RESTful API com Node.js

& React-Native

Alunos: Felipe Tavares, Nícholas Prado e Samuel Antunes Vieira

O que é RESTful API

Para uma API ser considerada com uma API REST devem ser seguidas as seguintes diretrizes:

- **Cliente-servidor** - Ao separar as *concerns* da interface do usuário das *concerns* com o armazenamento de dados, melhoramos a portabilidade da interface do usuário em várias plataformas e melhoramos a escalabilidade, simplificando os componentes do servidor.
- **Sem estado** - Cada solicitação do cliente para o servidor deve conter todas as informações necessárias para entender a solicitação e não pode tirar proveito de nenhum contexto armazenado no servidor. O estado da sessão é, portanto, mantido inteiramente no cliente.

UPF

- **Armazenamento em cache** - as restrições de cache exigem que os dados em uma resposta a uma solicitação sejam implicitamente ou explicitamente rotulados como armazenáveis em cache ou não em cache. Se uma resposta for armazenável em cache, o cache do cliente terá o direito de reutilizar esses dados de resposta para solicitações posteriores equivalentes.
- **Interface uniforme** - Ao aplicar o princípio de generalidade da engenharia de software à interface do componente, a arquitetura geral do sistema é simplificada e a visibilidade das interações é aprimorada. Para obter uma interface uniforme, são necessárias várias restrições arquiteturais para orientar o comportamento dos componentes. O REST é definido por quatro restrições de interface: identificação de recursos; manipulação de recursos através de representações; mensagens auto-descritivas; e hipermídia como o mecanismo do estado do aplicativo

UPF

- **Sistema em camadas** - O estilo do sistema em camadas permite que uma arquitetura seja composta de camadas hierárquicas, restringindo o comportamento do componente, de modo que cada componente não possa "ver" além da camada imediata com a qual está interagindo.

- **Código sob demanda (opcional)** - o REST permite que a funcionalidade do cliente seja estendida baixando e executando o código na forma de *applets* ou *scripts*. Isso simplifica os clientes, reduzindo o número de recursos necessários para a pré-implementação.

UPF

Recurso

A abstração principal das informações no REST é como um recurso. Qualquer informação que possa ser nomeada pode ser um recurso: um documento ou imagem, uma coleção de outros recursos, um objeto não virtual (por exemplo, uma pessoa) e assim por diante. O REST usa um **identificador de recurso** para identificar o recurso específico envolvido em uma interação entre componentes.

O estado do recurso em qualquer momento é conhecido como representação de recurso. Uma representação consiste em dados, metadados que descrevem os links de dados e hipermídia, que podem ajudar os clientes na transição para o próximo estado desejado.

UPF

O formato dos dados de uma representação é conhecido como um tipo de mídia . O tipo de mídia identifica uma especificação que define como uma representação deve ser processada. Uma API RESTful se parece com *hipertexto*. Toda unidade de informação endereçável carrega um endereço, explicitamente (por exemplo, atributos de link e ID) ou implicitamente (por exemplo, derivado da definição de tipo de mídia e estrutura de representação).

De acordo com Roy Fielding:

Hipertexto significa a apresentação simultânea de informações e controles, de forma que as informações se tornem o meio pelo qual o usuário (ou autômato) obtém escolhas e seleciona ações. Lembre-se de que o hipertexto não precisa ser HTML (ou XML ou JSON) em um navegador. As máquinas podem seguir os links quando entenderem o formato dos dados e os tipos de relacionamento.

UPF

Além disso, as representações de recursos devem ser auto-descritivas : o cliente não precisa saber se um recurso é funcionário ou dispositivo. Ele deve agir com base no tipo de mídia associado ao recurso. Portanto, na prática, você acabará criando muitos tipos de mídia personalizados - normalmente um tipo de mídia associado a um recurso.

Todo tipo de mídia define um modelo de processamento padrão. Por exemplo, o HTML define um processo de renderização para o hipertexto e o comportamento do navegador em torno de cada elemento. Ele não tem relação com os métodos de recurso **GET / PUT / POST / DELETE /...** além do fato de que alguns elementos do tipo de mídia definirão um modelo de processo que é como “elementos âncora com um atributo href criam um link de hipertexto que, quando selecionado, chama uma solicitação de recuperação (GET) no URI correspondente ao atributo href codificado em CDATA. "

UPF

Metódos de Recursos

Outra coisa importante associada ao REST são os métodos de recurso a serem usados para executar a transição desejada. Um grande número de pessoas relaciona incorretamente os métodos de recursos aos métodos HTTP GET / PUT / POST / DELETE .

