


# Semana OmniStack 11

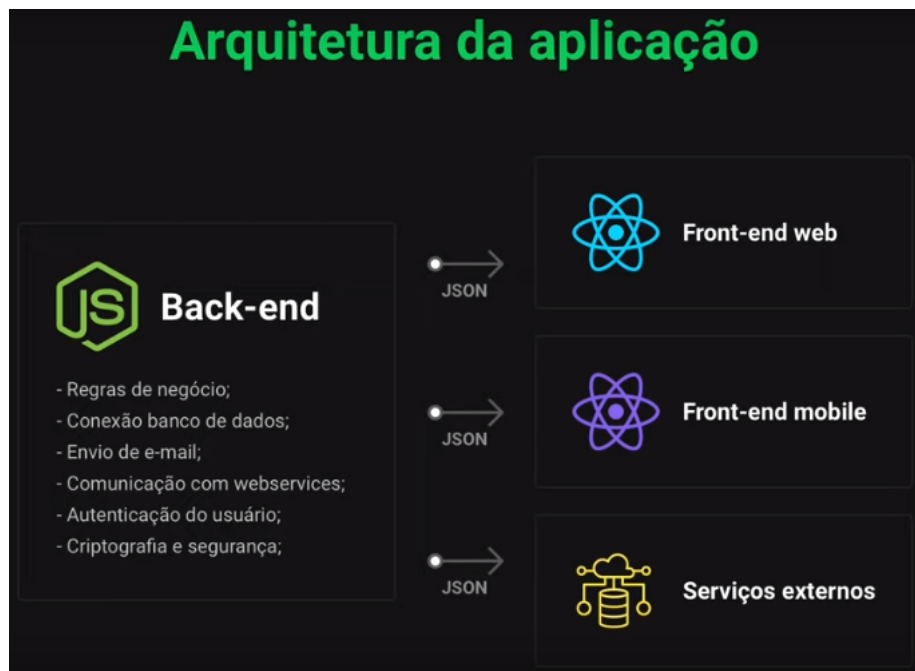
 Aulas realizadas em 23/03/20 - 27/03/20

## Conhecendo a OmniStack

### ▼ Configurar ambiente de desenvolvimento

- **NodeJS e npm**
  - Instalar usando package-manager Chocolatey:  
<https://nodejs.org/en/download/package-manager/#windows>
- **Visual Studio Code**

### ▼ Back-end x Front-end



### ▼ Criando projeto com NodeJS

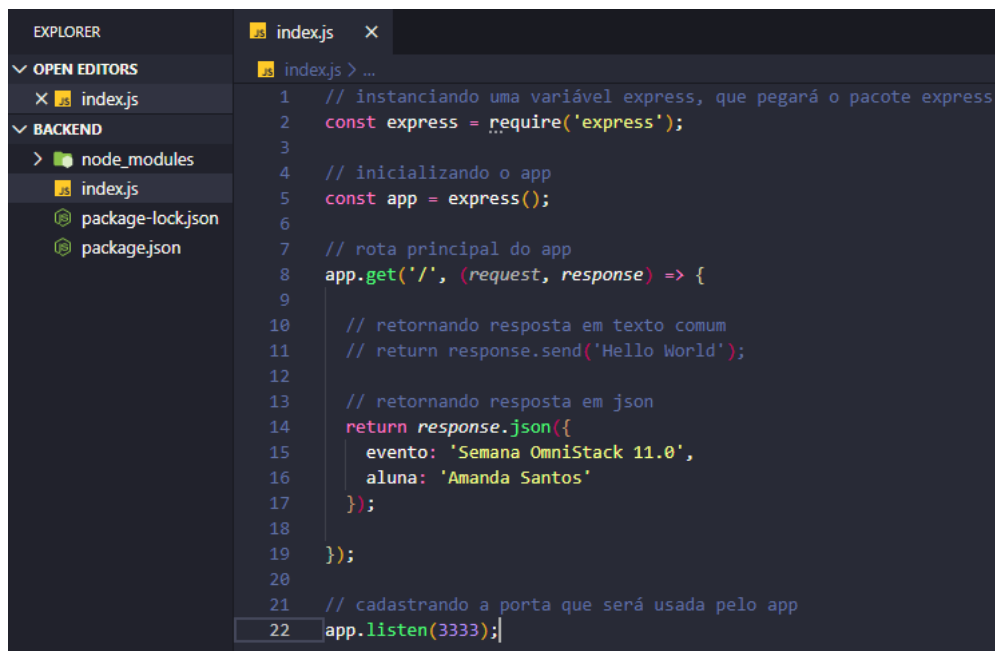
- Criar uma pasta app, diretório de toda a aplicação
- Dentro dela, criar uma pasta backend
- Após criar uma pasta backend, executar nela o comando

```
npm init -y
```

