



Front End Developer

Técnicas de Programação Avançada para a Web

Sara Monteiro

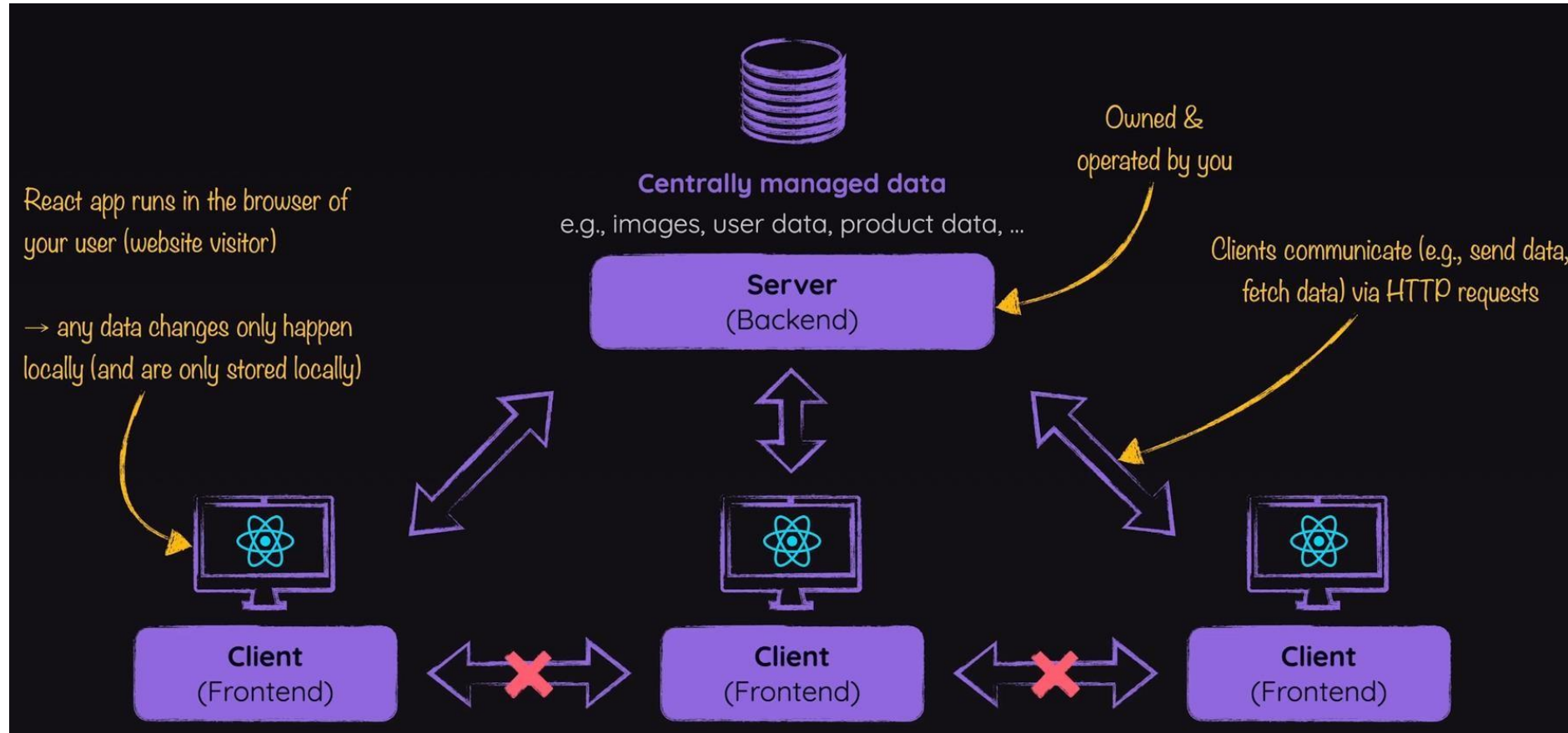
sara.monteiro.prt@msft.cesae.pt

Gerir a nossa App Globalmente

Até agora temos vindo a usar dados que estão armazenados no nosso *Client Side*.

E nos casos em que temos por exemplo uma lista de users que queremos quer várias pessoas actualizem?

Gerir a nossa App Globalmente



Pedidos HTTP e *Data Fetching*

Iremos então aprender como conectar o nosso React a um Backend ou a uma API Externa e trabalhar esses dados para os usarmos na nossa aplicação.

Iremos também ler um formulário preenchido por um user e alimentar com esses dados o nosso Backend.

Ligação a Bases de Dados & Segurança

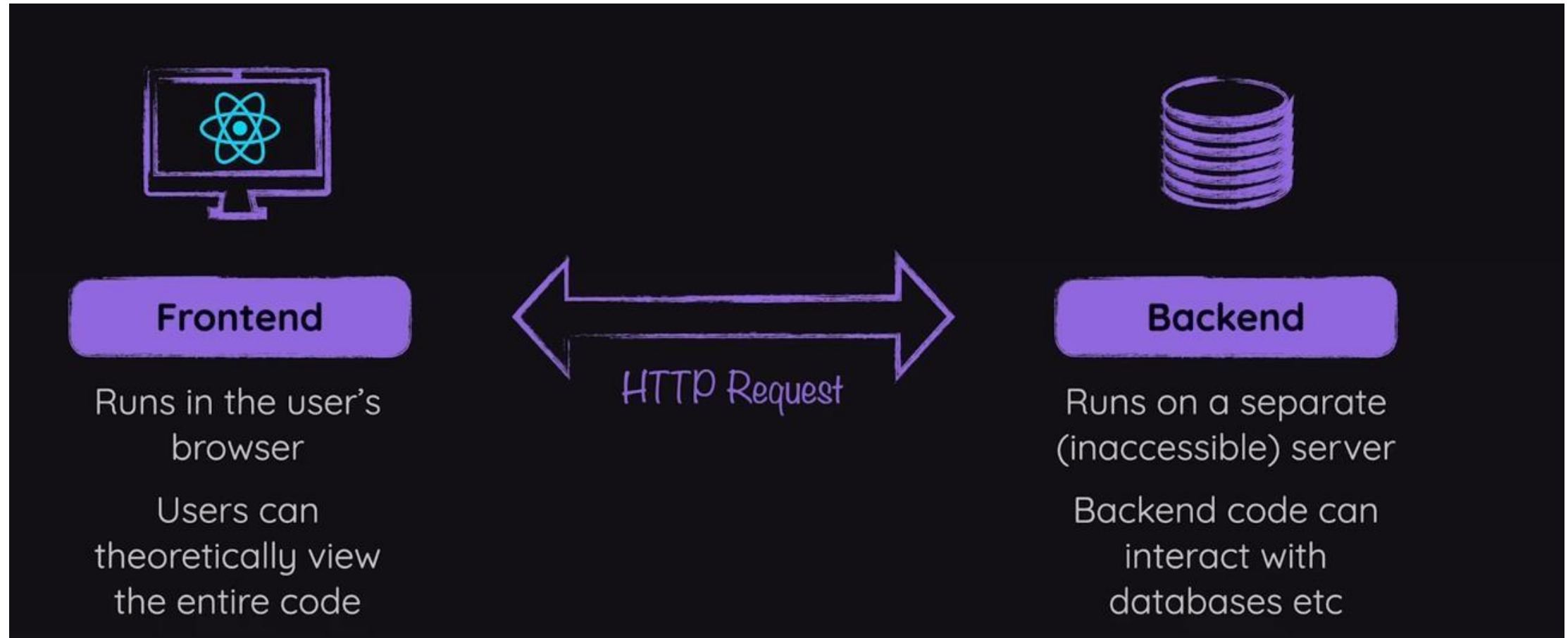
Quando conectamos a nossa aplicação a um servidor estamos a deixar o utilizador aceder a dados internos como passwords.

Ex: quando um formulário é submetido ele pode colocar *queries SQL* nesse formulário e com isso vulnerabilizar a nossa aplicação.

Por esta razão, **nunca devemos fazer um acesso directo do Client Side à Base de Dados!**

A interacção será então feita com um servidor de Backend que toma conta das ligações à Base de Dados.

