Rest API

API - это то, что можно использовать, чтобы загрузить React-приложения данными. Есть определенные операции, которые нельзя выполнить на стороне клиента, поэтому эти операции выполняются на стороне сервера. Затем можно использовать API-интерфейсы для использования данных на стороне клиента.

API состоят из набора данных, которые часто представлены в формате JSON с указанными конечными точками. Когда мы получаем доступ к данным из API, мы хотим получить доступ к определенным конечным точкам в этой структуре API. Также можно сказать, что API - это соглашение между двумя службами в форме запроса и ответа. Код является побочным продуктом. Он также содержит условия обмена данными.

В React доступны различные способы использования REST API в приложениях, в том числе использование встроенного JavaScript- метода fetch() и Axios, который является HTTP-клиентом на основе промисов для браузера и Node.js.

Что такое REST API

REST API - это API, который структурирован в соответствии с REST структурой для API. REST (Representational State Transfer) означает «передача состояния представления». Он состоит из различных правил, которым следуют разработчики при создании API.
На сегодняшний день это самые популярные и гибкие API-интерфейсы в Интернете. Клиент отправляет запросы на сервер в виде данных. Сервер использует этот клиентский ввод для запуска внутренних функций и возвращает выходные данные обратно клиенту.

Преимущества REST API

- Прост в освоении и понимании;
- Предоставляет разработчикам возможность организовывать сложные приложения в простые ресурсы;
- Внешним клиентам можно построить REST API без каких-либо сложностей;
- Очень легко масштабируется;
- REST API не зависит от языка или платформы, то есть может использоваться на любом языке или работать на любой платформе.

Пример ответа REST API

То, как структурирован REST API, зависит от продукта, для которого он был создан, но при этом должны соблюдаться правила REST.

```
{
"login": "hacktivist123",
"id": 26572907,
"node_id": "MDQ6VXNlcjI2NTcyOTA3",
"avatar_url": "https://avatars3.githubusercontent.com/u/26572907?v=4",
"gravatar_id": "",
"url": "https://api.github.com/users/hacktivist123",
"html_url": "https://github.com/hacktivist123",
"followers_url": "https://api.github.com/users/hacktivist123/followers",
"following_url": "https://api.github.com/users/hacktivist123/following{/other_user}",
"gists_url": "https://api.github.com/users/hacktivist123/gists{/gist_id}",
"starred_url": "https://api.github.com/users/hacktivist123/starred{/owner}{/repo}",
"subscriptions_url": "https://api.github.com/users/hacktivist123/subscriptions",
"organizations_url": "https://api.github.com/users/hacktivist123/orgs",
"repos_url": "https://api.github.com/users/hacktivist123/repos",
"events_url": "https://api.github.com/users/hacktivist123/events{/privacy}",
"received_events_url": "https://api.github.com/users/hacktivist123/received_events",
"type": "User",
"site_admin": false,
"name": "Shedrack akintayo",
"company": null,
"blog": "https://sheddy.xyz",
"location": "Lagos, Nigeria ",
"email": null,
"hireable": true,
"bio": "☕ Software Engineer | | Developer Advocate|| 💛 Everything JavaScript",
"public_repos": 68,
"public_gists": 1,
"followers": 130,
"following": 246,
"created_at": "2017-03-21T12:55:48Z",
"updated_at": "2020-05-11T13:02:57Z"
}
```


Приведенный пример, взят из REST API Github, когда выполняется GET-запрос к следующей конечной точке https://api.github.com/users/hacktivist123. Он возвращает все сохраненные данные о пользователе с именем hacktivist123. С помощью этого ответа можно решить, каким образом мы будем отображать его в React-приложении.

Применение API с помощью Fetch API

API fetch() является встроенным методом JavaScript, предназначенным для получения ресурсов от сервера или конечной точки API.

Метод fetch() API всегда принимает обязательный аргумент, представляющий собой путь или URL-адрес ресурса, который вы хотите получить. Он возвращает промис, который указывает на ответ от запроса, независимо от того, был ли запрос успешным или нет. При желании вы также можете передать объект параметров инициализации в качестве второго аргумента.

После получения ответа доступно несколько встроенных методов, позволяющих определить, каково содержимое тела и как его следует обрабатывать.

```
let promise = fetch(url, [options])
```

url – URL для отправки запроса.
options – дополнительные параметры: метод, заголовки и так далее.