- Com isso, é criado um arquivo package.json.
- Em seguida, executar dentro da pasta o comando a seguir, para instalar a pasta node\_modules

```
npm install express
```

- Criando um arquivo index.js e fazendo Hello World:

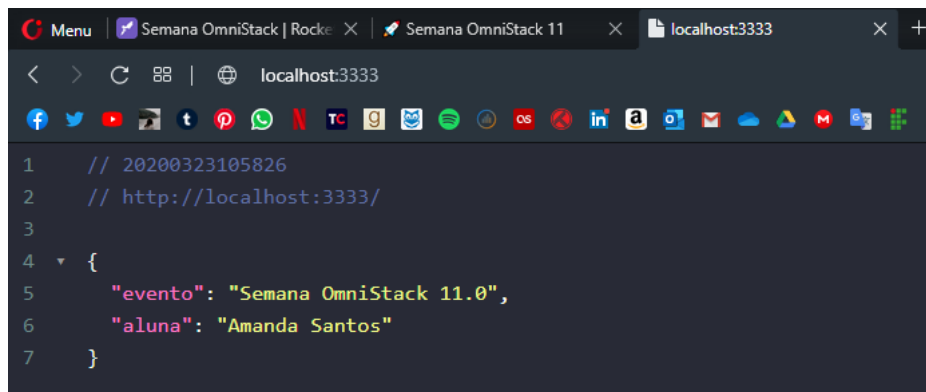


```
1 // instanciando uma variável express, que pegará o pacote express
2 const express = require('express');
3
4 // inicializando o app
5 const app = express();
6
7 // rota principal do app
8 app.get('/', (request, response) => {
9
10     // retornando resposta em texto comum
11     // return response.send('Hello World');
12
13     // retornando resposta em json
14     return response.json({
15         evento: 'Semana OmniStack 11.0',
16         aluna: 'Amanda Santos'
17     });
18
19 });
20
21 // cadastrando a porta que será usada pelo app
22 app.listen(3333);
```

- Rodar o comando a seguir na pasta do projeto:

```
node index.js
```

- Abrindo localhost:3333 no navegador



```
1 // 20200323105826
2 // http://localhost:3333/
3
4 {
5   "evento": "Semana OmniStack 11.0",
6   "aluna": "Amanda Santos"
7 }
```

## ▼ Criando projeto com ReactJS

- Abordagem tradicional x Abordagem de SPA



- Executar o comando a seguir dentro da pasta app para criar uma pasta frontend, onde ficará o projeto ReactJS:

```
npx create-react-app frontend
```

- Para executar o projeto, rodar o comando a seguir dentro de frontend

```
npm start
```

- Com isso, o projeto começa a rodar em localhost:3000

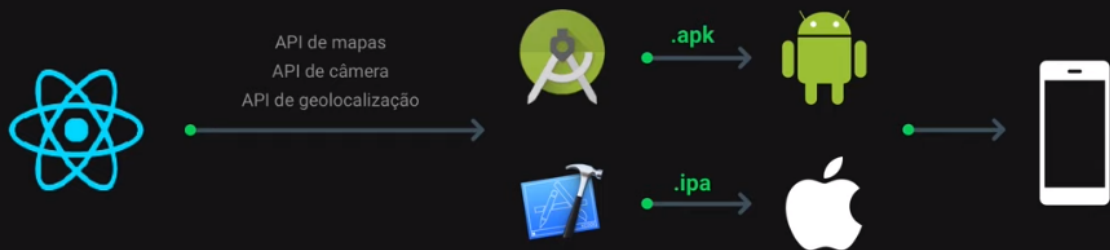
#### ▼ Criando projeto com React Native e Expo