Ligação a Bases de Dados & Segurança



Métodos HTTP

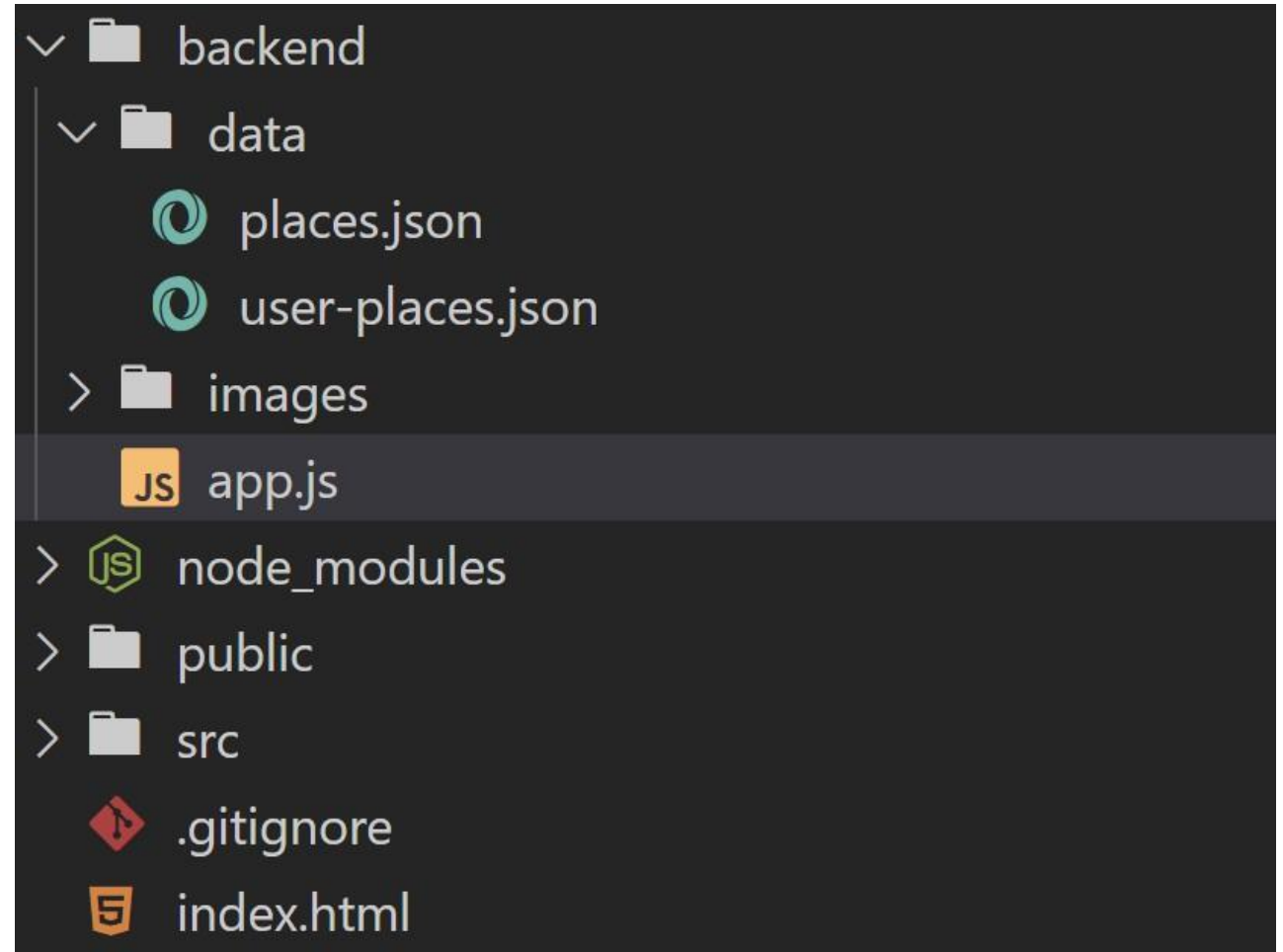
O protocolo HTTP define um conjunto de Métodos que indicam a acção a ser executada para um dado recurso. Mais informação [aqui](#).

HTTP Method	CRUD operation	Entire Collection (e.g. /users)	Specific Item (e.g. /users/{id})
GET	Read	200 (OK), list of entities. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single entity. 404 (Not Found), if ID not found or invalid.
POST	Create	201 (Created), Response contains response similar to GET /user/{id} containing new ID.	not applicable
PATCH	Update	Batch API	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	204 (No Content). 400(Bad Request) if no filter is specified.	204 (No Content). 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	not implemented	not implemented

Ligação a Backend

As ligações ao Servidor de *Backend* podem ser feitas internamente na nossa aplicação e ela torna-se *Full Stack*, ou podemos ligar a uma aplicação externa que pode ter a nossa linguagem ou qualquer outra (PHP, JAVA, etc).

Iremos então usar um Backend Interno para interagirmos com dados.



Ligação a Backend

A primeira coisa a ter em conta é que agora temos que ter dois servidores a correr:

- O primeiro, que compila e distribui o nosso Client Side
- O segundo, que compila e torna acessível o nosso Server Side

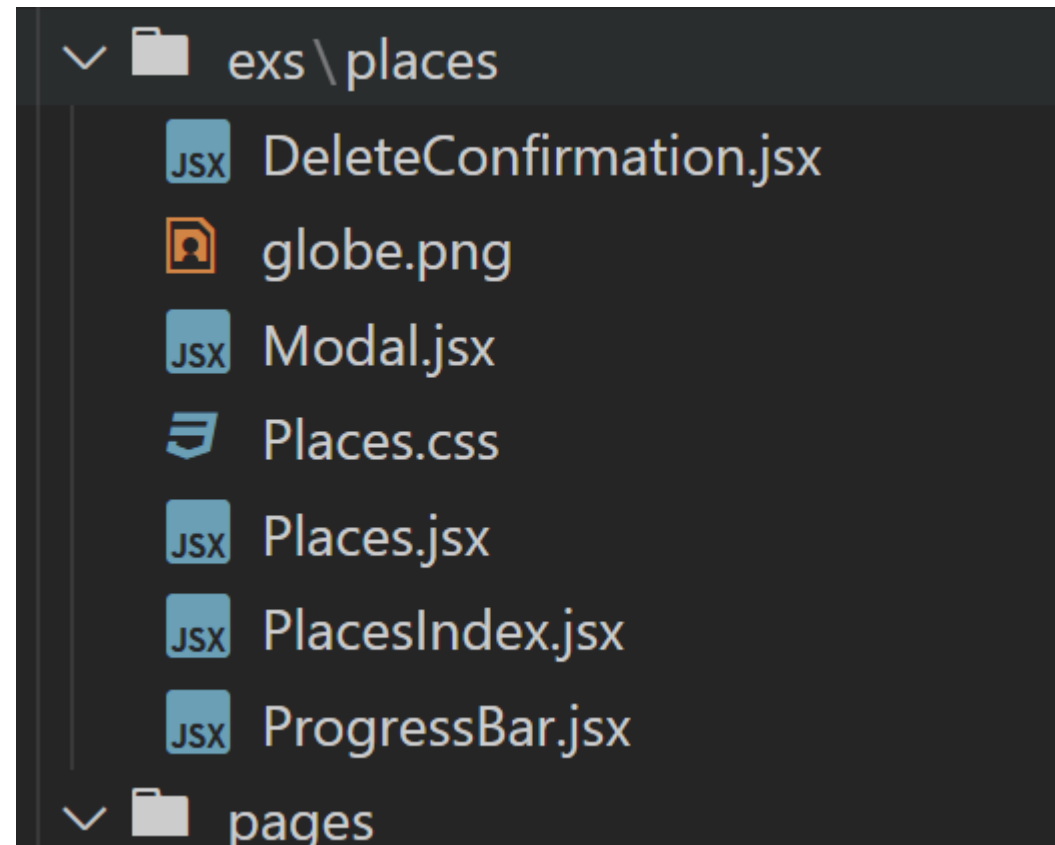
Para conseguirmos correr o servidor de Backend precisamos de instalar os pacotes usados: body-parser e express.

Ligação a Backend

Por fim, para correr o Server de Backend, no terminal, na raíz do projecto, execute:

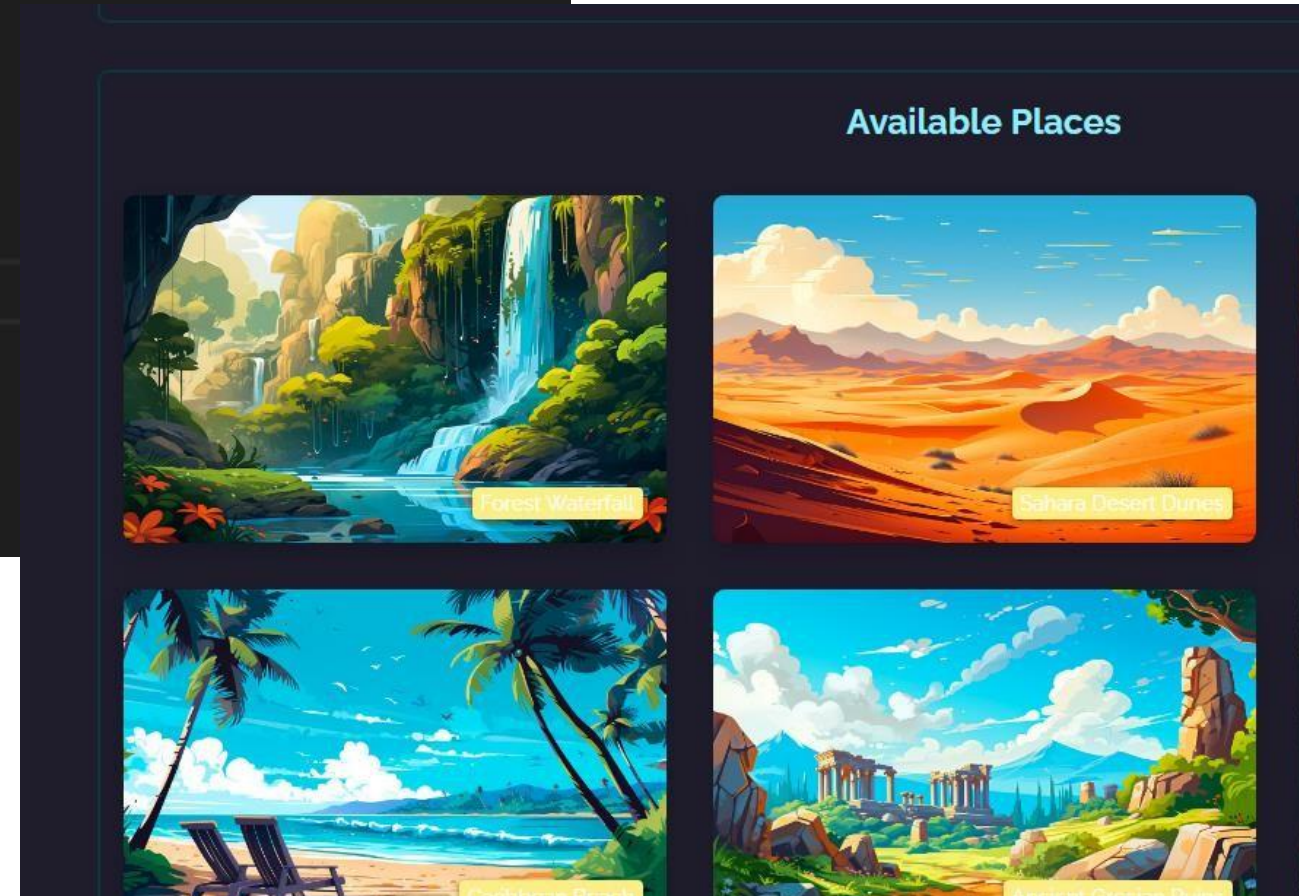
- `cd backend`
- `node app.js`

Iremos então montar a nossa aplicação de frontend com os elementos dados:



Ligação a Backend: preparar os dados e o uso do useEffect()

```
return (  
  <Places  
    title="Available Places"  
    places={availablePlaces}  
    fallbackText="No places available."  
    onSelectPlace={onSelectPlace}  
  />  
);
```



Ligação a Backend: preparar os dados e o uso do FETCH

Na raiz da nossa funcionalidade iremos buscar os dados e torná-los disponíveis para uso. Faremos então um pedido HTTP que nos leia os dados do servidor.

Para isso, usaremos disponível [o fetch\(\)](#), uma função do browser (também disponível em *vanilla JS*) e que é usada para fazer um pedido HTTP a um servidor.

Ligação a Backend: preparar os dados e o uso do FETCH

A função Fetch é uma [promise](#) que tem a seguinte estrutura:

```
fetch('http://localhost:3000/places')  
  .then((response) => response.json())  
  .then((resData) => {  
    console.log(resData.places);  
  })  
  
return (
```

Iremos precisar de actualizar o estado do nosso DOM com os nossos dados e neste caso o hook a usar será o [useEffect\(\)](#).

Ligação a Backend: preparar os dados e o uso do useEffect()

O useEffect permite controlar efeitos colaterais em componentes funcionais. É útil para:

- Fazer chamadas à API
- Sincronizar dados
- Manipular o DOM
- Criar e limpar eventos
- Controlar timers

Substitui os métodos de ciclo de vida (componentDidMount, componentDidUpdate e componentWillUnmount) das classes no React e faz com que o código seja executado assim que o estado da dependência mude (assim que tenhamos dados).

Evita ciclos infinitos pois a função deixa de ser executada quando recebe o que precisa.

Ligação a Backend: preparar os dados e o uso do useEffect()

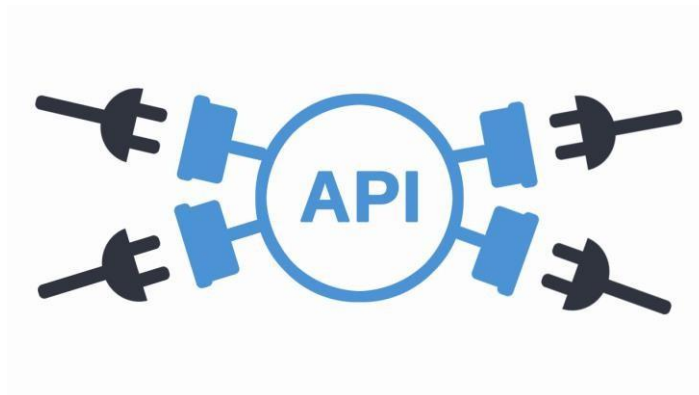
```
useEffect(() =>{  
  fetch('http://localhost:3000/places')  
    .then((response) => {return response.json()})  
    .then((resData) => {resData.places});  
}, []);
```


Ligação a Backend: preparar os dados e o uso do useEffect()

Usaremos agora o useState para indicar que há mudanças e podemos finalmente carregar os nossos dados!

```
export default function AvailablePlaces({ onSelectPlace }) {  
  const [availablePlaces, setAvailablePlaces] = useState([])  
  useEffect(() => {  
    fetch('http://localhost:3000/places')  
      .then((response) => {return response.json()})  
      .then((resData) => {  
        setAvailablePlaces(resData.places);  
      }  
    ), []);  
}
```


API



Para aprofundar o estudo do React, iremos usar o recurso de consultar uma API externa.

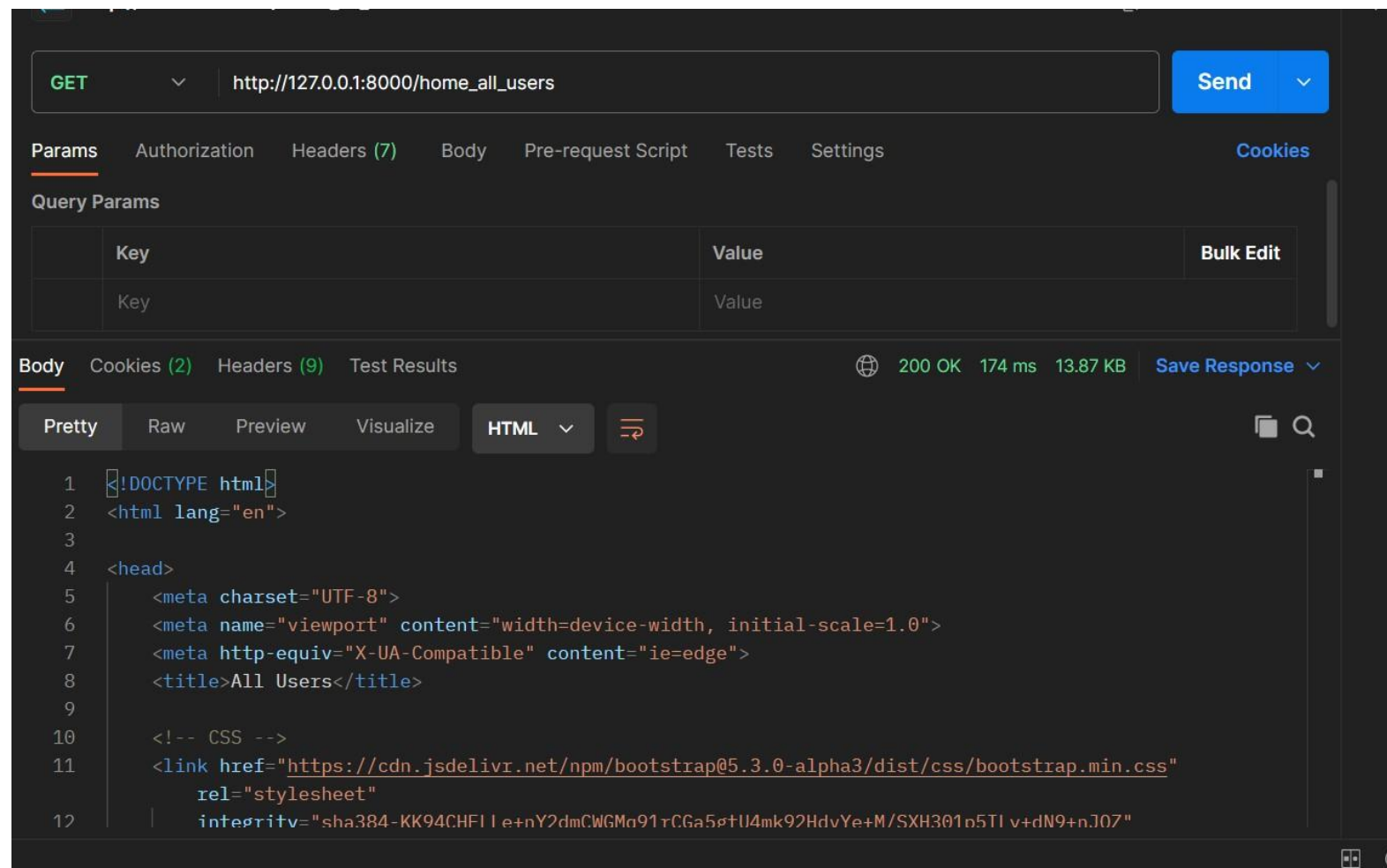
Application Programming Interface

- **Application:** Software que corre tarefas, como o Google Maps que nos dá direcções.
- **Programming:** onde damos instruções à aplicação para correr as tarefas para nós.
- **Interface:** o lugar onde as entidades comunicam um com a outra.