Без options это простой GET-запрос, скачивающий содержимое по адресу url

Параметры для Fetch API

resource

Путь к ресурсу, который вы хотите получить, это может быть либо прямая ссылка на путь к ресурсу, либо объект запроса.

init

Объект, содержащий любые пользовательские настройки или учетные данные, которые вы хотите предоставить для запроса fetch(). Ниже приведены некоторые из возможных параметров, которые могут содержаться в объекте init:

- **method**

  для указания метода HTTP-запроса, например, GET, POST и т. д.
- **headers**

  для указания любых заголовков, которые вы хотели бы добавить к запросу, обычно содержится в объекте или литерале объекта.
- **body**

  для определения тела, которое вы хотите добавить к запросу: это может быть объект Blob, BufferSource, FormData, URLSearchParams, USVString
- **mode**

  для задания режима, который необходимо использовать для запроса, например, cors, no-cors или same-origin.
- **credentials**

  необходим для указания учетных данных запроса, которые вы хотите использовать для запроса, этот параметр должен быть предоставлен, если вы используете автоматическую отправку файлов cookie для текущего домена.

Процесс получения ответа обычно происходит в два этапа.

Во-первых, promise выполняется с объектом встроенного класса Response в качестве результата, как только сервер пришлёт заголовки ответа.

На этом этапе можно проверить статус HTTP-запроса и определить, выполнился ли он успешно, а также посмотреть заголовки, но пока без тела ответа.

Промис завершается с ошибкой, если fetch не смог выполнить HTTP-запрос, например при ошибке сети или если нет такого сайта. HTTP-статусы 404 и 500 не являются ошибкой.

Можно увидеть HTTP-статус в свойствах ответа:

- **status** – код статуса HTTP-запроса, например 200.
- **ok** – логическое значение: будет true, если код HTTP-статуса в диапазоне 200-299.

```
let response = await fetch(url);

if (response.ok) { // если HTTP-статус в диапазоне 200-299
  // получаем тело ответа (см. про этот метод ниже)
  let json = await response.json();
} else {
  alert("Ошибка HTTP: " + response.status);
}
```

Во-вторых, для получения тела ответа нам нужно использовать дополнительный вызов метода.

Response предоставляет несколько методов, основанных на промисах, для доступа к телу ответа в различных форматах:

response.text() – читает ответ и возвращает как обычный текст,
response.json() – декодирует ответ в формате JSON,
response.formData() – возвращает ответ как объект FormData
response.blob() – возвращает объект как Blob (бинарные данные с типом),
response.arrayBuffer() – возвращает ответ как ArrayBuffer (низкоуровневое представление бинарных данных),
.

Например, получим JSON-объект с последними коммитами из репозитория на GitHub:

```
<!DOCTYPE html>
<script>
"use strict";

(async () => {
let url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.
  info/commits';
let response = await fetch(url);

let commits = await response.json(); // читаем ответ в формате JSON

alert(commits[0].author.login);
})()
</script>
```

То же самое без await, с использованием промисов:

```
<!DOCTYPE html>
<script>
"use strict";

fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/
  commits')
  .then(response => response.json())
  .then(commits => alert(commits[0].author.login));
</script>
```

Базовый синтаксис для использования Fetch API

Простой запрос на выборку может выглядеть следующим образом:

```
fetch('https://api.github.com/users/hacktivist123/repos')
  .then(response => response.json())
  .then(data => console.log(data));
```

В данном коде извлекаются данные из URL-адреса, который возвращает данные в формате JSON, а затем выводим их в консоль. Простейшая форма использования fetch() часто принимает только один аргумент - путь к ресурсу, который вы хотите получить, и затем возвращает промис, содержащий ответ на запрос на выборку. Этот ответ является объектом.

Ответ - это обычный HTTP-ответ, а не фактический JSON. Другими словами, чтобы получить содержимое тела JSON из ответа, нам нужно изменить ответ на фактический JSON, используя в ответе метод json().

Использование Fetch API в приложениях React

Использование Fetch API в React-приложениях - это стандартный способ, которым мы использовали бы Fetch API в JavaScript, синтаксис не изменится. Единственная проблема - решить, где выполнить запрос на выборку в React- приложении. Большинство запросов на выборку или HTTP-запросы любого рода обычно выполняются в React Component.