Roy Fielding nunca mencionou nenhuma recomendação sobre o método a ser usado em que condição. Tudo o que ele enfatiza é que deve ser uma interface uniforme . Se você decidir que o HTTP POST será usado para atualizar um recurso - em vez de a maioria das pessoas recomendar HTTP PUT - tudo bem e a interface do aplicativo será RESTful.

UPF

Idealmente, tudo o que é necessário para alterar o estado do recurso deve fazer parte da resposta da API para esse recurso - incluindo métodos e em que estado eles deixarão a representação.

Uma API REST deve ser inserida sem conhecimento prévio além do URI inicial (marcador) e conjunto de tipos de mídia padronizados que são apropriados para o público-alvo (ou seja, espera-se que seja entendido por qualquer cliente que possa usar a API). A partir desse ponto, todas as transições de estado do aplicativo devem ser conduzidas pela seleção do cliente de opções fornecidas pelo servidor que estão presentes nas representações recebidas ou implícitas pela manipulação do usuário dessas representações. As transições podem ser determinadas (ou limitadas por) o conhecimento do cliente sobre os tipos de mídia e os mecanismos de comunicação de recursos, os quais podem ser aprimorados imediatamente (por exemplo, código sob demanda).

Criando a base do projeto

Primeiramente, é necessário ter instalado na máquina local o Node.js, disponível no link: https://nodejs.org/en/download/

Com o Node instalado, pode-se utilizar o pacote create-react-app, com ele, é possível inserir o comando *create-react-app node-react* no terminal, e criar um novo projeto, chamado node-react.

Feito isso, ao entrar na pasta raiz do projeto, cria-se uma pasta *client*, movendo-se todo o código criado para dentro dela.

UPF

Comunicação com o Node.js

Na raíz do projeto, ao criar manualmente o arquivo *package.json*, serão feitas as definições de nome do projeto, versão, comandos e dependências.

```
{
  "name": "node-react",
  "version": "1.0.0",
  "scripts": {
    "client": "cd client && npm start",
    "server": "node server.js",
    "dev": "concurrently --kill-others-on-fail \"npm run server\" \"npm run client\""
  },
  "dependencies": {
    "express": "^4.16.2"
  },
  "devDependencies": {
    "concurrently": "^3.5.0"
  }
}
```

UPF

Ao declarar como dependência o pacote *concurrently*, é possível executar tanto o App React quanto o servidor ao mesmo tempo.

Para gerar um servidor, cria-se, na pasta raíz do projeto, o arquivo *server.js*, programando:

```
1  const express = require('express');
2
3  const app = express();
4  const port = process.env.PORT || 5000;
5
6  app.get('/api/mensagem', (req, res) => {
7    res.send({ express: 'Hello From Express' });
8  });
9
10 app.listen(port, () => console.log(`Listening on port ${port}`));
```

UPF

Com o uso do *express* é criada uma simples *API REST*, que cria um endpoint em */api/mensagem* e retorna mensagens a partir do método *res.send()*

Retornando ao diretório */client*, e editando o arquivo *package.json*, gerado automaticamente, adicionando: *"proxy": "http://localhost:5000/"*

Com isso, as requisições da API serão recebidas pelo servidor.

Para renderizar a primeira tela, e trocar mensagens entre servidor e cliente, edita-se o arquivo */client/src/App.js*

UPF

Renderização de tela e troca de mensagens

```
1  import React, {Component} from 'react';
2  import logo from './logo.svg';
3  import './App.css';
4
5  class App extends Component {
6    state = {
7      response: ''
8    };
9
10   componentDidMount() {
11     this.callApi()
12       .then(res => this.setState({ response: res.express }))
13       .catch(err => console.log(err));
14   }
15
16   callApi = async () => {
17     const response = await fetch('/api/mensagem');
18     const body = await response.json();
19     if (response.status !== 200) throw Error(body.message);
20
21     return body;
22   };
23
```

UPF

```
23
24    render() {
25      return (
26        <div className="App">
27          <header className="App-header">
28            <img src={logo} className="App-logo" alt="logo" />
29            <h1 className="App-title">Welcome to React</h1>
30          </header>
31          <p className="App-intro">{this.state.response}</p>
32        </div>
33      );
34    }
35  }
36
37  export default App;
```

Ao salvar os arquivos e executar o comando *npm install* na pasta raíz do projeto, já se pode testar a aplicação em funcionamento, com *npm run dev*.