Pedaço de Software que permite que uma aplicação comunique com outra aplicação.

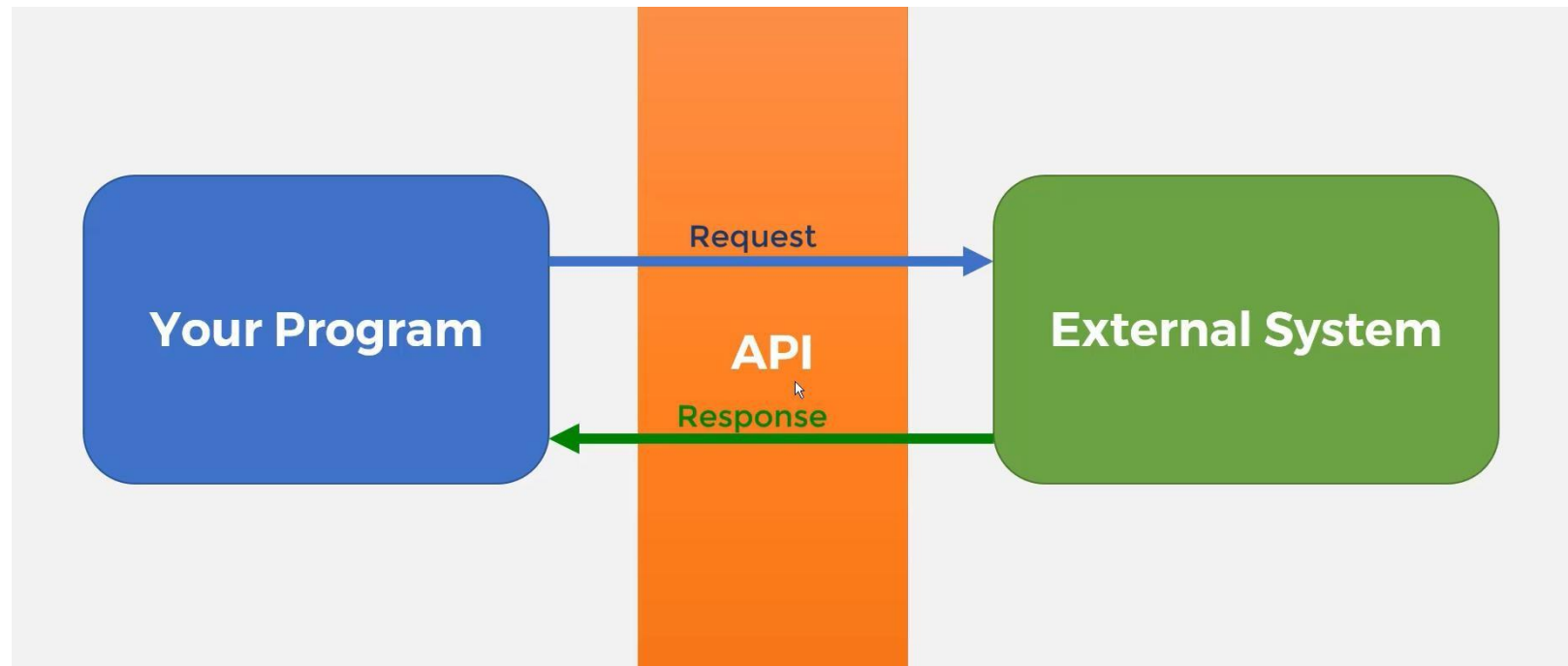
API – Postman

O melhor recurso para testar APIs
é o Postman e pode ser instalado
através do [site oficial](#) ou de
extensão no Visual Studio Code



APIs: Benefícios e Usos

- Proporciona aos programadores comandos base para que não tenham que escrever código de raiz
- Fazem com que a nossa aplicação possa comunicar com outra aplicação sem que tenhamos que saber como é que a outra está implementada
- Simplifica o desenvolvimento de aplicação, poupando tempo e dinheiro
- Interagem com outros Websites, obtendo dados em tempo real

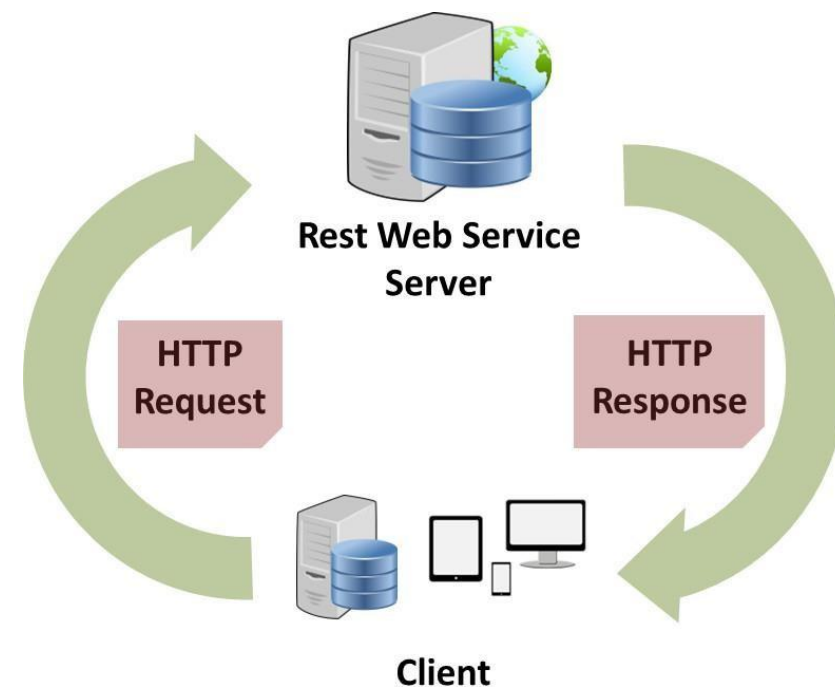


Web Services

Recurso que é tornado disponível através de uma rede, como por exemplo a internet.

Todos os Web Services são APIs mas nem todas as APIs são Web Services. Isto porque algumas APIs operam offline.

Um Web Service necessita de seguir um protocolo standard para transferir dados entre computadores. Normalmente, ele usa o HTTP.



API Endpoints



Um endpoint de API é normalmente o URL de um servidor ou serviço.

É o endereço de uma página ou recurso.

Iremos interagir com uma API fazendo um request e recebendo uma response.

Quando fazemos um request válido, a API responde dando-nos a localização do serviço ou recurso que queremos.

DEMO API

<http://open-notify.org/Open-Notify-API/ISS-Location-Now/>

[API Star Wars](#)

Pedidos HTTP

Existem 4 partes para um pedido HTTP:

1. Request Line
2. Headers
3. Blank Line
4. Body

Apenas a Request Line é obrigatória.

O propósito da Blank Line é separar os headers do body.

Pedidos HTTP

Request Line: linha inicial, onde incluimos o método HTTP a ser usado, o URI (Uniform Resource Indicator) do pedido e a versão do protocolo.

O URI é uma sequência de caracteres que indentificam o recurso.

Ex: GET /api/publishers HTTP/1.1.

Headers: lista de strings a ser enviada e recebida pelo cliente e Web Server.

Ex. Accept-Language: en-Us, fr, de

Body: dados associados ao o pedido ou resposta.

Content-Type e Content-Length headers especificam a natureza do Body.

Os Get normalmente não têm um body porque não precisamos de enviar informação, apenas os POST ou PUT, que marcam o texto a enviar.