Запрос может быть выполнен либо внутри метода жизненного цикла, если компонент является компонентом класса, либо внутри хука React useEffect(), если компонент является функциональным компонентом.

Например, в приведенном ниже коде выполним запрос на выборку внутри компонента класса, а это означает, что мы должны сделать это внутри метода жизненного цикла. В этом конкретном случае наш запрос на выборку будет выполнен внутри метода жизненного цикла componentDidMount, потому что мы ж [illegible] компонента React.

```
import React from 'react';

class myComponent extends React.Component {
  componentDidMount() {
    const apiUrl = 'https://api.github.com/users/hacktivist123/repos';
    fetch(apiUrl)
      .then((response) => response.json())
      .then((data) => console.log('This is your data', data));
  }
  render() {
    return <h1>my Component has Mounted, Check the browser 'console' </h1>;
  }
}
export default myComponent;
```

Метод fetch() принимает путь к ресурсу, который мы хотим извлечь, он присваивается переменной с именем apiUrl. После завершения запроса на выборку возвращается промис, содержащий объект ответа. Затем мы извлекаем из ответа содержимое тела JSON, используя метод json(), и, наконец, выводим окончательные данные из промиса в консоль.

Что такое адрес API и почему он важен?

Адреса API – это конечные точки взаимодействия в системе связи API. К ним относятся URL-адреса серверов, службы и другие конкретные цифровые местоположения, откуда информация отправляется и принимается между системами. Адреса API имеют решающее значение для предприятий по двум основным причинам.

1. Безопасность
Адреса API делают систему уязвимой для атак. Мониторинг API имеет решающее значение для предотвращения ненадлежащего использования.

2. Производительность
Адреса API, особенно с высоким трафиком, могут создавать узкие места и влиять на производительность системы.

Как обезопасить REST API?
Все API должны быть защищены посредством надлежащей аутентификации и мониторинга. Описание двух основных способов защиты безопасности REST API см. ниже.

1. Токены аутентификации
Они используются для авторизации пользователей для выполнения вызова API. Токены аутентификации проверяют, являются ли пользователи теми, за кого они себя выдают, и что у них есть права доступа для этого конкретного вызова API. Например, при входе на почтовый сервер почтовый клиент использует токены аутентификации для безопасного доступа.

2. Ключи API
Ключи API проверяют программу или приложение, выполняющее вызов API. Они идентифицируют приложение и гарантируют, что оно имеет права доступа, необходимые для выполнения конкретного вызова API. Ключи API не так безопасны, как токены, но они позволяют осуществлять мониторинг API для сбора данных об использовании. Возможно, вы заметили длинную строку символов и цифр в URL-адресе вашего браузера при посещении разных веб-сайтов. Эта строка представляет собой ключ API, который веб-сайт использует для выполнения внутренних вызовов API.

Создание API приложения погоды

Создадим 3 компонента для отображения информации на странице:

1. Info.jsx – для отображения информации о приложении
2. Form.jsx – с формой для ввода города и отправки запроса
3. Weathe.jsx – для вывода результатов запроса на API сервер

Info.jsx — src > components > Info.jsx > ...

```jsx
import React from 'react'

export default function Info() {
  return (
    <h2>Погода</h2>
  )
}
```

Form.jsx — src > components > Form.jsx > ...

```jsx
import React from 'react';

const Form = () => {
    return (
        <form>
            <input type="text" name='city' placeholder='город' />
            <button type='submit'>Получить погоду</button>
        </form>
    );
}

export default Form;
```

Weather.jsx — src > components > Weather.jsx > Weather

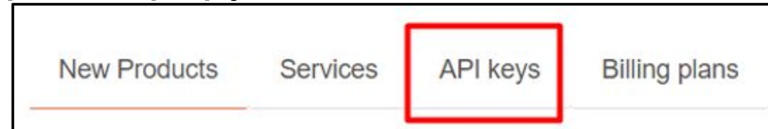
```jsx
import React from 'react';

const Weather = () => {
    return (
        <div>

            <div>
                <p>Местонахождение: Город -   Страна - </p>
                <p>Температура </p>
                <p>Восход солнца </p>
                <p>Закат солнца </p>
            </div>

        </div>
    );
}

export default Weather;
```