- *Abordagem tradicional x Abordagem do React Native*



- Expo

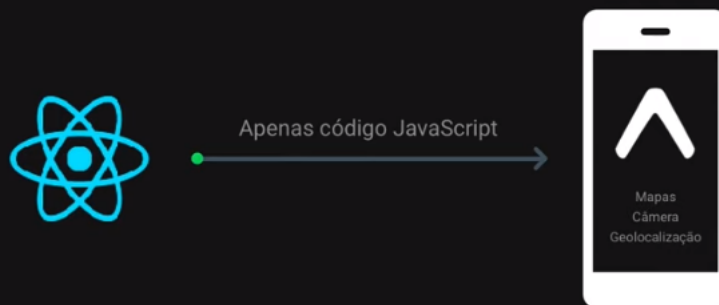
## Por que utilizaremos o Expo?



Sem o Expo, precisamos instalar em nosso sistema tanto o Android Studio para obter a SDK de desenvolvimento Android, e o Xcode (apenas no Mac) para obter a SDK do iOS.

Nesse caso, nossa iniciação no desenvolvimento fica mais penosa, já que essas SDK's não são extremamente simples de instalar e livres de erros.

## Arquitetura do Expo



Com o Expo, nós instalamos um aplicativo no celular chamado Expo, e dentro dele, tudo o que precisamos para desenvolver no React Native já está instalado, como as API's de mapas, geolocalização, câmera, sensores, calendário, etc...

Com isso, não precisamos nos preocupar em gerar o aplicativo pra Android e iOS já que o app do Expo instalado tem tudo o que precisamos e assim usamos apenas React.

### Back-End

#### ▼ Node.js e Express

##### • Rotas e recursos:

- Rotas são um determinado endereço do site. Ex.: `www.meusite.com/users/1`. Recursos são uma determinada informação que eu quero acessar por meio da rota, que geralmente está relacionada a uma tabela no banco de dados. Ex.: em `www.meusite.com/users/1`, quero acessar o usuário com a identificação 1.
- Para melhor organização, criar uma pasta `src` e colocar `index.js` dentro dela. Em seguida, criar um arquivo separado `routes.js`, onde ficarão todas as rotas do projeto. Garantir a exportação das rotas em `routes.js` e a sua importação pela aplicação em `index.js`.

- **Métodos HTTP:**

- **GET:** buscar uma informação do back-end
- **POST:** criar uma informação no back-end
- **PUT:** atualizar uma informação no back-end
- **DELETE:** apagar uma informação do back-end

- **Tipos de parâmetros:**

- **Query Params:** Parâmetros nomeados (você sabe qual o nome da variável enviada: nome=Maria, idade=20) enviados na rota após "?". Muito usados em filtros e paginação. Ex.: [www.meusite.com/users?name=Amanda](http://www.meusite.com/users?name=Amanda).
  - É acessado usando `request.query`.

```
app.get('/users', (request, response) => {  
  const params = request.query;  
  
  console.log(params);  
  
  return response.json({  
    evento: 'Semana OmniStack 11.0',  
    aluno: 'Diego Fernandes'  
  });  
});  
  
app.listen(3333);
```

- **Route Params:** Parâmetros utilizados para identificar recursos. Não são nomeados, pela URL você só sabe o valor do parâmetro. Ex.: [www.meusite.com/users/1](http://www.meusite.com/users/1) (/users/:id) vai buscar todos os usuários de id = 1.
  - É acessado usando `request.params`.

```
app.get('/users/:id', (request, response) => {  
  const params = request.params;  
  
  console.log(params);  
  
  return response.json({  
    evento: 'Semana OmniStack 11.0',  
    aluno: 'Diego Fernandes'  
  });  
});  
  
app.listen(3333);
```