Pedidos HTTP - Exemplo

POST /api/publishers HTTP/1.1

Host: abc.com

Content-Type: application/json

{

"Name": "Daniel Tait",

"Age": 32

}

← Request Line

} Headers

← Blank Line

} Message Body

HTTP Status Codes

[Documentação](#)

Status Code - Class Description	Translation
1xx: Informational response These codes let the client know that their request was received and understood by the server. But you need to wait for a final response as the request is still being processed.	<i>Hold on, we're working on it.</i>
2xx: Success This class of codes means that the request was received, understood, and accepted.	<i>Congratulations! Everything worked.</i>
3xx: Redirection These codes indicate that further action must be taken by the client in order to complete the request. Many of these codes are used in URL redirection.	<i>More work required.</i>
4xx: Client Error This class of codes means that the request contains incorrect syntax or cannot be fulfilled.	<i>You (the client) made a mistake.</i>
5xx: Server Error These codes means the server failed to fulfil an apparently valid request	<i>We (the server) made a mistake.</i>

{JSON}

- JavaScript Object Notation
- É o formato de ficheiros para armazenar e transmitir dados mais usado actualmente. É conhecido por ser leve e fácil de ler / escrever.
- Apesar de derivar do JS o JSON não é só usado para JS, mas também em outras APIs que forneçam dados para Web como Python, PHP, etc..
- O Objecto de JSON não pode ser usado directamente em JS, temos primeiro que o converter para um objecto JS através do método `JSON.parse(objectoJSON)`.

[documentação](#)

```
"browsers": {  
  "firefox": {  
    "name": "Firefox",  
    "pref_url": "about:config",  
    "releases": {  
      "1": {  
        "release_date": "2004-11-09",  
        "status": "retired",  
        "engine": "Gecko",  
        "engine_version": "1.7"  
      }  
    }  
  }  
}
```

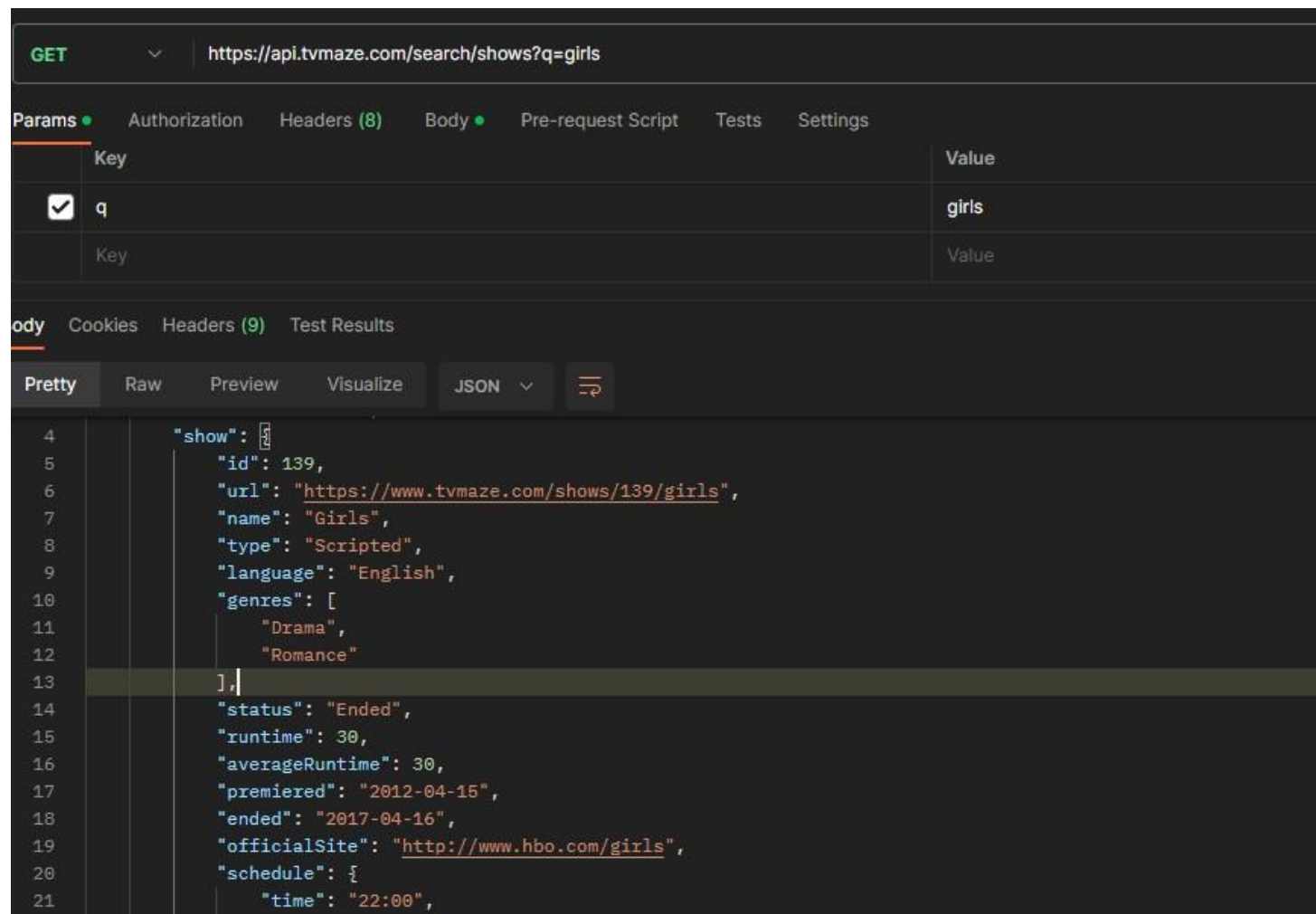
JSON Data Types

- Number 28 9.57 -30 4.5e6
- String "Bob" "Hello World!" "phone 911 now"
- Boolean true false
- Null null
- Array [1, 2, 5] ["Cat", "Dog"]
- Object {"name": "Ann", "age": 21, "cat": "Luna"}

Query Strings

- É um parâmetro de pesquisa que enviamos no url e que envia ao Backend a informação do que queremos procurar.
- Podemos ter vários parâmetros de pesquisa

[exemplo](#)



The screenshot shows a REST client interface with a GET request to `https://api.tvmaze.com/search/shows?q=girls`. The 'Params' tab is active, showing a single parameter `q` with the value `girls`. The 'Body' tab is also active, displaying the JSON response in 'Pretty' format. The response is a JSON array with one object representing a TV show.

```
4  {
5    "show": {
6      "id": 139,
7      "url": "https://www.tvmaze.com/shows/139/girls",
8      "name": "Girls",
9      "type": "Scripted",
10     "language": "English",
11     "genres": [
12       "Drama",
13       "Romance"
14     ],
15     "status": "Ended",
16     "runtime": 30,
17     "averageRuntime": 30,
18     "premiered": "2012-04-15",
19     "ended": "2017-04-16",
20     "officialSite": "http://www.hbo.com/girls",
21     "schedule": {
22       "time": "22:00",
```

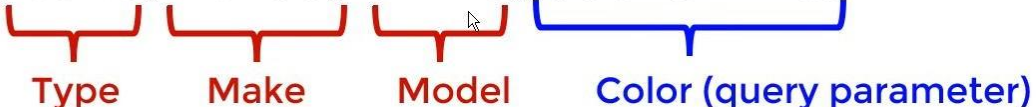
Path Parameters

- São partes variáveis de um caminho URL

`https://abc.com/publishers/george/articles?limit=50`

- Os path parameters são usados para identificar enquanto os query parameters são usados para filtrar os recursos.

GET `/cars/honda/civic/?color=red`


Type Make Model Color (query parameter)

Consultar uma API Externa

Com o React podemos [consultar uma API](#) externa e usar os dados para mostrar na nossa aplicação. Por exemplo, usar a API do Star Wars para mostrar os filmes com o endpoint `/people/`

```
const [people, setPeople] = useState([]);

useEffect(() => {
  fetch('https://swapi.dev/api/people/')
    .then((response) => {return response.json()})
    .then((resData) => {
      setPeople(resData.results);
    });
}, []);
```

Consultar uma API Externa

```
return (  
<section>  
  {(!people || people.length === 0) && <p className="fallback-text">Ups, não temos  
    personagens disponíveis</p>}  
  {(people && people.length) > 0 && (  
    <ul className="places">  
      {people.map((item) => (  
        <li key={item.id} className="place-item">  
          <p><b>{item.name}</b>: </b>{item.birth_year}, {item.gender}</p>  
        </li>  
      )  
    )  
  }  
  </ul>  
  )  
</section>  
)  
.
```


Consultar uma API Externa

Layout Master		
Aqui tens as personagens Star Wars		
Luke Skywalker: 19BBY, male	C-3PO: 112BBY, n/a	R2-D2: 33BBY, n/a
Darth Vader: 41.9BBY, male	Leia Organa: 19BBY, female	Owen Lars: 52BBY, male
Beru Whitesun lars: 47BBY, female	R5-D4: unknown, n/a	Biggs Darklighter: 24BBY, male
Obi-Wan Kenobi: 57BBY, male		