Импортируем их в App.jsx и подключим хук useState для добавления состояний данных

```
import Info from './components/Info';
import Form from './components/Form';
import Weather from './components/Weather';
import React, { useState } from 'react';
```

Перейдем на сайт https://api.openweathermap.org и зарегистрируемся там. ОБЯЗАТЕЛЬНО: подтвердить почту. Иначе api key будет не активным

Во вкладке API key копируем ключ для подключения

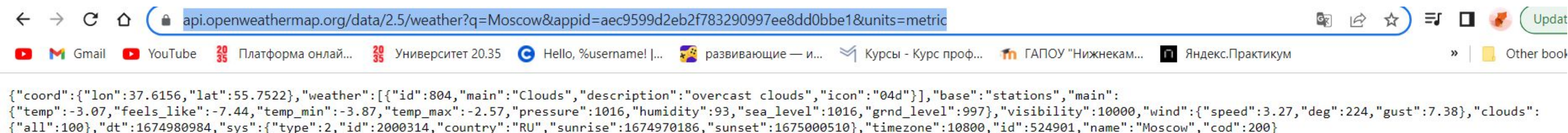


В самом низу переходим во вкладку Current and Forecast APIs. Где можно посмотреть возможности подключения по запросам.

```
https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={API key}
```

https://api.openweathermap.org/data/2.5/weather?q=Moscow&appid=aec9599d2eb2f783290997ee8dd0bbe1&units=metric

Данный запрос позволит увидеть данные по городу Москва.



Добавим переменную для хранения API ключа

```
const API_KEY = 'aec9599d2eb2f783290997ee8dd0bbe1';
```

```
const gettingWeather = async (event) => {
  event.preventDefault();
  const api_url = await fetch(`https://api.openweathermap.org/data/2.5/weather?q=Moscow&appid=${API_KEY}&units=metric`);
    const data = await api_url.json();
    console.log(data);

}
```

Выведем компоненты в App.jsx

```
return (
  <div className="wrapper">
    <Info />
    <Form />
    <Weather />
  </div>
);
```

Укажем новые состояния для получения информации

```
const [temp, setTemp] = useState(undefined);
const [city, setCity] = useState('');
const [country, setCountry] = useState(undefined);
const [sunrise, setSunrise] = useState(undefined);
const [sunset, setSunset] = useState(undefined);
const [error, setError] = useState(undefined);
```

Внутри функции gettingWeather установим состояния из полученный данных с сервера

```
setTemp(data.main.temp);
setCity(data.name)
setCountry(data.sys.country);
setSunrise(changeSun(data.sys.sunrise));
setSunset(changeSun(data.sys.sunset));
setError(undefined);
```

Передадим функцию о получении данных в компонент Form. Вытащим оттуда название введенног7о города в родительском компоненте в функции gettingWeather. И установим событие на форму

```
<Form weatherMethod={gettingWeather} />
```

```
event.preventDefault();
const city = event.target.elements.city.value;
```

```
<form onSubmit={weatherMethod}>
    <input type="text" name='city' placeholder='город' />
    <button type='submit'>Получить погоду</button>
</form>
```

Передадим полученные в стейтах данные в компонент Weather. Примем данные в компоненте. Деструктурируем их и выведем в соответствующие поля

```
<Weather temp={temp} city={city} country={country} sunrise={sunrise} sunset={sunset} error={error} />
```

```
const Weather = ({ temp, city, country, sunrise, sunset, error }) => {
    return (
        <div>
            {temp &&
                <div>
                    <p>Местонахождение: Город - {city} Страна - {country}</p>
                    <p>Температура {temp}</p>
                    <p>Восход солнца {sunrise}</p>
                    <p>Закат солнца {sunset}</p>
                </div>
            }
            <p>{error}</p>
        </div>
    );
}
```