UPF

And a hello from express RESTful API!

UPF

Programando um app Sorteador de Números

Para seguir com um desenvolvimento mais específico, serão utilizados pacotes mais específicos, de forma a navegar entre páginas e evitar instabilidades.

Os comandos a seguir são inseridos no diretório raíz do projeto.

UPF

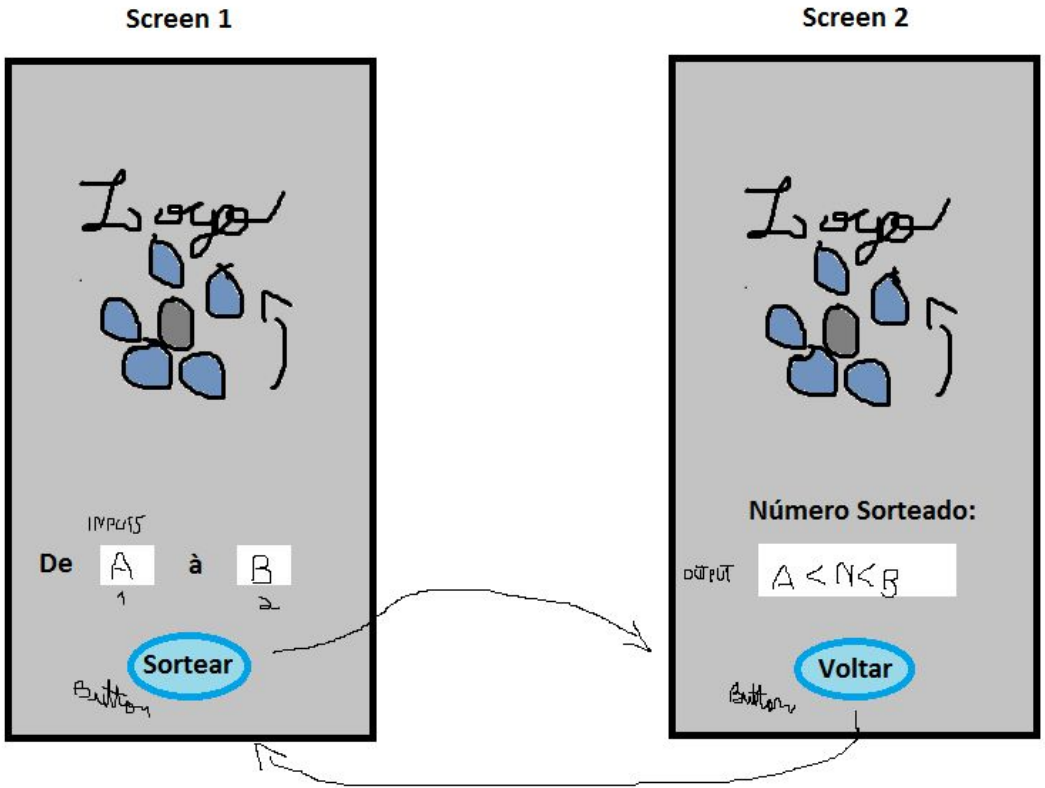
```
npm install --global yarn
npm install
npm install react
npm install react-dom
npm install react-sripts
npm install react-native
npm install react-native-web
npm install react-art
npm install react-router-native
npm install react-router-dom
npm install
npm install --save @react-native-community/masked-view
npm install react-native-safe-area-context
npm install
npm audit fix
yarn add react-navigation
yarn add react-native-screens
yarn add react-navigation-stack
yarn add react-native-gesture-handler
react-native link react-native-gesture-handler
npm install --save-dev babel-core
npm install --save-dev @babel/preset-flow
npm install --save-dev @babel/preset-typescript
npm install --save-dev @babel/plugin-transform-flow-strip-types
npm install @react-navigation/native @react-navigation/stack
npm install react-native-reanimated react-native-gesture-handler react-native-screens react-native-safe-area-context @react-native-community/masked-view

npm install
npm run dev

(caso erro de cache: npm rebuild)
```

