList, StyleSheet, View } from "react-native";

// Novidade aqui
import { connect } from "react-redux";

class ListarPoneysScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <View>
        <FlatList
         {/* Novidade aqui */}
          data={this.props.poneys.list}
          renderItem={({ item }) => (
            <ListItem noIndent>
              <Left>
                <Text style={styles.item}>{item.nome}</Text>
              </Left>
            </ListItem>
          )}
          keyExtractor={item => item._id}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  itemContainer: {
    flex: 1,
    flexDirection: "row",
    borderBottomWidth: 1
  },
```

```
  item: {
    padding: 10,
    fontSize: 18,
    height: 44,
    width: 120
  }
});

// Várias novidades aqui
const mapStateToProps = state => {
  return {
    poneys: state.poneys
  };
};

const mapDispatchToProps = {};

ListarPoneysScreen.propTypes = {
  poneys: PropTypes.object
};

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

PropTypes

Conforme seu aplicativo cresce, você pode começar a enfrentar muitos bugs devido a problemas com tipos. Você pode usar extensões do JavaScript como Flow ou TypeScript para facilitar o trabalho com tipos em um projeto, porém, mesmo que você não os use, o React possui algumas ferramentas que permitem a checagem de tipos.

PropTypes possui uma gama de validadores que podem ser usados para garantir que os dados recebidos sejam válidos. Quando um valor inválido é fornecido para um prop, um aviso será mostrado no console do JavaScript. Por motivos de desempenho, propTypes é verificado apenas no modo de desenvolvimento.

Botão para se Logar no sistema

Nosso botão de login/logout ficará no canto superior direito da aplicação, como haverão mais botões neste local, vamos criar um componente para contê-los e iniciar nossa lógica de login:

```
// src/components/HeaderButtonsComponent.js
import {
  Button,
```

```
  Form,
  Icon,
  Input,
  Item,
  Label,
  Text,
  View
} from "native-base";
import React from "react";
import { Alert, Image, Modal, StyleSheet } from "react-native";

class HeaderButtonsComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      modalVisible: false,
      inputUserName: "",
      inputPassword: "",
      user: null
    };
  }

  closeLoginModal = () => {
    this.setState({ modalVisible: false });
  };

  openLoginModal = () => {
    this.setState({ modalVisible: true });
  };

  login = user => {
    this.setState({
      user
    });
  };

  logout = () => {
    this.setState({
      user: null
    });
  };

  handleLogin = () => {
    this.closeLoginModal();
    if (
      this.state.inputUserName === "admin" &&
      this.state.inputPassword === "123456"
    ) {
      this.login({ name: this.state.inputUserName });
```

```
    } else {
      Alert.alert("Erro", "Credenciais inválidas", [{ text: "OK" }]);
    }
  };

  handleLogout = () => {
    Alert.alert(
      "Logout",
      this.state.user.name + ", confirma o logout?",
      [{ text: "Sim", onPress: this.logout }, { text: "Não", style: "cancel" }],
      { cancelable: false }
    );
  };

  renderLoginModal = () => {
    return (
      <Modal
        animationType="slide"
        visible={this.state.modalVisible}
        onRequestClose={this.closeLoginModal}
      >
        <View>
          <Form>
            <Item floatingLabel>
              <Label>Usuário</Label>
              <Input
                autoCapitalize="none"
                onChangeText={inputUserName => this.setState({ inputUserName })}
                value={this.state.inputUserName}
              />
            </Item>
            <Item floatingLabel last style={{ marginBottom: 20 }}>
              <Label>Senha</Label>
              <Input
                secureTextEntry={true}
                autoCapitalize="none"
                onChangeText={inputPassword => this.setState({ inputPassword })}
                value={this.state.inputPassword}
              />
            </Item>
            <Button
              full
              primary
              style={{ marginBottom: 20 }}
              onPress={this.handleLogin}
            >
              <Text>Login</Text>
            </Button>
            <Button full light onPress={this.closeLoginModal}>
```

```
              <Text>Cancelar</Text>
            </Button>
          </Form>
        </View>
      </Modal>
    );
  };

  render() {
    return (
      <View style={styles.headerButtonContainer}>
        {this.state.user ? (
          <View style={styles.headerButtonContainer}>
            <Button transparent onPress={this.handleLogout}>
              <Image
                style={styles.headerIconMargin}
                source={require("../assets/admin.png")}
              />
            </Button>
          </View>
        ) : (
          <Button transparent onPress={this.openLoginModal}>
            <Icon
              style={[styles.headerIconFont, styles.headerIconMargin]}
              name="contact"
            />
          </Button>
        )}
        {this.renderLoginModal()}
      </View>
    );
  }
}

const styles = StyleSheet.create({
  headerButtonContainer: {
    flex: 1,
    flexDirection: "row"
  },
  headerIconMargin: {
    marginLeft: 10,
    marginRight: 10
  },
  headerIconFont: {
    fontSize: 35
  }
});

export default HeaderButtonsComponent;
```

O próximo passo é alterar o componente CoponeyMobNav para que ele exiba o botão HeaderButtonsComponent no cabeçalho da tela de listagem de poneis:

```
// src/CoponeyMobNav.js
import React from "react";
import { StyleSheet, View } from "react-native";
import { createStackNavigator } from "react-navigation";
import ListarPoneysScreen from "./components/ListarPoneysScreen";

// Novidade aqui
import HeaderButtonsComponent from "./components/HeaderButtonsComponent";

const RootStack = createStackNavigator(
  {
    ListarPoneys: {
      screen: ListarPoneysScreen,
      navigationOptions: {
        title: "Lista de Poneys",

        // Novidade aqui
        headerRight: <HeaderButtonsComponent />
      }
    }
  },
  {
    initialRouteName: "ListarPoneys"
  }
);
// Código posterior omitido
```

Mantento o estado de autenticação no Redux

Nossa próxima missão será exibir os botões de inclusão, alteração e exclusão de poneis, mas estes botões só podem ser exibidos se o usuário estiver logado, para facilitar que a informação do usuário autenticado esteja disponível em toda a nossa aplicação, devemos manter estas informações de autenciação em um estado gerido pelo redux.

