Front-end <Web /> Development

Lesson 12: API

University of Tehran

ACM
Summer of Code 2024

Lecturer:
Misagh Mohaghegh

A Short Review…

- React Hooks
- useState
- Lifting state up
- Updating object states
- Context
- Context Provider
- useEffect
- React Router

Time for…

FRONT-END

+

BACK-END

API

HTTP API

Application Programming Interface

API stands for Application Programming Interface.
It is a protocol or contract that allows one application to communicate with another.
It is a software interface or bridge offering service to other software.

An API specification is a document that describes how to use such an interface or connection.

HTTP

A common way to use APIs is through HTTP requests.
An API can be build on top of any communication protocol (such as HTTP or gRPC or WebSocket)

Our front-end needs to connect to a back-end to get the data it needs.
A web API which allows communication over the internet is needed for this.

HTTP (HyperText Transfer Protocol) is the method used for web APIs.
We have talked about HTTP in the first lesson.
Refer to the last slides of the first lesson for an overview on the HTTP protocol.

Endpoints

Endpoints are the access points or URLs that clients interact with an API.
Each endpoint represents individual functionalities exposed by the API.

Each HTTP endpoint is identified by a HTTP method and a URL path.
For example:
GET https://randomuser.me/api?gender=male

HTTP Method: GET
Base URL: https://randomuser.me
Resource Path: /api
Query Parameters: gender=male

Response Formats

After making a request to an endpoint, we expect a response.
The response data can be in many different formats.
The API can have defined its own format or used a standard data transfer format:

- XML
- CSV
- JSON
- TOML
- Protobuf

JSON is the most widely used API format.
This is both for sending and receiving data.
Such an API is called a REST API.

Requests

We can manually make some requests using tools such as cURL or Postman.

```
curl -X POST -H "Content-Type: application/json" -d '{"username": "Misagh"}' http://example.site
```

The -X flag sets the HTTP request method.

Each -H flag sets an additional HTTP request header.

The -d flag sets the request body (usually not needed for a GET request).

The -v flag prints the request and response headers before the response data.

JSON

JavaScript Object Notation

JSON or JavaScript Object Notation is a data transfer format.
It uses human-readable key-value text.

JSON is language-independent and is widely used for data interchange.
A lot of programming languages include built-in libraries to parse and generate JSON.

Format

JSON is a string representation of a JavaScript object:

```
'{"name":"Misagh","age":22,"car":null,"dead":false}'
```

We can rewrite the string using good indentation:

```
`{
    "name": "Misagh",
    "age": 22,
    "car": null,
    "dead": false
}`
```

Objects

We can see that a JSON string corresponds closely with a JS object:

```
const json = `{
    "name": "Misagh",
    "age": 22,
    "car": null,
    "dead": false
}`;
```

```
const me = {
    name: "Misagh",
    age: 22,
    car: null,
    dead: false
};
```

Data Types

Each JSON field can be one of the following types:
Each field is separated by a comma.

There are no comments in JSON by design.
Sometimes in "JSON with Comments" or jsonc there can be comments but that is not standard.

```
const json = `{
    "name": "Misagh", // string
    "age": 22,        // number
    "dead": false,    // boolean
    "favFoods": [     // array
        "Pizza",
        "Burger"
    ],
    "laptop": {       // object
        "brand": "Lenovo"
    },
    "car": null       // null
}`;
```

JSON Files

We can store JSON data in a file.json format.
In that case, the file should start with either { (for objects) or [(for arrays).

Syntax highlighting in a JSON file:

```json
{
    "name": "Misagh",
    "age": 22,
    "dead": false
}
```

JS Parsing

We can easily turn a JSON string into a JS object using the JSON.parse method:

```
const jsonString = '{"name":"test"}';
const myObj = JSON.parse(jsonString);
```

If the top-level of the JSON string is an array, a JS array is returned:

```
const jsonString = '["apple","orange"]';
const myArr = JSON.parse(jsonString);
```

JS Stringify

We can also convert a JS object to a JSON string:

```
const myObj = { name: "test" };
const jsonString = JSON.stringify(myObj);
```

We can also stringify a JS array:

```
const myArr = ["apple", "orange"];
const jsonString = JSON.stringify(myArr);
```

The result is a minified JSON string:

```
JSON.stringify(myObj); // {"name":"test"}
JSON.stringify(myObj, null, 2); //{
                                //  "name": "test"
                                //}
```