UPF

Protótipo



O índice de renderização

No arquivo */client/scr/index.js*, é renderizada a tela que será mostrada pelo *localhost*. Pode-se apresentar a tela a ser importada. Neste caso, substituirá-se *App.js* por *Apontador.js*, que apontará as telas *Screen1.js* e *Screen2.js*.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './Apontador';

ReactDOM.render(
  <React.StrictMode>
      <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Desenvolvimento Apontador - Imports

São feitos os imports que serão usados no arquivo, incluindo as telas a serem apresentadas, o container de navegação e a pilha de navegação.

```
import * as React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import Screen1 from './Screen1';
import Screen2 from './Screen2';
```

UPF

Desenvolvimento Apontador - Pilha de Navegação

Cria-se a pilha de navegação *Stack*, e na função *App()* é criado o container que conterá a pilha de navegação. Esta função é exportada para ser utilizada pelo *Index.js*.

Nesta pilha, serão postas as telas a serem navegadas, e seus atributos como nome atribuição.

```
const Stack = createStackNavigator()

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen
          name="Screen1"
          component={Screen1}
          options={{ title: 'Welcome' }}
        />
        <Stack.Screen
            name="Screen2"
            component={Screen2}
        />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

UPF

Desenvolvimento Screen 1 - Imports

São feitos os imports que serão usados no arquivo:

```
import React, { Component } from 'react';
import { StyleSheet, View, ActivityIndicator, Button } from 'react-native';
import logo from './logo.svg';
import './Screen1.css';
```

UPF

Desenvolvimento Screen 1 - Método Construtor

Em seguida é feito o método construtor, inicializando o local state através da atribuição de objetos ao this.state.
min: valor mínimo do sorteio
max: valor máximo do sorteio
response: resposta da API
post: envio dos valores para API
responseToPost1: resposta ao envio do valor mínimo
responseToPost2: resposta ao envio do valor máximo

```
export default class Screen1 extends Component {
    constructor(props) {
        super(props);

        this.state = {
            min: '',
            max: '',
            response: '',
            post: '',
            responseToPost1: '',
            responseToPost2: '',
        };
    }
```

Desenvolvimento Screen 1 - Função acessarApp()

Para acessar a segunda página (Screen 2), a função acessarApp() é programada.

```
acessarApp() {
    this.props.navigation.replace('Screen2');
}
```

UPF

Desenvolvimento Screen 1 - Função renderButton()

Como é possível programar outras funções para que a função render seja configurada como queremos, programaremos a função renderButton. Nela será adicionado o botão que realizará o sorteio entre o intervalo.

Em seu onPress adiciona-se a função acessarApp(), programada no slide anterior. Dessa maneira ao clicar no botão “SORTEAR” a segunda página é aberta.

```
renderButton() {
    return (
        <View style={styles.btn}>
            <Button
                title='SORTEAR'
                color='#a08af7'
                onPress={() => this.acessarApp()}
            />
        </View>
    )
}
```

UPF

Desenvolvimento Screen 1 - Função callApi()

A função assíncrona callApi faz uma chamada na API e fica esperando o servidor mandar a mensagem que será armazenada em response, logo após é convertido para objeto json e fica armazenado no body. Em seguida retorna o body.

```
callApi = async () => {
    const response = await fetch('/api/mensagem');
    const body = await response.json();
    if (response.status !== 200) throw Error(body.message);

    return body;
};
```

UPF

Desenvolvimento Screen 1 - componentDidMount()

Para a chamada da API é interessante que seja feita apenas depois que tudo está pronto na página, para não correr o risco de renderizar algo que ainda não está pronto. Dessa maneira utiliza-se o componentDidMount, que será executado apenas depois de todos os componentes estarem montados.

Uma vez que a função callApi é assíncrona, utilizando a promise .then espera-se receber a resposta da API.

```
componentDidMount() {
    this.callApi()
      .then(res => this.setState({ response: res.express }))
      .catch(err => console.log(err));
}
```

UPF

Desenvolvimento Screen 1 - Função handleSubmit()

A função handleSubmit enviará para a api, através do método POST, o número que será digitado no primeiro input (num min) do render().

```
handleSubmit = async e => {
    e.preventDefault();
    const response = await fetch('/api/num1', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ post: this.state.min }),
    });
    const body = await response.text();

    this.setState({ responseToPost1: body });
};
```

UPF

Desenvolvimento Screen 1 - Função handleTimbus()

A função handleTimbus enviará para a api, através do método POST, o número que será digitado no segundo input (num max) do render().