Vamos criar um reducer que será responsável por gerenciar o profile do usuário logado:

```
// src/reducers/profile.js
import { LOGIN, LOGOUT } from "../constants";

const initialState = {};

export default function profileReducer(state = initialState, action) {
```

```
  switch (action.type) {
    case LOGIN:
      return {
        user: action.data
      };
    case LOGOUT:
      return {};
    default:
      return state;
  }
}
```

Mas também teremos que criar um arquivo com as constantes que representam as ações:

```
// src/constants.js
const LOGIN = "LOGIN";
const LOGOUT = "LOGOUT";

export { LOGIN, LOGOUT };
```

Não podemos nos esquecer de alterar o arquivo de index.js que cria o `rootReducer` incluindo o `profileReducer` que criamos:

```
// src/reducers/index.js
import { combineReducers } from "redux";
import poneysReducer from "./poneys";
import profileReducer from "./profile";

const rootReducer = combineReducers({
  profile: profileReducer,
  poneys: poneysReducer
});

export default rootReducer;
```

Em seguida, criaremos um arquivo contendo 'actions' que retornam objetos que serão utilizados pelo Redux e representam as ações que são realizadas sobre os estados:

```
// src/actions.js
import { LOGIN, LOGOUT } from "./constants";

export function login(data) {
  return {
    type: LOGIN,
    data
  };
```

```
}

export function logout() {
  return {
    type: LOGOUT
  };
}
```

Pronto, agora já está tudo preparado para que o componente 'HederButtonsComponent' se ligue aos estados e ações do Redux:

```
// src/components/HeaderButtonsComponent.js
// Código anterior omitido
import {
  Button,
  Form,
  Icon,
  Input,
  Item,
  Label,
  Text,
  View
} from "native-base";
import React from "react";
import { Alert, Image, Modal, StyleSheet } from "react-native";

// Novidades aqui
import PropTypes from "prop-types";
import { connect } from "react-redux";
import { bindActionCreators } from "redux";
import { login, logout } from "../actions";

class HeaderButtonsComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      modalVisible: false,
      inputUserName: "admin",
      inputPassword: "123456"
    };
  }

  closeLoginModal = () => {
    this.setState({ modalVisible: false });
  };

  openLoginModal = () => {
    this.setState({ modalVisible: true });
```

```
  };

  handleLogin = () => {
    this.closeLoginModal();
    if (
      this.state.inputUserName === "admin" &&
      this.state.inputPassword === "123456"
    ) {
      // Novidade aqui
      this.props.login({ name: this.state.inputUserName });
    } else {
      Alert.alert("Erro", "Credenciais inválidas", [{ text: "OK" }]);
    }
  };

  handleLogout = () => {
    Alert.alert(
      "Logout",

      // Novidade aqui
      this.props.profile.user.name + ", confirma o logout?",
      [
        // Novidade aqui
        { text: "Sim", onPress: this.props.logout },

        { text: "Não", style: "cancel" }
      ],
      { cancelable: false }
    );
  };

  // Código atual omitido

  render() {
    return (
      <View style={styles.headerButtonContainer}>
        {/* Novidades aqui */}
        {this.props.profile.user ? (
          <View style={styles.headerButtonContainer}>
            <Button transparent onPress={this.handleLogout}>
              <Image
                style={styles.headerIconMargin}
                source={require("../assets/admin.png")}
              />
            </Button>
          </View>
        ) : (
          <Button transparent onPress={this.openLoginModal}>
            <Icon
```

```
              style={[styles.headerIconFont, styles.headerIconMargin]}
              name="contact"
            />
          </Button>
        )}
        {this.renderLoginModal()}
      </View>
    );
  }
}

const styles = StyleSheet.create({
  headerButtonContainer: {
    flex: 1,
    flexDirection: "row"
  },
  headerIconMargin: {
    marginLeft: 10,
    marginRight: 10
  },
  headerIconFont: {
    fontSize: 35
  }
});

// Novidades aqui
HeaderButtonsComponent.propTypes = {
  profile: PropTypes.object,
  poneys: PropTypes.object
};

const mapStateToProps = state => {
  return {
    profile: state.profile,
    poneys: state.poneys
  };
};

const mapDispatchToProps = dispatch =>
  bindActionCreators({ login, logout }, dispatch);

HeaderButtonsComponent.propTypes = {
  login: PropTypes.func,
  logout: PropTypes.func
};

export default connect(
  mapStateToProps,
  mapDispatchToProps
```

```
)(HeaderButtonsComponent);
```

Botão pra Incluir Novo Poney

Agora que já temos o Redux controlando o estado da autenticação, nosso próximo passo é exibir um botão de inclusão de ponei caso o usuário esteja logado:

```
// src/components/HeaderButtonsComponent.js
// Código anterior omitido
  render() {
    return (
      <View style={styles.headerButtonContainer}>
        {this.props.profile.user ? (
          <View style={styles.headerButtonContainer}>
            {/* Novidade aqui */}
            <Button transparent>
              <Icon
                style={[styles.headerIconFont, styles.headerIconMargin]}
                name="add"
                onPress={() =>
                  Alert.alert("Incluir", "Aqui irá a tela de incluir ponei", [
                    { text: "OK" }
                  ])
                }
              />
            </Button>
            <Button transparent onPress={this.handleLogout}>
              <Image
                style={styles.headerIconMargin}
                source={require("../assets/admin.png")}
              />
            </Button>
          </View>
        ) : (
          <Button transparent onPress={this.openLoginModal}>
            <Icon
              style={[styles.headerIconFont, styles.headerIconMargin]}
              name="contact"
            />
          </Button>
        )}
        {this.renderLoginModal()}
      </View>
    );
  }
// Código posterior omitido
```

Botão pra Editar e Excluir Ponei

Temos mais dois botões a serem colocados na tela, mas estes vão ficar ao lado de cada ponei na listagem, são os botões de editar e excluir:

```
// src/components/ListarPoneysScreen.js
// Código anterior omitido
  render() {
    return (
      <View>
        <FlatList
          data={this.props.poneys.list}
          renderItem={({ item }) => (
            <ListItem noIndent>
              <Left>
                <Text style={styles.item}>{item.nome}</Text>
              </Left>
              <Right>
                {this.props.profile.user && (
                  <View style={{ flexDirection: "row", flex: 1 }}>
                    <Button
                      primary
                      onPress={() =>
                        Alert.alert(
                          "Alterar",
                          "Aqui irá a tela de Alterar ponei",
                          [{ text: "OK" }]
                        )
                      }
                      style={{ marginRight: 10 }}
                    >
                      <Icon name="create" />
                    </Button>
                    <Button
                      danger
                      onPress={() =>
                        Alert.alert(
                          "Excluir",
                          "Aqui exibirá a confirmação de exclusão do ponei",
                          [{ text: "OK" }]
                        )
                      }
                    >
                      <Icon name="trash" />
                    </Button>
                  </View>
                )}
              </Right>
```

```
            </ListItem>
          )}
          keyExtractor={item => item._id}
        />
      </View>
    );
  }
}

// Código atual omitido

const mapStateToProps = state => {
  return {
    poneys: state.poneys,

    // Novidade aqui
    profile: state.profile
  };
};

const mapDispatchToProps = {};

ListarPoneysScreen.propTypes = {
  poneys: PropTypes.object,

  // Novidade aqui
  profile: PropTypes.object
};

// Código posterior omitido
```