- **Request Body:** é enviado o corpo da requisição (um arquivo JSON contendo os dados que vieram de um formulário, por exemplo), utilizado para inserir ou alterar uma informação/recurso.
  - É acessado com `request.body`.

```
app.post('/users', (request, response) => {  
  const body = request.body;  
  
  console.log(body);  
  
  return response.json({  
    evento: 'Semana OmniStack 11.0',  
    aluno: 'Diego Fernandes'  
  });  
});  
  
app.listen(3333);
```

▼ **Utilizando o Insomnia:** útil para testar as rotas de POST, PUT e DELETE, por exemplo, enquanto ainda não há um front-end.

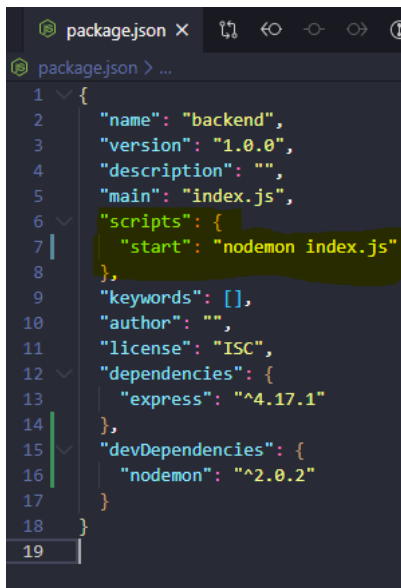
<https://insomnia.rest>

▼ **Configurando Nodemon:**

- O Nodemon permite que o servidor seja atualizado automaticamente após uma alteração no código, durante o desenvolvimento da aplicação. Para instalá-lo, basta rodar o comando a seguir (-D para instalar somente como uma dependência de desenvolvimento - `devDependencies` -, já que ele é usado somente durante o desenvolvimento).

```
npm install nodemon -D
```

- Com isso, basta definir um comando dentro de `package.json > scripts` para executar o Nodemon. Nesse caso, o comando definido foi `start`.



```

1 {
2   "name": "backend",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "start": "nodemon index.js"
8   },
9   "keywords": [],
10  "author": "",
11  "license": "ISC",
12  "dependencies": {
13    "express": "^4.17.1"
14  },
15  "devDependencies": {
16    "nodemon": "^2.0.2"
17  }
18 }
19

```

- Assim, para executar o projeto basta rodar o comando a seguir:

```
npm start
```

#### ▼ Diferenças entre banco de dados:

- SQL x NoSQL
- Neste caso, será usado o SGBD SQL **SQLite**.

#### ▼ Configurando banco de dados:

- Pode-se acessar o banco através do código usando drivers (escrever as queries diretamente no código - ex.: `SELECT * FROM users`) ou usando um **Query Builder**, que constrói a query usando código JavaScript (ex.: `table('users').select('*').where(...)`).
- Neste caso, será usado o Query Builder **Knex** (<http://knexjs.org>). Para instalá-lo, basta rodar

```
npm install knex
```

- Depois, rodar o comando de acordo com o banco que será instalado. Neste caso,

```
npm install sqlite3
```

- Rodar o comando

```
npx knex init
```

para criar o arquivo que contém as configurações do banco de dados. Nele, existem as configurações de *development* (desenvolvimento), *staging* (simulação do ambiente de produção para os devs testarem) e *production* (produção).



```
knexfile.js x
knexfile.js > ...
1 // Update with your config settings.
2
3 module.exports = {
4
5   development: {
6     client: 'sqlite3',
7     connection: {
8       filename: './dev.sqlite3'
9     }
10  },
11
12  > staging: { ...
26  },
27
28  > production: { ...
42  }
43
44  };
```