```
handleTimbus = async f => {
    f.preventDefault();
    const response = await fetch('/api/num2', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ post: this.state.max }),
    });
    const body = await response.text();

    this.setState({ responseToPost2: body });
};
```

UPF

Desenvolvimento Screen 1 - render()

No *render()* é criado o header contendo o logo e o título.
A "*Screen1-intro*" apresentará a mensagem recebida através da primeira chamada da api.
No primeiro input o valor escrito é enviado para o *handleSubmit* ao clicar no botão *SENTmin*.
O valor é armazenado em value e o onChange permite a atualização para visualização da mudança

```
render() {
    return (
        <div className="Screen1">
            <header className="Screen1-header">
                <img src={logo} className="Screen1-logo" alt="logo" />
                <h4 className="Screen1-title">Sorteador</h4>
            </header>
            <p className="Screen1-intro">{this.state.response}</p>

            <form onSubmit={this.handleSubmit}>
                <input
                    placeholder="From: (min number)"
                    type="text"
                    value={this.state.min}
                    onChange={e => this.setState({ min: e.target.value })}
                />
                <button type="submit"
                        color='white'
                >SENTmin
                </button>
            </form>
```

UPF

Desenvolvimento Screen 1 - render()

No segundo input o valor escrito é enviado para o *handleTimbus* ao clicar no botão *SENTmax*.
O valor é armazenado em value e o onChange permite a atualização para visualização da mudança.
Em seguida mostra a resposta para os posts.
Por fim chama o *renderButton()*.

```
                <form onSubmit={this.handleTimbus}>
                    <input
                        placeholder="To: (max number)"
                        type="text"
                        value={this.state.max}
                        onChange={f => this.setState({ max: f.target.value })}
                    />
                    <button
                        type="submit"
                        color='white'
                    >SENTmax
                    </button>
                </form>
                <p>{this.state.responseToPost1}</p>
                <p>{this.state.responseToPost2}</p>
                {this.renderButton()}
            </div>
        );
    }
}
```

UPF

Desenvolvimento Screen 1 - StyleSheet

Aqui foi criado o *StyleSheet*, contendo o estilo dos botões.

```
const styles = StyleSheet.create({
    btn: {
        paddingTop: 20,
        fontSize: 11,
    }
```

UPF

Desenvolvimento Screen 2 - Imports

São feitos os imports que serão usados no arquivo:

```
import React, { Component } from 'react';
import { View, Text, StyleSheet, ActivityIndicator, Button } from 'react-native';
import logo from './logo.svg';
import './Screen1.css';
```

UPF

Desenvolvimento Screen 2 - Método Construtor

Em seguida é feito o método construtor, inicializando o local state através da atribuição do objeto ao *this.state.response:* resposta da API

```
export default class Screen2 extends Component {
    constructor(props) {
        super(props);

        this.state = {
            response: '',
        };
    }
```

Desenvolvimento Screen 2 - Função acessarScreen1()

Para voltar à primeira página (Screen 1), a função *acessarScreen1()* é programada.

```
acessarScreen1(){
    this.props.navigation.replace('Screen1');
}
```

UPF

Desenvolvimento Screen 2 - Função renderButton()

Como é possível programar outras funções para que a função render seja configurada como queremos, programaremos a função *renderButton*. Nela será adicionado o botão que realizará o sorteio entre o intervalo.

Em seu onPress adiciona-se a função *acessarScreen1()*, programada no slide anterior. Dessa maneira ao clicar no botão "*VOLTAR*" a primeira página é aberta.

```
renderButton() {
    return (
        <View style={styles.btn}>
            <Button
                title='VOLTAR'
                color='#a08af7'
                onPress={() => this.acessarScreen1()}
            />
        </View>
    )
}
```

UPF

Desenvolvimento Screen 2 - Função callApi()

A função assíncrona *callApi* faz uma chamada na API e fica esperando o servidor mandar o número sorteado que será armazenado em response, logo após é convertido para objeto json e fica armazenado no body. Em seguida retorna o body.

```
callApi = async () => {
    const response = await fetch('/api/sort');
    const body = await response.json();
    if (response.status !== 200) throw Error(body.message);

    return body;
};
```

Desenvolvimento Screen 2 - componentDidMount()

Para a chamada da API é interessante que seja feita apenas depois que tudo está pronto na página, para não correr o risco de renderizar algo que ainda não está pronto. Dessa maneira utiliza-se o *componentDidMount*, que será executado apenas depois de todos os componentes estarem montados.