Mas apesar de nossa lógica estar aparentemente correta, os botões não são exibidos após o usuário se logar... Para que isso funcione, precisamos "avisar" a lista que ela precisa se atualizar:

```
// src/components/ListarPoneysScreen.js
// Código anterior omitido
  render() {
    return (
      <View>
        <FlatList
          data={this.props.poneys.list}
          extraData={this.props.profile}
          renderItem={({ item }) => (
            <ListItem noIndent>
              <Left>
                <Text style={styles.item}>{item.nome}</Text>
```

```
                </Left>
// Código posterior omitido
```

Mapeamento do Serviço Rest para Listagem dos Poneys

O serviço REST que servirá de backend para nossa aplicação já se encontra pronto, sua URL base é https://coponeyapi.herokuapp.com. Os seguintes end-points estão disponíveis:

- GET/POST: https://coponeyapi.herokuapp.com/v1/poneys
- GET/PUT/DELETE: https://coponeyapi.herokuapp.com/v1/poneys/{id}

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

O código de acionamento de nossa API ficará isolado em uma pasta 'api' dentro de 'src', mas para realizar os requests usaremos a biblioteca 'superagent', vamos instalá-la em nosso projeto:

```
> npm install --save superagent
```

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

```
// src/api/index.js
import request from "superagent";

// const URI = "http://localhost:3000/v1/poneys";
const URI = "https://coponeyapi.herokuapp.com/v1/poneys";

export function loadPoneysAPI() {
  return request.get(URI).set("Accept", "application/json");
}
```

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

```
// src/actions.js
import { loadPoneysAPI } from "./api";
import { LOAD_PONEYS, LOGIN, LOGOUT } from "./constants";

export function loadPoneys() {
  loadPoneysAPI()
    .then(res => ({
      type: LOAD_PONEYS,
      data: res.body
    }))
    .catch(error => {
      alert(error.message);
    });
```

```
}

// Código posterior omitido
```

Devemos também ajustar o arquivo de constantes para incluir a nova constante 'LOAD_PONEYS':

```
// src/constants.js
const LOGIN = "LOGIN";
const LOGOUT = "LOGOUT";
const LOAD_PONEYS = "LOAD_PONEYS";

export { LOGIN, LOGOUT, LOAD_PONEYS };
```

Agora, vamos ajustar nosso store de poneis para acertar a ação 'LOAD_PONEYS':

```
// src/reducers/poneys.js
import { LOAD_PONEYS } from "../constants";

const initialState = { list: [], viewDeleted: false };

export default function poneysReducer(state = initialState, action) {
  switch (action.type) {
    case LOAD_PONEYS:
      return {
        ...state,
        list: [...action.data]
      };
    default:
      return state;
  }
}
```

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

```
// src/components/ListaPoneysScreen.js
import { Button, Icon, Left, ListItem, Right, Text } from "native-base";
import PropTypes from "prop-types";
import React from "react";
import { Alert, FlatList, StyleSheet, View } from "react-native";
import { connect } from "react-redux";

// Novidade aqui
import { bindActionCreators } from "redux";
import { loadPoneys } from "../actions";
```

```
class ListarPoneysScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  // Novidade aqui
  componentDidMount() {
    this.props.loadPoneys();
  }

  // Código atual omitido
}

// Código atual omitido

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

ListarPoneysScreen.propTypes = {
  poneys: PropTypes.object,
  profile: PropTypes.object,

  // Novidade aqui
  loadPoneys: PropTypes.func
};

// Código posterior omitido
```

Tudo parece certo, porém quando tentamos executar o código, recebemos a seguinte tela de erro:

Ou seja, uma ação deve retornar um "objeto plano" ou devemos utilizar um middleware customizado, neste caso, vamos utilizar o middleware customizado chamado de `redux-thunk`, primeiro, devemos instalá-lo:

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

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

```
// src/configureStore.js.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`:

```
// src/actions.js
import { loadPoneysAPI } from "./api";
import { LOAD_PONEYS, LOGIN, LOGOUT } from "./constants";

export function loadPoneys() {
  return dipatch => {
    loadPoneysAPI()
      .then(res => {
        dipatch({
          type: LOAD_PONEYS,
          data: res.body
        });
      })
      .catch(error => {
        console.log(error);
      });
  };
}
// Código posterior omitido
```

Extra: Filtrar poneys por status (Excluido: True/False)

Se fizermos `console.log` da resposta veremos que na verdade vários dos poneis sofreram exclusão lógica e mesmo assim estão sendo exibidos, precisamos filtrá-los na listagem, assim como criar um botão que exiba somente os excluídos ou somente os não excluídos:

```
// src/actions.js
// Código anterior omitido
export function loadPoneys() {
  return dipatch => {
    loadPoneysAPI()
      .then(res => {
        // Novidade aqui
        console.log(">>>>>>>>>>>>>>>>>>");
        console.log(res.body);

        dipatch({
          type: LOAD_PONEYS,
          data: res.body
        });
      })
      .catch(error => {
        alert(error.message);
      });
```

```
  };
}
// Código posterior omitido
```

A resposta será algo como:

```
>>>>>>>>>>>>>>>>>
Array [
  Object {
    "_id": "AUqbjNFR17LqwOac",
    "cor": "Branca",
    "excluido": true,
    "nome": "Epona",
  },
  Object {
    "_id": "cU2BieASCEEIgF8E",
    "cor": "Preta",
    "excluido": true,
    "nome": "Pé de Pano",
  },
  Object {
    "_id": "z7wyKpalWQe7LeJz",
    "cor": "Malhada",
    "excluido": true,
    "nome": "Thor",
  },
  Object {
    "_id": "MIofwryRe6FKpO9G",
    "cor": "Baio",
    "nome": "Trovão",
  },
]
```

Vamos então colocar o botão com a funcionalidade:

```
// src/components/HeaderButtonsComponent.js
// Código anterior omitido
import { bindActionCreators } from "redux";