- Alterar as configurações de conexão para o arquivo onde será armazenado o banco de dados. Neste caso, dentro de *src* foi criada uma pasta *database*, onde ficará o arquivo *db.sqlite*.

```
EXPLORER
OPEN EDITORS
x knexfile.js U
BACKEND
  > node_modu...
  > src
    > database
      index.js U
      routes.js U
      knexfile.js U
      package-loc... M
      package.json M
knexfile.js x
knexfile.js > ...
1 // Update with your config settings.
2
3 module.exports = {
4
5   development: {
6     client: 'sqlite3',
7     connection: {
8       filename: './src/database/db.sqlite'
9     }
10  },
11
12  > staging: { ...
26  },
27
28  > production: { ...
42  }
43
44  };
```

#### ▼ Entidades e funcionalidades da aplicação:

##### • Entidades:

- ONG
- Caso (incident)

##### • Funcionalidades:

- Login de ONG
- Logout de ONG
- Cadastro de ONG
- Cadastro de novos casos
- Exclusão de casos
- Listagem de casos de uma ONG específica
- Listagem de todos os casos
- Entrar em contato com a ONG

- Para criar as tabelas rapidamente, serão usadas migrations. **Migrations** são um "controle de versão" de bancos de dados, que permitem armazenar todos os estados do banco e as alterações que são aplicadas nele.
- Criar um diretório migrations dentro de database.
- Configurar a migration dentro de knexfile.js:

```
knexfile.js X 20200324152425_create_ongs.js
knexfile.js > ...
1 // Update with your config settings.
2
3 module.exports = {
4
5   development: {
6     client: 'sqlite3',
7     connection: {
8       filename: './src/database/db.sqlite'
9     },
10    migrations: {
11      directory: './src/database/migrations'
12    },
13    useNullAsDefault: true,
14  },
15
16 > staging: { ...
30 },
31
32 > production: { ...
46 },
47
48 };
```

- Criando uma migration para criar a tabela ONGs:

```
npx knex migrate:make create_ongs
```

- É criado um arquivo dentro da pasta migrations. Nele, colocar o seguinte código:
  - A função *up* tem o que deve ser feito quando a migration for executada. Neste caso, é um código para criar a tabela *ongs*.
  - A função *down* tem o que deve ser feito caso a migration dê errado. Neste caso, é um código para excluir a tabela *ongs*.



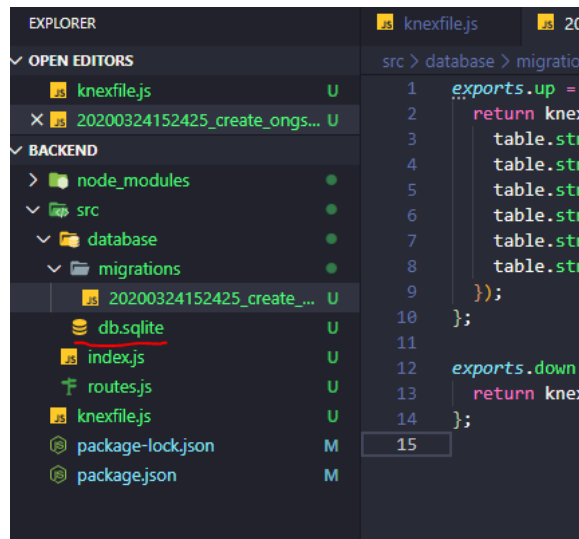
```
EXPLORER
OPEN EDITORS
  knexfile.js U
  20200324152425_create_ongs... U
BACKEND
  node_modules
  src
    database
      migrations
        20200324152425_create_... U
    db.sqlite U
    index.js U
    routes.js U
    knexfile.js U
    package-lock.json M
    package.json M

knexfile.js X 20200324152425_create_ongs.js X
src > database > migrations > 20200324152425_create_ongs.js > ...
1 exports.up = function(knex) {
2   return knex.schema.createTable('ongs', function (table) {
3     table.string('id').primary();
4     table.string('name').nullable();
5     table.string('email').nullable();
6     table.string('whatsapp').nullable();
7     table.string('city').nullable();
8     table.string('uf', 2).nullable();
9   });
10 };
11
12 exports.down = function(knex) {
13   return knex.schema.dropTable('ongs');
14 };
15
```

- Para executar a migration:

```
npx knex migrate:latest
```

- Com isso, o banco de dados já é criado com a tabela *ongs*.