```
componentDidMount() {
    this.callApi()
      .then(res => this.setState({ response: res.express }))
      .catch(err => console.log(err));
}
```

Uma vez que a função *callAPI* é assíncrona, utilizando a promise *.then* espera-se receber a resposta da API.

UPF

Desenvolvimento Screen 2 - render()

```
render() {
    return (
            <div className="Screen1">
                <header className="Screen1-header">
                    <img src={logo} className="Screen1-logo" alt="logo" />
                    <h4 className="Screen1-title">Numero Sorteado</h4>
                </header>

                <View>
                    <Text style={{ color: 'red', fontSize: 20 }} h1>{this.state.response}</Text>
                </View>
                {this.renderButton()}
            </div>
    );
}
```

No *render()* é criado o header contendo o logo e o título.

Na *tag View* é apresentado o número sorteado recebido pela API.

Por fim chama o *renderButton()*.

UPF

Desenvolvimento Screen 2 - StyleSheet

Aqui é criado o *StyleSheet*, contendo o estilo do botão.

```
const styles = StyleSheet.create({
    btn: {
        paddingTop: 20,
        fontSize: 11,
    }
```

UPF

Desenvolvimento Server - Variáveis e APIs

Inicia-se declarando as variáveis e APIs a serem utilizadas.

express lidará com as trocas de mensagem.

bodyParser lidará com objetos no formato *json*.

As demais variáveis globais representam valores de sorteio e porta local.

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = process.env.PORT || 5000;

var minimum = 1;
var maximum = 20;
var sortimum = 0;

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

UPF

Desenvolvimento Server - Função getRandom()

É criada a função *getRandomIntInclusive()*, que recebe como parâmetro o mínimo e o máximo de um intervalo e retorna um número aleatório dentro dele, incluindo os dois parâmetros.

```
function getRandomIntInclusive(min, max) {
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

UPF

Desenvolvimento Server - Função convertObjToNum()

Em seguida é criada a função *convertObjToNum()* para converter a string recebida pela api para tipo inteiro.

```
function convertObjToNum(obj) {
    var a = JSON.stringify(obj.post);
    var b = JSON.parse(a);
    var c = parseInt(b);
    return c;
}
```

Desenvolvimento Server - GET

O método *GET* controla a mensagem inicial do app, que é mostrada logo no início da aplicação.
Controla também o número sorteado através da função anterior, enviando-o através do send, quando é solicitado.

```
app.get('/api/mensagem', (req, res) => {
  res.send({ express: 'And a hello from express RESTful API!' });
});

app.get('/api/sort', (req, res) => {
    sortimum = getRandomIntInclusive(minimum, maximum);
    console.log(sortimum);
    res.send(
        { express: `Number: ${sortimum}` }
    );
});
```

UPF

Desenvolvimento Server - POST

Métodos *POST* lidam com os valores mínimo e máximo a serem sorteados, convertendo-os para um número inteiro e retornando um aviso de que os valores foram armazenados pelo servidor.

```
app.post('/api/num1', (req, res) => {
    minimum = convertObjToNum(req.body);
    console.log(typeof minimum);
    console.log(minimum);
    res.send(
        `I received your POST request. This is what you sent me: ${minimum}`,
    );
});

app.post('/api/num2', (req, res) => {
    maximum = convertObjToNum(req.body);
    console.log(typeof maximum);
    console.log(maximum);
    res.send(
        `I received your POST request. This is what you sent me: ${maximum}`,
    );
});
```