// Novidade aqui
import { login, logout, toggleViewDeletedPoneys } from "../actions";

class HeaderButtonsComponent extends React.Component {
  // Código omitido

  render() {
    return (
```

```
      <View style={styles.headerButtonContainer}>
        {/* Novidade aqui */}
        <Button transparent onPress={this.props.toggleViewDeletedPoneys}>
          <Icon
            style={[styles.headerIconFont, styles.headerIconMargin]}
            name={this.props.poneys.viewDeleted ? "eye-off" : "eye"}
          />
        </Button>

        {this.props.profile.user ? (
          <View style={styles.headerButtonContainer}>
            <Button transparent>
              <Icon
                style={[styles.headerIconFont, styles.headerIconMargin]}
                name="add"
                onPress={() =>
                  Alert.alert("Incluir", "Aqui irá a tela de incluir ponei", [
                    { text: "OK" }
                  ])
                }
              />
            </Button>
            <Button transparent onPress={this.handleLogout}>
              <Image
                style={styles.headerIconMargin}
                source={require("../assets/admin.png")}
              />
            </Button>
          </View>
        ) : (
          <Button transparent onPress={this.openLoginModal}>
            <Icon
              style={[styles.headerIconFont, styles.headerIconMargin]}
              name="contact"
            />
          </Button>
        )}
        {this.renderLoginModal()}
      </View>
    );
  }
}

// Código omitido

HeaderButtonsComponent.propTypes = {
  profile: PropTypes.object,
  poneys: PropTypes.object,
```

```
  // Novidade aqui
  toggleViewDeletedPoneys: PropTypes.func
};

const mapStateToProps = state => {
  return {
    profile: state.profile,
    poneys: state.poneys
  };
};

const mapDispatchToProps = dispatch =>
  // Novidade aqui
  bindActionCreators({ login, logout, toggleViewDeletedPoneys }, dispatch);

// Código posterior omitido
```

Agora, temos que criar uma constante para uma nova ação de habilitar e desabilitar a exibição de poneys ocultos, assim como tratar esta ação no reducer e criar uma action correspondente:

```
// src/constants.js
const LOGIN = "LOGIN";
const LOGOUT = "LOGOUT";
const LOAD_PONEYS = "LOAD_PONEYS";

// Novidade aqui
const TOGGLE_VIEW_DELETED_PONEYS = "TOGGLE_VIEW_DELETED_PONEYS";

// Novidade aqui
export { LOGIN, LOGOUT, LOAD_PONEYS, TOGGLE_VIEW_DELETED_PONEYS };
```

```
// src/reducers/poneys.js
// Novidade aqui
import { LOAD_PONEYS, TOGGLE_VIEW_DELETED_PONEYS } from "../constants";

const initialState = { list: [], viewDeleted: false };

export default function poneysReducer(state = initialState, action) {
  switch (action.type) {
    case LOAD_PONEYS:
      return {
        ...state,
        list: [...action.data]
      };

    // Novidade aqui
```

```
    case TOGGLE_VIEW_DELETED_PONEYS:
      return {
        ...state,
        viewDeleted: !state.viewDeleted
      };
    default:
      return state;
  }
}
```

```
// src/actions.js
// Código anterior omitido
import { loadPoneysAPI } from "./api";
import {
  LOAD_PONEYS,
  LOGIN,
  LOGOUT,

  // Novidade aqui
  TOGGLE_VIEW_DELETED_PONEYS
} from "./constants";

// Código omitido

// Novidade aqui
export function toggleViewDeletedPoneys() {
  return {
    type: TOGGLE_VIEW_DELETED_PONEYS
  };
}
// Código posterior omitido
```

E o último passo é responder a esta nova propriedade do nosso estado na view de listagem de poneys:

```
// src/components/ListarPoneysScreen.js
// Código anterior omitido
class ListarPoneysScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  componentDidMount() {
    this.props.loadPoneys();
  }
```

```
  render() {

    // Novidades aqui
    let { poneys } = this.props;

    let listExibicao = poneys.list.filter(
      p => this.props.poneys.viewDeleted === !!p.excluido
    );

    return (
      <View>
        <FlatList

          {/* Novidades aqui */}
          data={listExibicao}

          extraData={this.props.profile}
          renderItem={({ item }) => (
            <ListItem noIndent>
              <Left>
                <Text
                  {/* Novidades aqui */}
                  style={[styles.item, item.excluido ? styles.tachado : ""]}
                >
                  {item.nome}
                </Text>
              </Left>
              {/* Código omitido */}
            </ListItem>
          )}
          keyExtractor={item => item._id}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  itemContainer: {
    flex: 1,
    flexDirection: "row",
    borderBottomWidth: 1
  },
  item: {
    padding: 10,
    fontSize: 18,
    height: 44,
    width: 120
  },
```

```
  // Novidades aqui
  tachado: {
    textDecorationLine: "line-through",
    textDecorationStyle: "solid"
  }
});

// Código posterior omitido
```

Mas agora temos um dilema, não seria mais correto ocultar os botões de edição e exclusão de poneis para os poneis que já foram excluídos?

Vamos fazer isso:

```
// src/components/ListarPoneysScreen.js
// Código anterior omitido
class ListarPoneysScreen extends React.Component {
  // Código omitido

  render() {
    // Código omitido

    return (
      <View>
        <FlatList
          data={listExibicao}
          extraData={this.props.profile}
          renderItem={({ item }) => (
            <ListItem noIndent>
              <Left>
                <Text
                  style={[styles.item, item.excluido ? styles.tachado : ""]}
                >
                  {item.nome}
                </Text>
              </Left>
              <Right>
                {/* Novidades aqui */}
                {this.props.profile.user && !item.excluido && (
                  <View style={{ flexDirection: "row", flex: 1 }}>
                    <Button
                      primary
                      onPress={() =>
                        Alert.alert(
                          "Alterar",
                          "Aqui irá a tela de Alterar ponei",
                          [{ text: "OK" }]
```

```
                        )
                      }
                      style={{ marginRight: 10 }}
                    >
                      <Icon name="create" />
                    </Button>
                    <Button
                      danger
                      onPress={() =>
                        Alert.alert(
                          "Excluir",
                          "Aqui exibirá a confirmação de exclusão do ponei",
                          [{ text: "OK" }]
                        )
                      }
                    >
                      <Icon name="trash" />
                    </Button>
                  </View>
                )}
              </Right>
            </ListItem>
          )}
          keyExtractor={item => item._id}
        />
      </View>
    );
  }
}

// Código posterior omitido
```

Criando a tela de Cadastro de Poneys

Criando rota para a tela

Nossa próxima tarefa é criar um formulário que permita o cadastramento de novos poneis no sistema, vamos criar o componente:

```
// src/components/AdicionarPoneyScreen.js
import { Text } from "native-base";
import React from "react";
import { connect } from "react-redux";

class AdicionaPoneyScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return <Text>Tela de Inclusão de Poneis</Text>;
  }
}

const mapStateToProps = {};

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

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

Temos também que adicionar um roteamento para esta tela:

```
// src/CoponeyMobNav.js
// Código anterior omitido
import AdicionarPoneyScreen from "./components/AdicionarPoneyScreen";

const RootStack = createStackNavigator(
  {
    ListarPoneys: {
      screen: ListarPoneysScreen,
      navigationOptions: {
        title: "Lista de Poneys",
        headerRight: <HeaderButtonsComponent />
      }
```

```
    },

    // Novidade aqui
    AdicionarPoney: {
      screen: AdicionarPoneyScreen,
      navigationOptions: {
        title: "Adicionar Poney"
      }
    }
  },
  {
    initialRouteName: "ListarPoneys"
  }
);
// Código posterior omitido
```

Agora temos que "navegar" de fato da tela de listagem para a tela que adiciona os poneys pelo evento de click do usuário no botão de inclusão de ponei que fica no componente de cabeçalho:

```
// src/components/HeaderButtonsComponent.js
// Código anterior omitido
render() {
  return (
    <View style={styles.headerButtonContainer}>
      <Button transparent onPress={this.props.toggleViewDeletedPoneys}>
        <Icon
          style={[styles.headerIconFont, styles.headerIconMargin]}
          name={this.props.poneys.viewDeleted ? "eye-off" : "eye"}
        />
      </Button>
      {this.props.profile.user ? (
        <View style={styles.headerButtonContainer}>
          <Button transparent>
            <Icon
              style={[styles.headerIconFont, styles.headerIconMargin]}
              name="add"

              {/* Novidade aqui */}
              onPress={() => this.props.navigation.navigate("AdicionarPoney")}
            />
          </Button>
          <Button transparent onPress={this.handleLogout}>
            <Image
              style={styles.headerIconMargin}
              source={require("../assets/admin.png")}
            />
          </Button>
        </View>
```

```
      ) : (
        <Button transparent onPress={this.openLoginModal}>
          <Icon
            style={[styles.headerIconFont, styles.headerIconMargin]}
            name="contact"
          />
        </Button>
      )}
      {this.renderLoginModal()}
    </View>
  );
}

// Código omitido


HeaderButtonsComponent.propTypes = {
  profile: PropTypes.object,
  poneys: PropTypes.object,
  toggleViewDeletedPoneys: PropTypes.func,

  // Novidade aqui
  navigation: PropTypes.object
};

// Código posterior omitido
```

Mas este objeto navigation tem que ser disponibilizado para o nosso componente, e isso é feito alterando-se o componente de navegação:

```
// src/CoponeyMobNav.js
// Código anterior omitido
import AdicionarPoneyScreen from "./components/AdicionarPoneyScreen";

const RootStack = createStackNavigator(
  {
    // Novidade aqui
    ListarPoneys: {
      screen: ListarPoneysScreen,
      navigationOptions: ({ navigation }) => ({
        title: "Lista de Poneys",
        headerRight: <HeaderButtonsComponent navigation={navigation} />
      })
    },

    AdicionarPoney: {
      screen: AdicionarPoneyScreen,
      navigationOptions: {
```

```
        title: "Adicionar Poney"
      }
    }
  },
  {
    initialRouteName: "ListarPoneys"
  }
);
// Código posterior omitido
```

Criando o formulário de inclusão de poneis

Agora que já temos um componente pronto para ser nossa tela de inclusão de poneis e o roteamento também já está funcionando, é hora de começarmos a trabalhar no formulário, uma boa estratégia é criar um componente separado para o formulário de modo que ele possa ser reaproveitado para o fluxo de alteração de poneis.

Para facilitar o processo de validação do formulário, assim como o controle dos dados impostados, utilizaremos a biblioteca `redux-form`, logo, o primeiro passo é instalá-la:

```
> npm install --save redux-form
```

```
// src/CoponeyMob.js
import { Button, Text, Item, Label, Input, Picker, Icon } from "native-base";
import React from "react";
import { View } from "react-native";
import { Field, reduxForm } from "redux-form";

const validate = values => {
  const error = {
    nome: "",
    cor: ""
  };
  let nome = values.nome || "";
  let cor = values.cor || "";
  if (nome.length < 3) {
    error.nome = "Muito curto";
  }
  if (nome.length > 8) {
    error.nome = "Máximo de 8 caracteres";
  }
  if (!cor) {
    error.cor = "Cor inválida";
  }
  return error;
```

```
};

class MantemPoneyForm extends React.Component {
  renderInput = ({ input, label, meta: { error } }) => {
    var hasError = false;
    if (error !== undefined) {
      hasError = true;
    }
    return (
      <Item floatingLabel error={hasError}>
        <Label>
          {label} {hasError && " - " + error}
        </Label>
        <Input {...input} />
      </Item>
    );
  };

  renderPicker = ({
    input: { onChange, value, ...inputProps },
    ...pickerProps
  }) => {
    return (
      <Item picker>
        <Picker
          selectedValue={value}
          onValueChange={value => onChange(value)}
          {...inputProps}
          {...pickerProps}
          mode="dropdown"
          iosIcon={<Icon name="ios-arrow-down-outline" />}
          style={{ width: undefined }}
          placeholder="Selecione a cor"
          placeholderStyle={{ color: "#bfc6ea" }}
          placeholderIconColor="#007aff"
        >
          <Picker.Item label="Selecione a cor" value="" />
          <Picker.Item label="Amarelo" value="Amarelo" />
          <Picker.Item label="Baio" value="Baio" />
          <Picker.Item label="Branco" value="Branco" />
          <Picker.Item label="Preto" value="Preto" />
        </Picker>
      </Item>
    );
  };

  render() {
    const { invalid, handleSubmit } = this.props;
    return (
```

```
      <View style={{ marginTop: 10 }}>
        <Field name="nome" label="Nome" component={this.renderInput} />
        <Field name="cor" label="Cor" component={this.renderPicker} />
        <Button
          disabled={invalid}
          bordered={invalid}
          full
          primary
          style={{ marginBottom: 20 }}
          onPress={handleSubmit}
        >
          <Text>Salvar</Text>
        </Button>
        <Button
          full
          warning
          onPress={() => this.props.navigation.navigate("ListarPoneys")}
        >
          <Text>Cancelar</Text>
        </Button>
      </View>
    );
  }
}

export default reduxForm({
  form: "mantemPonei",
  validate
})(MantemPoneyForm);
```