Exercício API



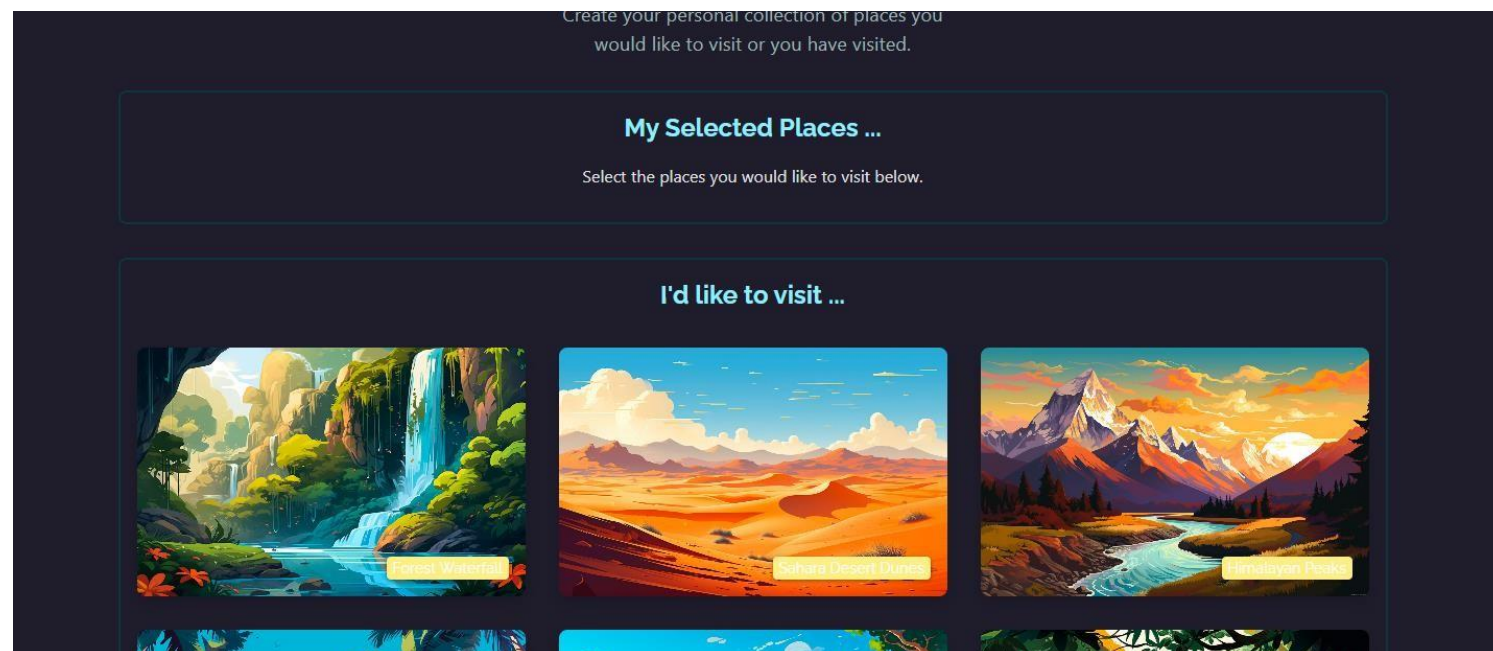
Usando o que aprendemos em relação às consultas ao Backend e APIS crie uma página na nossa aplicação uma página que mostre uma lista com todos os filmes do StarWars e os seguintes atributos: title, opening_crawl, release_date.

A lista deve ser mostrada numa rota da nossa aplicação específica para o fim.

Enviar dados: o PUT

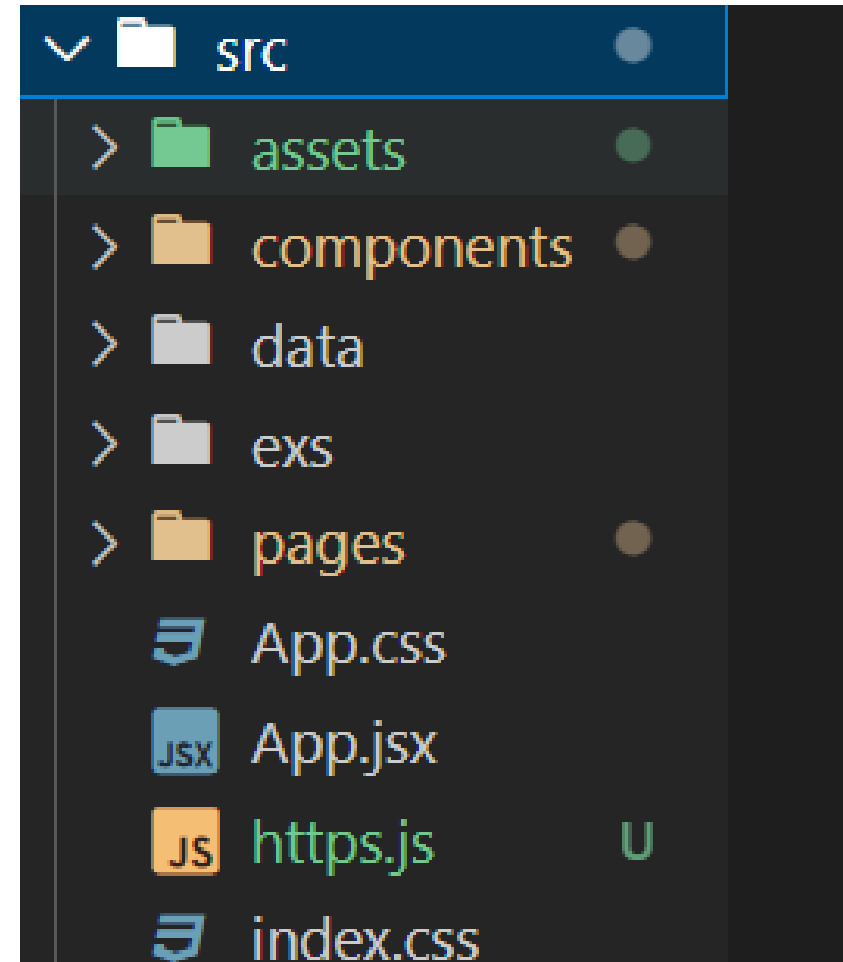
Iremos preparar agora a nossa aplicação não só para ler pedidos dos Backend como também para guardar dados que fiquem registados no backend.

Para tal, preparemos o front end para guardar um array dos lugares que eu escolhi.



Organizar pedidos a API

Iremos prosseguir para actualizar a nossa api de user-places usando o método PUT. Para tal, e uma vez que não vamos precisar de usar *Hooks* neste caso, criaremos um ficheiro `http.js` para colocar o update e manter o código organizado.



API: PUT

```
<main>
  <Places
    title="My selected Places"
    fallbackText="Select the places you would like to visit
    below."
  />
  <Places
    title="I'd like to visit ..."
    fallbackText="Select the places you would like to visit
    below."
    places={places}
    onSelectPlace={handleSelectPlace}
  />
</main>
```

API: PUT

Por último, chamamos a nossa função que toma conta do adicionar sítios novos.

```
function handleSelectPlace(selectedPlace) {  
  setUserPlaces((prevPickedPlaces) => {  
    if (!prevPickedPlaces) {  
      prevPickedPlaces = [];  
    }  
    if (prevPickedPlaces.some((place) => place.id === selectedPlace.id)) {  
      return prevPickedPlaces;  
    }  
    return [selectedPlace, ...prevPickedPlaces];  
  });  
  updatePlaces([selectedPlace, ...userPlaces]);  
}
```