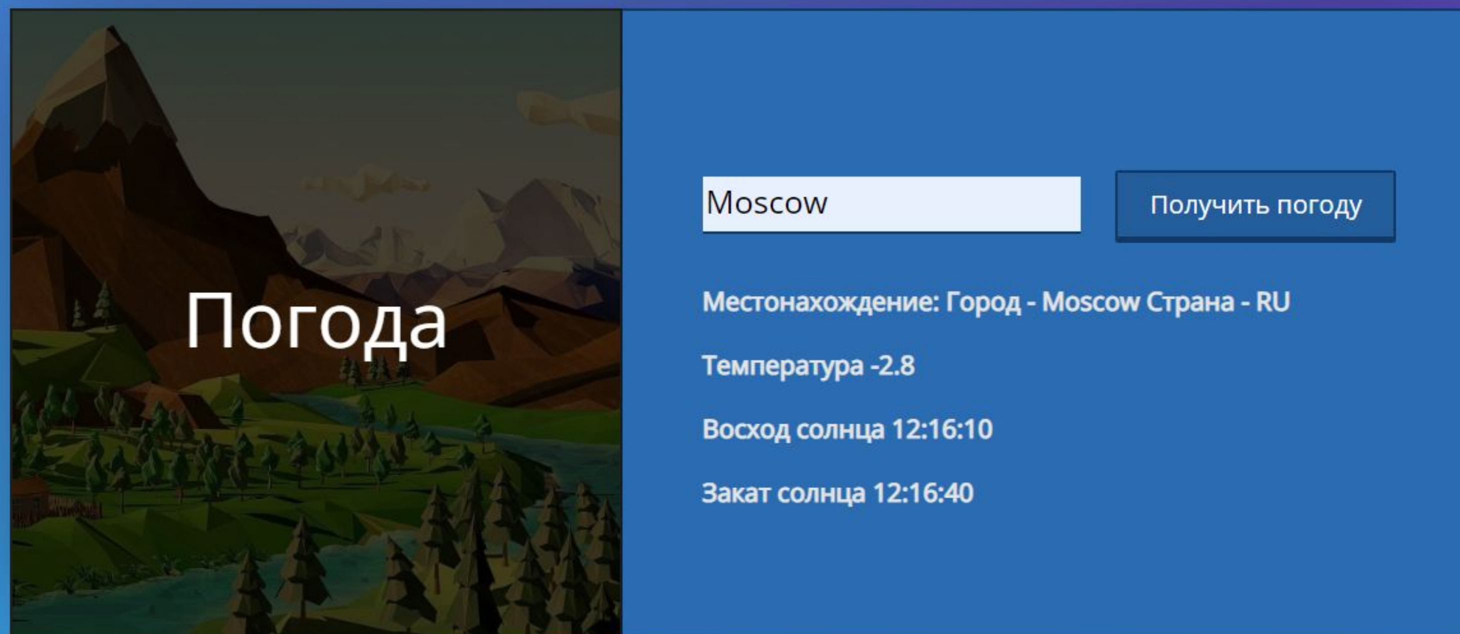
Добавим функцию для перевода данных о восходе в часы, минуты и секунды

```
function changeSun(sun) {
  let date = new Date();
  date.setTime(sun);
  let sun_date = date.getHours() + ':' + date.getMinutes() + ':' + date.getSeconds();
  return sun_date;
}
```

Добавим проверку на ошибку отправки пустого поля в функции gettingWeather

```
const gettingWeather = async (event) => {
  event.preventDefault();
  const city = event.target.elements.city.value;
  if (city) {
    const api_url = await fetch(`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${API_KEY}&units=metric`);
    const data = await api_url.json();
    console.log(data);
    setTemp(data.main.temp);
    setCity(data.name)
    setCountry(data.sys.country);
    setSunrise(changeSun(data.sys.sunrise));
    setSunset(changeSun(data.sys.sunset));
    setError(undefined);
  } else {
    setTemp(undefined);
    setCity(undefined)
    setCountry(undefined);
    setSunrise(undefined);
    setSunset(undefined);
    setError("Введите город");
  }
}
```

Стилизуем приложение



Задание

Сделать приложение для заказа коктейля с помощью API с сайта
www.thecocktaildb.com/api/json/v1/1/filter.php?c=Cocktail
Данные придут в виде


```
"strDrink":"155 Belmont",
"strDrinkThumb":"https:\/\/www.thecocktaildb.com\/images\/media\/drink\/yqvvqs1475667388.jpg"
"idDrink":"15346"
```


На главной странице сделать карточки товаров с названием и картинкой.
Добавить кнопку Заказать и функцию добавления в заказ. На странице с заказом добавить возможность удаления напитка из заказа. Сверху сделать меню для авторизации и регистрации (ОБЯЗАТЕЛЬНО с валидацией)