Para que o `redux-form` funcione, precisamos incluí-lo como um reducer quando criamos a nossa store:

```
// src/reducers/index.js
import { combineReducers } from "redux";
import poneysReducer from "./poneys";
import profileReducer from "./profile";

// Novidade aqui
import { reducer as formReducer } from "redux-form";

const rootReducer = combineReducers({
  profile: profileReducer,
  poneys: poneysReducer,

  // Novidade aqui
  form: formReducer
});
```

```
export default rootReducer;
```

Quando incluimos o componente do formulário que criamos, é necessário passar uma função como parâmetro `handleSubmit` que irá receber o evento de submissão juntamente com os dados do ponei que foi cadastrado. Vamos ajustar agora o componente `AdicionaPoneyScreen` para que ele incorpore o formulário de poneis:

```
// src/components/AdicionarPoneyScreen.js
import React from "react";
import { connect } from "react-redux";

// Novidade aqui
import PropTypes from "prop-types";
import MantemPoneyForm from "./MantemPoneyForm";

class AdicionaPoneyScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  // Novidade aqui
  handleAddPoney = poney => {
    alert("Ponei adicionado: " + JSON.stringify(poney));
    this.props.navigation.navigate("ListarPoneys");
  };

  // Novidade aqui
  render() {
    return (
      <MantemPoneyForm
        navigation={this.props.navigation}
        onSubmit={this.handleAddPoney}
      />
    );
  }
}

// Novidade aqui
AdicionaPoneyScreen.propTypes = {
  navigation: PropTypes.object
};

// Código posterior omitido
```

Mapeamento do Serviço Rest para Inclusão de Ponei

Agora que já temos um mapeamento e um formulário funcional para cadastramento de poneis, precisamos nos comunicar com o back-end para que seja feita a persistência dos dados.

Como sempre, vamos criar constantes, chamada de API, ações e alterar nosso reducer incluindo esta funcionalidade:

- Constantes:

```
// src/constants.js
const LOGIN = "LOGIN";
const LOGOUT = "LOGOUT";
const LOAD_PONEYS = "LOAD_PONEYS";
const TOGGLE_VIEW_DELETED_PONEYS = "TOGGLE_VIEW_DELETED_PONEYS";

// Novidade aqui
const ADD_PONEY = "ADD_PONEY";

// Novidade aqui
export { ADD_PONEY, LOGIN, LOGOUT, LOAD_PONEYS, TOGGLE_VIEW_DELETED_PONEYS };
```

- API:

```
// src/api/index.js
import request from "superagent";

const URI = "https://coponeyapi.herokuapp.com/v1/poneys";

export function loadPoneysAPI() {
  return request.get(URI).set("Accept", "application/json");
}

// Novidade aqui
export function addPoneyAPI(poney) {
  return request.post(URI).send(poney);
}
```

- Ações:

```
// src/actions.js
// Novidade aqui
import { loadPoneysAPI, addPoneyAPI } from "./api";

import {
  LOAD_PONEYS,
  LOGIN,
  LOGOUT,
  TOGGLE_VIEW_DELETED_PONEYS,
```

```
  // Novidade aqui
  ADD_PONEY
} from "./constants";

// Código omitido

export function addPoney(data) {
  return dipatch => {
    addPoneyAPI(data)
      .then(res => {
        dipatch({
          type: ADD_PONEY,
          data: { ...data, _id: res.body }
        });
      })
      .catch(error => {
        alert(error.message);
      });
  };
}

// Código posterior omitido
```

- Reducer:

```
// src/reducers/poneys.js
// Código anterior omitido
import {
  // Novidade aqui
  LOAD_PONEYS,
  TOGGLE_VIEW_DELETED_PONEYS,
  ADD_PONEY
} from "../constants";

const initialState = { list: [], viewDeleted: false };

export default function poneysReducer(state = initialState, action) {
  switch (action.type) {
    case LOAD_PONEYS:
      return {
        ...state,
        list: [...action.data]
      };
    case TOGGLE_VIEW_DELETED_PONEYS:
      return {
        ...state,
        viewDeleted: !state.viewDeleted
```

```
      };

    // Novidade aqui
    case ADD_PONEY:
      return {
        ...state,
        list: [...state.list, action.data]
      };
    default:
      return state;
  }
}
```

E por fim, vamos ligar a função do componente `AdicionarPoneyScreen` à ação que foi criada:

```
// src/components/AdicionarPoneyScreen.js
// Código anterior omitido
import PropTypes from "prop-types";
import React from "react";
import { connect } from "react-redux";
import MantemPoneyForm from "./MantemPoneyForm";

// Novidade aqui
import { bindActionCreators } from "redux";
import { addPoney } from "../actions";

class AdicionaPoneyScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  handleAddPoney = poney => {
    // Novidade aqui
    this.props.addPoney(poney);

    this.props.navigation.navigate("ListarPoneys");
  };

  render() {
    return (
      <MantemPoneyForm
        navigation={this.props.navigation}
        onSubmit={this.handleAddPoney}
      />
    );
  }
}
```

```
// Novidade aqui
AdicionaPoneyScreen.propTypes = {
  addPoney: PropTypes.func,
  navigation: PropTypes.object
};

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

// Novidade aqui
const mapDispatchToProps = dispatch =>
  bindActionCreators(
    {
      addPoney
    },
    dispatch
  );
// Código posterior omitido
```

Criando a tela de Alteração de Poneys

Criando a rota para a tela

Basicamente iremos repetir os passos que fizemos quando criamos a tela de cadastramento de poneis, primeiro, criaremos um componente que conterá o formulário que será reaproveitado:

```
// src/components/AtualizarPoneyScreen.js
import PropTypes from "prop-types";
import React from "react";
import { connect } from "react-redux";
import MantemPoneyForm from "./MantemPoneyForm";

class AtualizarPoneyScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  handleUpdatePoney = poney => {
    alert("Atualizar ponei: " + JSON.stringify(poney));
    this.props.navigation.navigate("ListarPoneys");
  };

  render() {
    return (
      <MantemPoneyForm
        initialValues={this.props.navigation.getParam("poney")}
        navigation={this.props.navigation}
        onSubmit={this.handleUpdatePoney}
      />
    );
  }
}

AtualizarPoneyScreen.propTypes = {
  updatePoney: PropTypes.func,
  navigation: PropTypes.object
};

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

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

export default connect(
  mapStateToProps,
```

```
  mapDispatchToProps
)(AtualizarPoneyScreen);
```

