Using the Fetch API

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the protocol, such as requests and responses.
It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network. For making a request and fetching a resource, use the fetch() method.

```
fetch("http://example.com/movies.json") .
then((response) => response.json()) .
then((data) => console.log(data));
```

● Here we are fetching a JSON file across the network and printing it to the console. The simplest use of fetch() takes one argument — the path to the resource you want to fetch — and does not directly return the JSON response body but instead returns a promise that resolves with a Response object.

● The Response object, in turn, does not directly contain the actual JSON response body but is instead a representation of the entire HTTP response.

● So, to extract the JSON body content from the Response object, we use the json() method, which returns a second promise that resolves with the result of parsing the response body text as JSON.

How to make HTTP requests using Fetch API and Promises

```
const getCountryData = function (country) {

 const data1 = fetch(`https://restcountries.com/v3.1/name/${country}`).
then(response => response.json()).
then(data => console.log(data)).catch(error => console.log("api error"));

 }

 getCountryData("india");
```

Promise

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value. A Promise is in one of these states:

- pending: initial state, neither fulfilled nor rejected.
- fulfilled: meaning that the operation was completed successfully.
- rejected: meaning that the operation failed.

Object

The `Object` type represents one of JavaScript's data types. It is used to store various keyed collections and more complex entities.

> The `Object.freeze()` static method *freezes* an object. Freezing an object prevents extensions and makes existing properties non-writable and non-configurable. A frozen object can no longer be changed: new properties cannot be added, existing properties cannot be removed, their enumerability, configurability, writability, or value cannot be changed, and the object's prototype cannot be re-assigned. `freeze()` returns the same object that was passed in.

```
const obj = {
  prop: 42
};

Object.freeze(obj);

obj.prop = 33;
// Throws an error in strict mode

console.log(obj.prop);
// Expected output: 42
```

>The `hasOwnProperty()` method returns a boolean indicating whether the object has the specified property as its own property (as opposed to inheriting it).

```
const object1 = {};
object1.property1 = 42;
```

```
console.log(object1.hasOwnProperty('property1'));
// Expected output: true

console.log(object1.hasOwnProperty('toString'));
// Expected output: false

console.log(object1.hasOwnProperty('hasOwnProperty'));
// Expected output: false
```

>The `Object.seal()` static method *seals* an object. Sealing an object prevents extensions and makes existing properties non-configurable. A sealed object has a fixed set of properties: new properties cannot be added, existing properties cannot be removed, their enumerability and configurability cannot be changed, and its prototype cannot be re-assigned. Values of existing properties can still be changed as long as they are writable. `seal()` returns the same object that was passed in.

```
const object1 = {
  property1: 42
};

Object.seal(object1);
object1.property1 = 33;
console.log(object1.property1);
// Expected output: 33

delete object1.property1; // Cannot delete when sealed
console.log(object1.property1);
// Expected output: 33
```

> The `Object.keys()` static method returns an array of a given object's own enumerable string-keyed property names.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false
```

```
};

console.log(Object.keys(object1));
// Expected output: Array ["a", "b", "c"]
```