- Exemplo com a tabela *incidents*, que possui uma chave estrangeira referenciando *ongs*:



- Para desfazer uma migration, caso tenha cometido algum erro, basta fazer um *rollback* com o comando a seguir:

```
npx knex migrate:rollback
```

- Para listar todas as migrations executadas:

```
npx knex migrate:status
```

## ▼ Construção do back-end

### ▼ Adicionando módulo CORS

- É um módulo para adicionar segurança à aplicação. Pode ser adicionado com o comando a seguir:

```
npm install cors
```

- Em `index.js`, importar o CORS:

```

1 // instanciando uma variável express, que pegará o pacote express
2 const express = require('express');
3
4 // importando o módulo de segurança cors
5 const cors = require('cors');
6
7 // importando o arquivo com as rotas
8 const routes = require('./routes');
9
10 // inicializando o app
11 const app = express();
12
13 // cors
14 app.use(cors());
15

```

#### ▼ Enviando back-end ao GitHub

### Front-End

#### ▼ Conceitos do ReactJS:

- **Componente:** função que retorna HTML
- **JSX:** JavaScript + XML - HTML integrado ao JavaScript
- **Propriedades**
- **Estado e Imutabilidade:**
  - Estado é uma variável do componente que, quando é alterada, o componente é renderizado novamente. Assim, todas as alterações são refletidas na tela de forma dinâmica.
  - A variável não pode ser alterada diretamente, somente usando `setState` ou a declaração `[valor, funcaoQueAlteraValor] = useState(<valorInicial>)` ⇒ **imutabilidade**.
  - Deve-se sempre importar `{ useState }` no componente que possui estado.
  - Exemplo com um contador dinâmico:

```

1 import React, { useState } from 'react';
2
3 import Header from './Header';
4
5 function App() {
6   const [counter, setCounter] = useState(0);
7
8   function increment() {
9     setCounter(counter + 1);
10  }
11
12  return (
13    <div>
14      <Header>Contador: {counter}</Header>
15      <button onClick={increment}>Incrementar</button>
16    </div>
17  );
18 }
19
20 export default App;
21

```

#### ▼ Página de Login

#### ▼ Configurando rotas

#### ▼ Cadastro de ONGs

#### ▼ Listagem de casos

#### ▼ Cadastro de um novo caso

#### ▼ Conectando aplicação à API

#### ▼ Enviar projeto ao GitHub

#### ▼ Pacotes para instalar:

- Para usar ícones do Material Icons, FontAwesome, etc. dentro do React:

```
npm install react-icons
```

- Para usar rotas:

```
npm install react-router-dom
```

- Para fazer requisições HTTP à API:

```
npm install axios
```

## App Mobile

#### ▼ Instalando Expo

- Instalar o Expo de forma global na máquina usando o comando a seguir:

```
npm install -g expo-cli
```

- Para verificar se a instalação deu certo, executar o seguinte comando:

```
expo -h
```

#### ▼ Criando projeto React Native

- Comando para criar um projeto React Native chamado *mobile*:

```
expo init mobile
```

#### ▼ Executando projeto

##### • No celular:

- Executar o seguinte comando dentro da pasta do projeto:

```
yarn start
```

- Instalar o app *Expo* no celular
- Escanear o QRCode

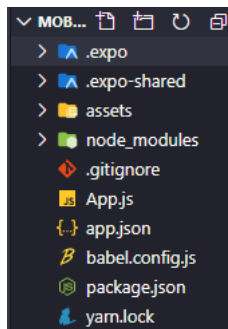
##### • Emuladores:

<https://www.youtube.com/watch?v=eSjFDWYkdxM&vl=pt>

##### • Expo Snack:

<https://snack.expo.io>

#### ▼ Estrutura de pastas



### ▼ Diferenças para o ReactJS

```
1 import React from 'react'; 8.5K (gzipped: 3.4K)
2 import { StyleSheet, Text, View } from 'react-native';
3
4 export default function App() {
5   return (
6     <View style={styles.container}>
7       <Text style={styles.title}>Hello OmniStack</Text>
8     </View>
9   );
10 }
11
12 const styles = StyleSheet.create({
13   container: {
14     flex: 1,
15     backgroundColor: '#7159c1',
16     alignItems: 'center',
17     justifyContent: 'center',
18   },
19
20   title: {
21     color: 'white',
22     fontSize: 20,
23     fontWeight: 'bold'
24   },
25 });
26
```

#### • Elementos HTML e Semântica:

- **Qualquer container** usa a tag `<View />`.
- **Qualquer texto** usa a tag `<Text />`.
- Não existe `<div>`, `<h1>`, `<p>`, `<header>`, `<span>`, etc.

- **Estilização:** Para estilização, adicionar a tag `style` e definir uma constante `styles` no mesmo arquivo ou em outro arquivo `styles.js`.

- **Flexbox:** Todos os elementos são `display: 'flex'` por padrão. Não existe `block`, `inline-block`, etc.
- **Propriedades:** Propriedades com *CamelCase*, sem hífen. Ex.: `backgroundColor` em vez de `background-color`.
- **Herança de estilos e estilização própria:** Não existe herança de estilos. Caso deseje que uma `<View />` possua texto com cor branca, incluindo o `<Text />` dentro dela, é necessário definir estilização separada para o `<Text />`. Só colocar na estilização da `<View />` não adianta.

### ▼ Ícone e Splash Screen

- O arquivo com o ícone do app deve ser `icon.png`.
- O arquivo com a Splash Screen do app deve ser `splash.png`.
- Ambos devem ficar na pasta `assets`.
- Em `app.json`, mudar a `backgroundColor` para a cor da Splash Screen do app, caso estejam sobrando bordas brancas quando a Splash Screen é mostrada.

```
app.json x App.js
app.json > {} expo > {} splash > backgroundColor
1 {
2   "expo": {
3     "name": "Be The Hero",
4     "slug": "bethehero",
5     "privacy": "public",
6     "sdkVersion": "36.0.0",
7     "platforms": [
8       "ios",
9       "android",
10      "web"
11    ],
12    "version": "1.0.0",
13    "orientation": "portrait",
14    "icon": "./assets/icon.png",
15    "splash": {
16      "image": "./assets/splash.png",
17      "resizeMode": "contain",
18      "backgroundColor": "#E02041"
19    },
20    "updates": {
```

### ▼ Configurando navegação

- <https://reactnavigation.org/docs/getting-started>
- Executar os comandos a seguir para instalar o React Navigation para Expo:

```
npm install @react-navigation/native
```

```
expo install react-native-gesture-handler react-native-reanimated react-native-screens react-native-safe-area-context @react-native-community/masked-view
```

- <https://reactnavigation.org/docs/hello-react-navigation>
- Neste projeto, será usado o `createStackNavigator`, que permite uma navegação simples entre páginas, usando apenas botões. Para instalá-lo, usar o comando a seguir:

```
npm install @react-navigation/stack
```

### ▼ Página de casos

- Instalar a biblioteca Constants do Expo para acessar variáveis constantes do sistema mobile, como, no nosso caso, a `statusBarHeight`.

```
expo install expo-constants
```

- Para permitir a formatação em reais do valor, instalar a biblioteca Intl:

```
npm install intl
```

### ▼ Detalhe do caso

#### • Abrindo WhatsApp e E-mail

- Para abrir um app de e-mail diretamente a partir do meu app, rodar o comando a seguir para instalar a biblioteca de e-mail do composer:

```
expo install expo-mail-composer
```

### ▼ Conexão com a API

- Para fazer a conexão com a API, instalar o Axios

```
npm install axios
```

#### ▼ Enviando projeto ao GitHub

### Outros

📌 Site para prototipação de telas: <https://www.figma.com>

📌 Site para anotações: <https://www.notion.so>