Agora, incluimos uma rota nova no componente de navegação:

```
// src/CoponeyMobNav.js
// Código anterior omitido

// Novidade aqui
import AtualizarPoneyScreen from "./components/AtualizarPoneyScreen";

const RootStack = createStackNavigator(
  {
    ListarPoneys: {
      screen: ListarPoneysScreen,
      navigationOptions: ({ navigation }) => ({
        title: "Lista de Poneys",
        headerRight: <HeaderButtonsComponent navigation={navigation} />
      })
    },
    AdicionarPoney: {
      screen: AdicionarPoneyScreen,
      navigationOptions: {
        title: "Adicionar Poney"
      }
    },

    // Novidade aqui
    AtualizarPoney: {
      screen: AtualizarPoneyScreen,
      navigationOptions: {
        title: "Atualizar Poney"
      }
    }
  },
  {
    initialRouteName: "ListarPoneys"
  }
);

// Código posterior omitido
```

Por fim, ajustamos a tela de listagem de poneis para que o botão de edição navege para a tela que foi criada, passando como parâmetro o ponei que será atualizado:

```
// src/components/ListarPoneysScreen.js
// Código anterior omitido
```

```
    return (
      <View>
        <FlatList
          data={listExibicao}
          extraData={this.props.profile}
          renderItem={({ item }) => (
            <ListItem noIndent>
              <Left>
                <Text
                  style={[styles.item, item.excluido ? styles.tachado : ""]}
                >
                  {item.nome}
                </Text>
              </Left>
              <Right>
                {this.props.profile.user && !item.excluido && (
                  <View style={{ flexDirection: "row", flex: 1 }}>
                    <Button
                      primary

                      {/* Novidade aqui */}
                      onPress={() =>
                        this.props.navigation.navigate("AtualizarPoney", {
                          poney: item
                        })
                      }

                      style={{ marginRight: 10 }}
                    >
                      <Icon name="create" />
                    </Button>
                    <Button
                      danger
                      onPress={() =>
                        Alert.alert(
                          "Excluir",
                          "Aqui exibirá a confirmação de exclusão do ponei",
                          [{ text: "OK" }]
                        )
                      }
                    >
                      <Icon name="trash" />
                    </Button>
                  </View>
                )}
              </Right>
            </ListItem>
          )}
          keyExtractor={item => item._id}
```

```
        />
      </View>
    );
  }
}

// Código omitido

ListarPoneysScreen.propTypes = {
  poneys: PropTypes.object,
  profile: PropTypes.object,

  // Novidades aqui
  loadPoneys: PropTypes.func,
  navigation: PropTypes.object
};

// Código posterior omitido
```

Mapeamento do Serviço Rest para Atualização do Poney

Novamente chega a hora de nos comunicarmos com o back-end para que seja feita a persistência dos dados.

Como sempre, vamos criar constantes, chamada de API, ações e alterar nosso reducer incluindo esta nova funcionalidade:

- Constantes:

```
// src/constants.js
const ADD_PONEY = "ADD_PONEY";
const LOGIN = "LOGIN";
const LOGOUT = "LOGOUT";
const LOAD_PONEYS = "LOAD_PONEYS";
const TOGGLE_VIEW_DELETED_PONEYS = "TOGGLE_VIEW_DELETED_PONEYS";

// Novidade aqui
const UPDATE_PONEY = "UPDATE_PONEY";

export {
  ADD_PONEY,
  LOGIN,
  LOGOUT,
  LOAD_PONEYS,
  TOGGLE_VIEW_DELETED_PONEYS,
  // Novidade aqui
  UPDATE_PONEY
```

```
};
```

- API:

```
// src/api/index.js
import request from "superagent";

const URI = "https://coponeyapi.herokuapp.com/v1/poneys";

export function loadPoneysAPI() {
  return request.get(URI).set("Accept", "application/json");
}

export function addPoneyAPI(poney) {
  return request.post(URI).send(poney);
}

// Novidade aqui
export function updatePoneyAPI(poney) {
  return request.put(URI + "/" + poney._id).send(poney);
}
```

- Ações:

```
// src/actions.js
// Novidade aqui
import { loadPoneysAPI, addPoneyAPI } from "./api";

import {
  LOAD_PONEYS,
  LOGIN,
  LOGOUT,
  TOGGLE_VIEW_DELETED_PONEYS,
  ADD_PONEY,

  // Novidade aqui
  UPDATE_PONEY
} from "./constants";

// Código omitido

export function updatePoney(data) {
  return dipatch => {
    updatePoneyAPI(data)
      .then(() => {
        dipatch({
          type: UPDATE_PONEY,
```

```
          data
        });
      })
      .catch(error => {
        alert(error.message);
      });
  };
}

// Código posterior omitido
```

- Reducer:

```
// src/reducers/poneys.js
// Código anterior omitido
import {
  LOAD_PONEYS,
  TOGGLE_VIEW_DELETED_PONEYS,
  ADD_PONEY,

  // Novidade aqui
  UPDATE_PONEY
} from "../constants";

const initialState = { list: [], viewDeleted: false };

export default function poneysReducer(state = initialState, action) {
  switch (action.type) {
    case LOAD_PONEYS:
      return {
        ...state,
        list: [...action.data]
      };
    case TOGGLE_VIEW_DELETED_PONEYS:
      return {
        ...state,
        viewDeleted: !state.viewDeleted
      };

    case ADD_PONEY:
      return {
        ...state,
        list: [...state.list, action.data]
      };

    // Novidade aqui
    case UPDATE_PONEY:
      return {
```

```
        ...state,
        list: state.list.map(p =>
          p._id === action.data._id ? { ...action.data } : p
        )
      };
    default:
      return state;
  }
}
```

E por fim, vamos ligar a função do componente `AtualizarPoneyScreen` à ação que foi criada:

```
// src/components/AtualizarPoneyScreen.js
// Código anterior omitido

// Novidades aqui
import { bindActionCreators } from "redux";
import { updatePoney } from "../actions";
import MantemPoneyForm from "./MantemPoneyForm";

class AtualizarPoneyScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  handleUpdatePoney = poney => {
    // Novidades aqui
    this.props.updatePoney(poney);

    this.props.navigation.navigate("ListarPoneys");
  };

  // Código omitido
}

// Código omitido

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

// Código posterior omitido
```

Exclusão lógica de Poneys

Modal para confirmar a exclusão

Já avançamos muito com nosso sistema, nossa próxima funcionalidade a ser implementada é a exclusão lógica de poneis.