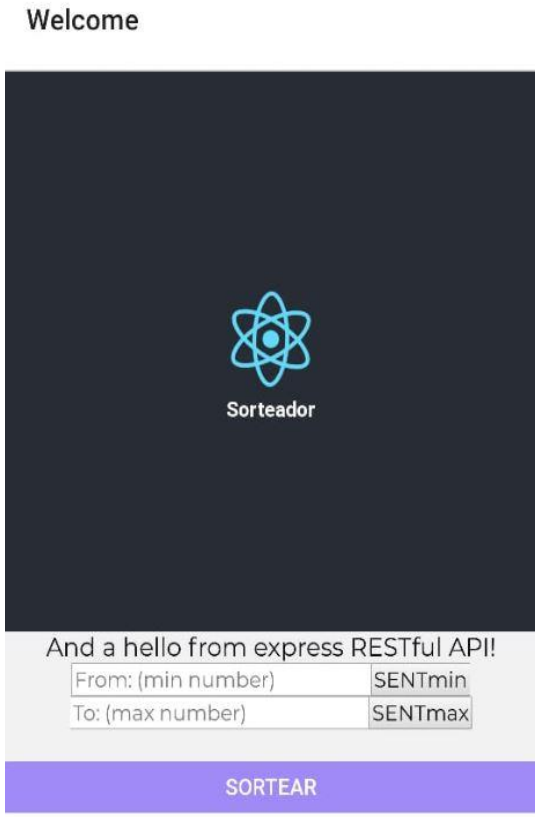
UPF

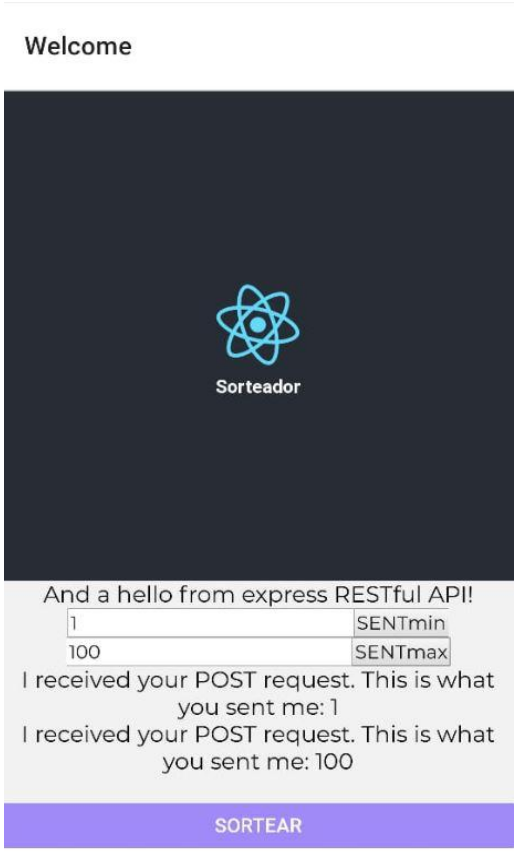
Desenvolvimento Server - LISTEN

Cria um server que ouve na porta “port” (5000) do computador.

```
app.listen(port, () => console.log(`Listening on port ${port}`));
```

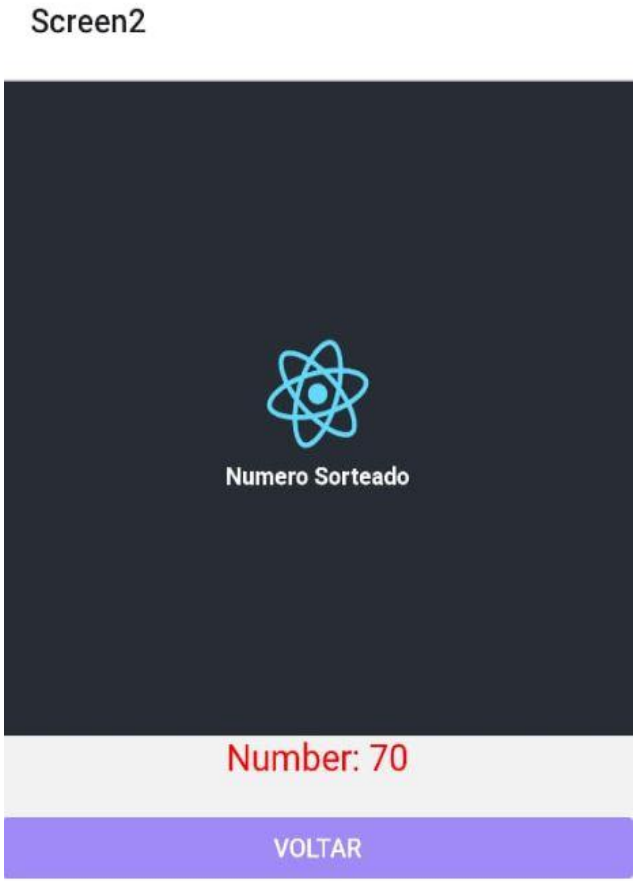
Aplicação Final - Tela 1





UPF

Aplicação Final - Tela 2



Fim

UPF