API: PUT

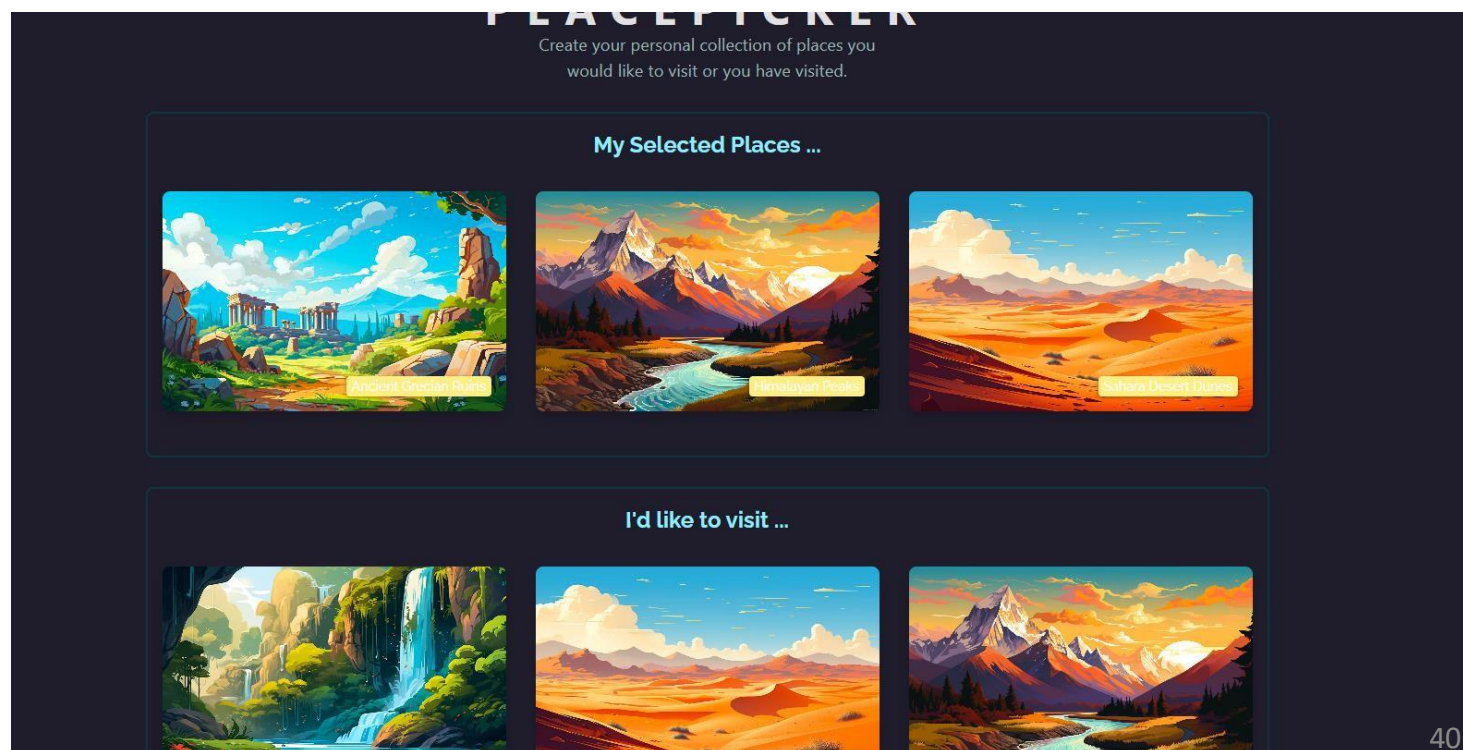
No ficheiro http.js iremos então colocar a função de update.

De notar que esta função de *update* deverá ser uma função assíncrona, porque ela só enviará os dados assim que se conseguir conectar com a API.

```
export async function updatePlaces(userPlaces) {  
  const response = await fetch("http://localhost:3000/user-places", {  
    method: "PUT",  
    body: JSON.stringify({ places: userPlaces }),  
    headers: {  
      "Content-Type": "application/json",  
    },  
  });  
  
  const data = await response.json();  
  if (!response.ok) {  
    console.log("Failed");  
  }  
  
  return data.message;  
}
```


API: Actualização do Delete

- Por último, iremos actualizar o código do Backend quando eu apagar um lugar. Por exemplo, se eu colocar lugares nos lugares escolhidos ele guarda no Backend e já não é possível remover.



API: Delete

Para sabermos o elemento onde estamos a clicar para que saber os atributos a eliminar usaremos o *hook* `useRef`.

A função `useRef()` cria uma referência mutável que persiste entre renderizações, sem causar re-renderizações quando o valor muda.

```
const selectedPlace = useRef(null);
```

```
<Places  
  title="My selected Places"  
  fallbackText="Select the places you would like to visit  
  below."  
  places={userPlaces}  
  onSelectPlace={handleStartRemovePlace}  
/>  
<Places
```

```
function handleStartRemovePlace(place) {  
  selectedPlace.current = place;  
  handleRemovePlace();  
}
```

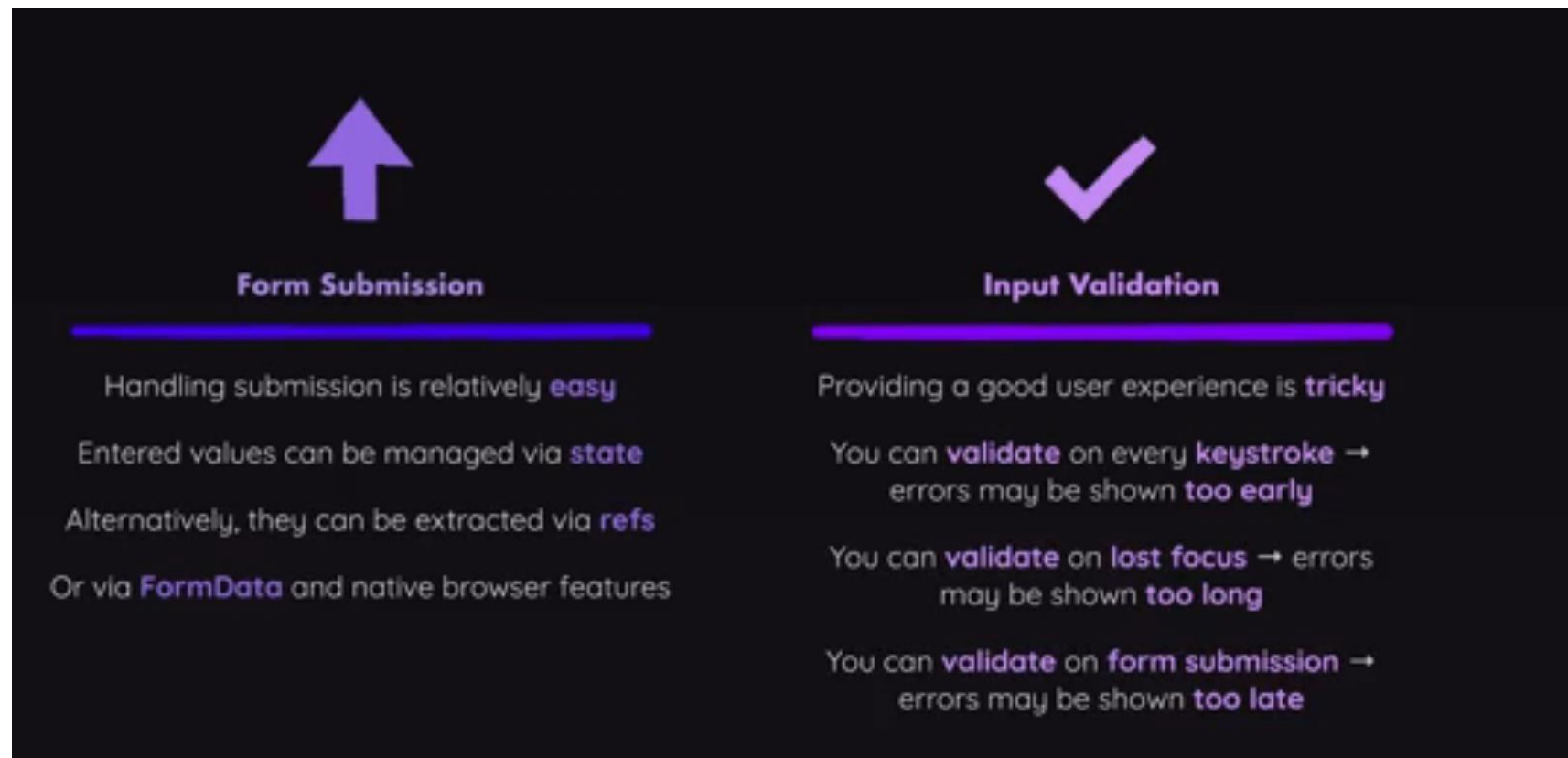
API: Delete

Na função que toma conta do Delete configuramos então o update para que ele nos actualize os lugares sem o que acabamos de remover.

```
const handleRemovePlace = useCallback(async function  
handleRemovePlace() {  
  setUserPlaces((prevPickedPlaces) =>  
    prevPickedPlaces.filter((place) => place.id !== selectedPlace.  
      current.id)  
  );  
  
  updatePlaces(  
    userPlaces.filter((place) => place.id !== selectedPlace.  
      current.id)  
  );  
  setModalIsOpen(false);  
}, [userPlaces]);
```