Como é de praxe, o botão de exclusão irá solicitar uma confirmação ao usuário antes de proceder a exclusão de fato, vamos implementar esse recurso:

```
// src/components/ListarPoneysScreen.js
// Código anterior omitido
class ListarPoneysScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  // Novidades aqui
  handleDeletePoney = poney => {
    Alert.alert("Exclusão", `Confirma a exclusão do poney ${poney.nome}?`, [
      { text: "Sim", onPress: () => alert("Ponei excluído: " + poney.nome) },
      { text: "Não", style: "cancel" }
    ]);
  };

  componentDidMount() {
    this.props.loadPoneys();
  }

  render() {
    let { poneys } = this.props;

    let listExibicao = poneys.list.filter(
      p => this.props.poneys.viewDeleted === !!p.excluido
    );

    return (
      <View>
        <FlatList
          data={listExibicao}
          extraData={this.props.profile}
          renderItem={({ item }) => (
            <ListItem noIndent>
              <Left>
                <Text
```

```
                  style={[styles.item, item.excluido ? styles.tachado : ""]}
                >
                  {item.nome}
                </Text>
              </Left>
              <Right>
                {this.props.profile.user && !item.excluido && (
                  <View style={{ flexDirection: "row", flex: 1 }}>
                    <Button
                      primary
                      onPress={() =>
                        this.props.navigation.navigate("AtualizarPoney", {
                          poney: item
                        })
                      }
                      style={{ marginRight: 10 }}
                    >
                      <Icon name="create" />
                    </Button>

                    {/* Novidades aqui */}
                    <Button danger onPress={() => this.handleDeletePoney(item)}>
                      <Icon name="trash" />
                    </Button>
                  </View>
                )}
              </Right>
            </ListItem>
          )}
          keyExtractor={item => item._id}
        />
      </View>
    );
  }
}

// Código posterior omitido
```

Mapeamento do Serviço Rest para Exclusão do Poney

E mais uma vez precisamos nos comunicarmos com o back-end para que seja feita a exclusão dos poneis no banco de dados.

Como sempre, vamos criar constantes, chamada de API, ações e alterar nosso reducer incluindo esta nova funcionalidade:

- Constantes:

```
// src/constants.js
const ADD_PONEY = "ADD_PONEY";
const LOGIN = "LOGIN";
const LOGOUT = "LOGOUT";
const LOAD_PONEYS = "LOAD_PONEYS";
const TOGGLE_VIEW_DELETED_PONEYS = "TOGGLE_VIEW_DELETED_PONEYS";
const UPDATE_PONEY = "UPDATE_PONEY";

// Novidade aqui
const DELETE_PONEY = "DELETE_PONEY";

export {
  ADD_PONEY,
  LOGIN,
  LOGOUT,
  LOAD_PONEYS,
  TOGGLE_VIEW_DELETED_PONEYS,
  UPDATE_PONEY,
  // Novidade aqui
  DELETE_PONEY
};
```

- API:

```
// src/api/index.js
import request from "superagent";

const URI = "https://coponeyapi.herokuapp.com/v1/poneys";

export function loadPoneysAPI() {
  return request.get(URI).set("Accept", "application/json");
}

export function addPoneyAPI(poney) {
  return request.post(URI).send(poney);
}

export function updatePoneyAPI(poney) {
  return request.put(URI + "/" + poney._id).send(poney);
}

// Novidade aqui
export function deletePoneyAPI(id) {
  return request.delete(URI + "/exclusaologica/" + id);
}
```

- Ações:

```
// src/actions.js
import {
  addPoneyAPI,
  deletePoneyAPI,
  loadPoneysAPI,

  // Novidade aqui
  updatePoneyAPI
} from "./api";

import {
  LOAD_PONEYS,
  LOGIN,
  LOGOUT,
  TOGGLE_VIEW_DELETED_PONEYS,
  ADD_PONEY,
  UPDATE_PONEY,

  // Novidade aqui
  DELETE_PONEY
} from "./constants";

// Código omitido

export function deletePoney(id) {
  return dipatch => {
    deletePoneyAPI(id)
      .then(() => {
        dipatch({
          type: DELETE_PONEY,
          data: id
        });
      })
      .catch(error => {
        alert(error.message);
      });
  };
}

// Código posterior omitido
```

- Reducer:

```
// src/reducers/poneys.js
// Código anterior omitido
import {
  LOAD_PONEYS,
```

```
  TOGGLE_VIEW_DELETED_PONEYS,
  ADD_PONEY,
  UPDATE_PONEY,

  // Novidade aqui
  DELETE_PONEY
} from "../constants";

const initialState = { list: [], viewDeleted: false };

export default function poneysReducer(state = initialState, action) {
  switch (action.type) {
    case LOAD_PONEYS:
      return {
        ...state,
        list: [...action.data]
      };
    case TOGGLE_VIEW_DELETED_PONEYS:
      return {
        ...state,
        viewDeleted: !state.viewDeleted
      };

    case ADD_PONEY:
      return {
        ...state,
        list: [...state.list, action.data]
      };

    case UPDATE_PONEY:
      return {
        ...state,
        list: state.list.map(p =>
          p._id === action.data._id ? { ...action.data } : p
        )
      };

    // Novidade aqui
    case DELETE_PONEY:
      return {
        ...state,
        list: state.list.map(p =>
          p._id === action.data ? { ...p, excluido: true } : p
        )
      };
    default:
      return state;
  }
}
```

E por fim, vamos ligar a função do componente `ListarPoneyScreen` à ação que foi criada:

```
// src/components/ListarPoneyScreen.js
// Código anterior omitido
// Novidade aqui
import { loadPoneys, deletePoney } from "../actions";

class ListarPoneysScreen extends React.Component {
  constructor(props) {
    super(props);
  }

  handleDeletePoney = poney => {
    Alert.alert("Exclusão", `Confirma a exclusão do poney ${poney.nome}?`, [
      // Novidade aqui
      { text: "Sim", onPress: () => this.props.deletePoney(poney._id) },
      { text: "Não", style: "cancel" }
    ]);
  };

  // Código omitido
}

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

ListarPoneysScreen.propTypes = {
  poneys: PropTypes.object,
  profile: PropTypes.object,
  loadPoneys: PropTypes.func,
  navigation: PropTypes.object,

  // Novidade aqui
  deletePoney: PropTypes.func
};
// Código posterior omitido
```