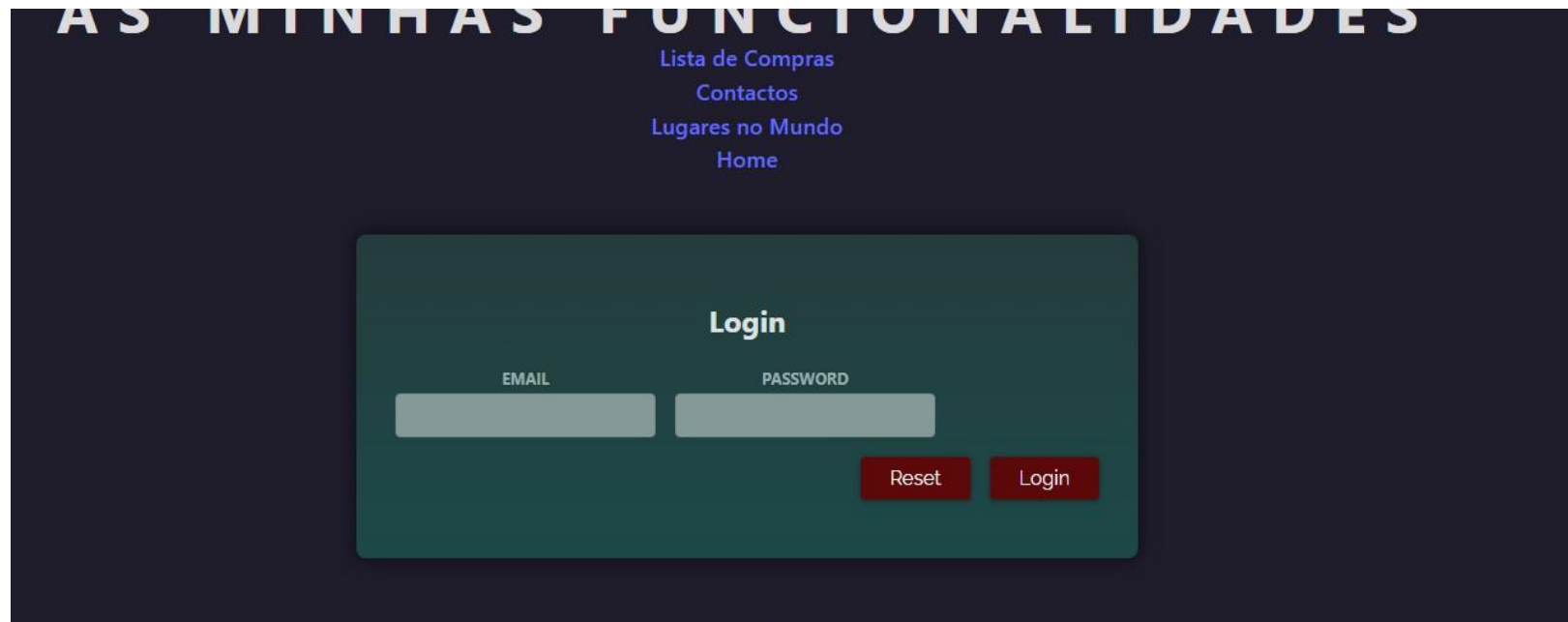
Tratamento de Formulários

Em quase todas as aplicações temos um formulário. Seja para fazer um novo registo ou acrescentar um novo produto, a certo ponto o utilizador irá preencher um formulário e no react teremos que o receber e validar.



Tratamento de Formulários

Iremos simular o tratamento da dados através de um formulário de login presente na pasta `exs->auth->login` que iremos adicionar a uma rota para login.



AS MINHAS FUNCIONALIDADES

- Lista de Compras
- Contactos
- Lugares no Mundo
- Home

Login

EMAIL

PASSWORD

Tratamento de Formulários

A primeira coisa a ter em atenção é o facto de termos um DOM virtual no react. Logo, quando existe uma alteração de valores do lado do user teremos que actualizar o estado do nosso DOM virtual. Criaremos um estado que receberá um objecto com o email e password do user e não um valor único.

```
export default function Login() {  
  
  const [enteredValues, setEnteredValues] = useState({  
    email: '',  
    password: '',  
  });  
}
```

Tratamento de Formulários

Em seguida preparamos o formulário para chamar a função `handleSubmit()` quando é submetido e definimos a função.

```
return (  
  <form onSubmit={handleSubmit}>  
    <h2>Login</h2>  
  
    <div className="control-row">  
      <div className="control no-margin">  
        <label htmlFor="email">Email  
        <input id="email" type="email" />  
      </div>  
    </div>  
  </form>  
)  
  
function handleSubmit(event){  
  event.preventDefault();  
  console.log(enteredValues);  
}
```

Tratamento de Formulários

Vamos agora criar e usar a função que pegará nos dados quando houver uma alteração do user com o evento onChange e os colocará no nosso doVirtual com a alteração do estado.

```
<div className="control no-margin">
  <label htmlFor="email">Email</label>
  <input id="email" type="email" name="email" onChange={(event) => handleInputChange('email', event.target.value)}
  value={enteredValues.email}/>
</div>
```

Nota: a função setState tem incorporado um argumento a que podemos aceder para ter os valores anteriores. Desta forma quando actualizamos o objecto ele não 'perde' dados.

```
function handleInputChange(indentifier, value){
  setEnteredValues((prevValues) =>({
    ...prevValues,
    [indentifier]: value,
  }));
}
```


Tratamento de Formulários: outras formas

Iremos agora fazer o nosso registo de user e para tal usaremos uma função incorporada do próprio browser, o FormData.

Para usar esta função todos os atributos de name='nossonome' devem estar preenchidos.

```
function handleSubmit(event){  
  event.preventDefault();  
  const formData = new FormData(event.target);  
  const data = Object.fromEntries(formData.entries());  
  console.log(data);  
}
```

Tratamento de Formulários: validação

Em web já aprendemos que existem validação já incorporadas no html.

Temos o caso do required, do type="email" e do atributo min para definir tamanho obrigatório da pass, por exemplo.

Mas a estas validações podemos e devemos acrescentar as nossas próprias, como por exemplo validar se a password e a confirmação da password condizem. Para isso podemos criar um estado para a validação e trabalhar os erros com ele.

Tratamento de Formulários: validação

```
export default function Signup() {  
  const [passwordsAreNotEqual, setPasswordsNotEqual] = useState(false);  
  
  function handleSubmit(event){  
    event.preventDefault();  
    const formData = new FormData(event.target);  
    const data = Object.fromEntries(formData.entries());  
  
    if(data.password !== data.confirmPassword){  
      setPasswordsNotEqual(true);  
      return;  
    };  
  }  
}
```

```
<div className="control">  
  <label htmlFor="confirm-password">Confirm Password</label>  
  <input  
    id="confirm-password"  
    type="password"  
    name="confirmPassword"  
  />  
  {passwordsAreNotEqual &&  
    <div>Passwords must match</div>  
  }  
</div>  
</div>
```

Tratamento de Formulários: Inserção de users no json

Na nossa função de trabalhar o formulário, chamamos o pedido HTTP de POST criado no BE para enviar o nosso novo user.

```
const user = {  
  email: data.email,  
  password: data.password,  
  firstName: data['first-name'],  
  lastName: data['last-name'],  
  role: data.role,  
  termsAccepted: data.terms === "on"  
};  
  
const response = fetch("http://localhost:3000/signup", {  
  method: "POST",  
  headers: { "Content-Type": "application/json" },  
  body: JSON.stringify(user),  
});
```

Tratamento de Formulários: Inserção de users no json

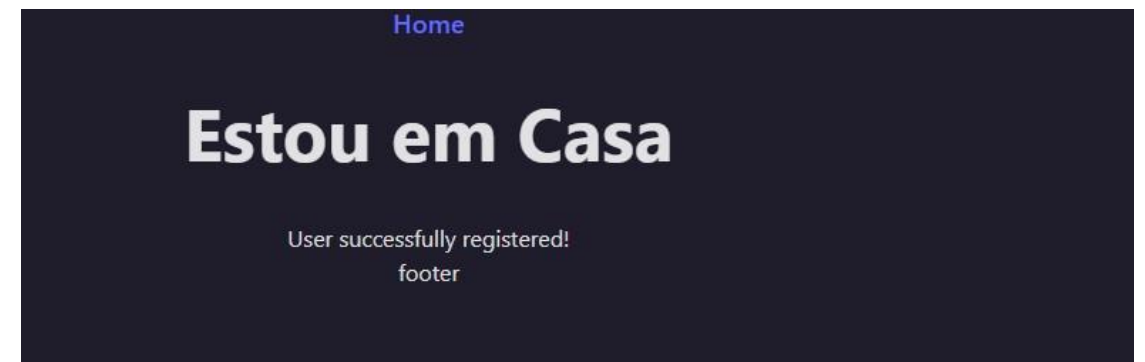
Por último, instalamos o pacote **react-router-dom** e para navegar para a HomePage e informar que foi inserido o user com sucesso.

```
import { useNavigate } from 'react-router-dom';
```

```
const navigate = useNavigate();
```

```
navigate("/", { state: { message: "User successfully registered!" } });
```

```
import { useLocation } from 'react-router-dom';  
  
export default function HomePage() {  
  const location = useLocation();  
  const message = location.state?.message;  
  return(  
    <div>  
      <h1>Estou em Casa</h1>  
      {message && <div>{message}</div>}  
    </div>  
  );  
}
```



Funcionalidade 2



Realize e entregue a tarefa proposta.

Recursos

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- <https://react.dev/>