JSON vs XML

XML was used way before JSON. It is a lot less readable and also takes much more space:

```json
{
  "employees": [
    {
      "firstName": "A",
      "lastName": "B"
    },
    {
      "firstName": "C",
      "lastName": "D"
    },
  ]
}
```

```xml
<employees>
  <employee>
    <firstName>A</firstName>
    <lastName>B</lastName>
  </employee>
  <employee>
    <firstName>C</firstName>
    <lastName>D</lastName>
  </employee>
</employees>
```

AJAX

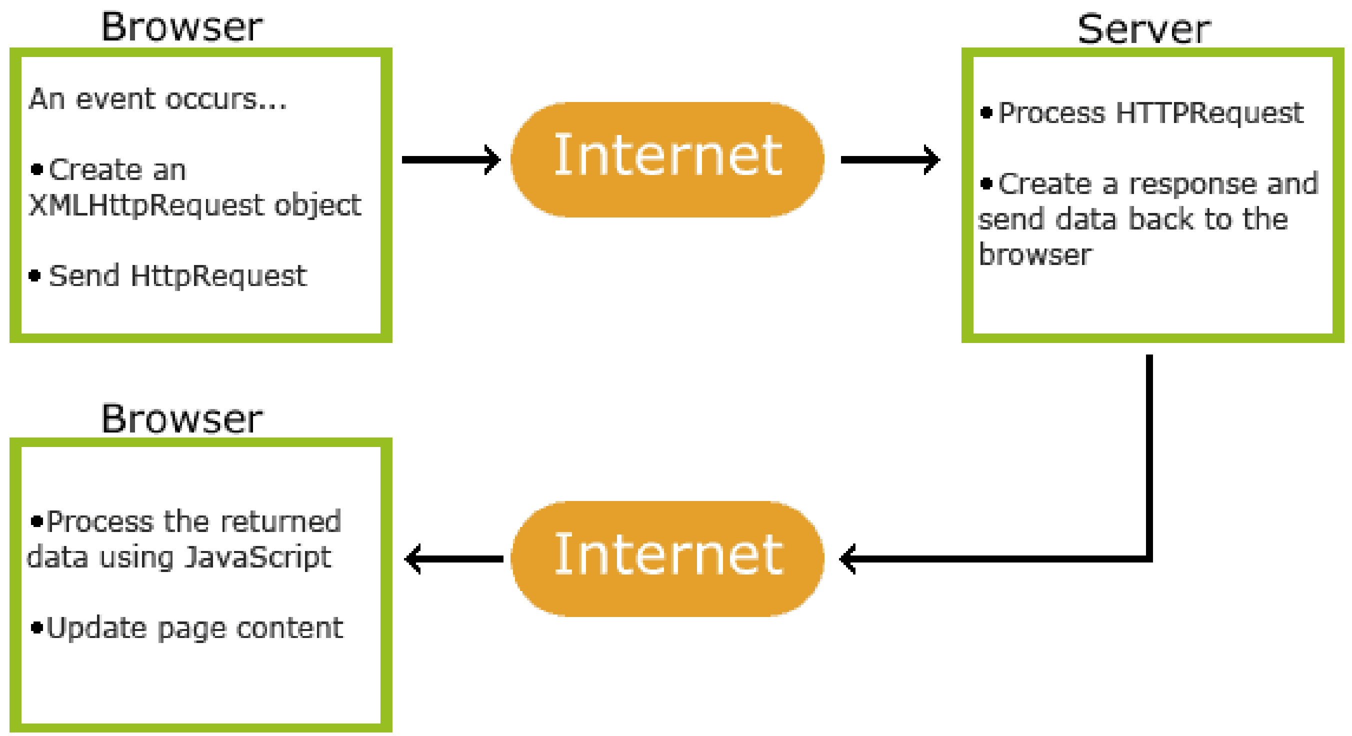
Asynchronous JavaScript And XML

AJAX is a JavaScript feature that allows sending requests and receiving responses without reloading the page.

The name is misleading because the data transfer format does not necessary have to be XML. It can also be JSON and other formats.

This is done through an XMLHttpRequest object.
The modern way of making requests is through the Fetch API and not this.

AJAX In Action


Browser
An event occurs...
•Create an XMLHttpRequest object
•Send HttpRequest
Internet
Server
•Process HTTPRequest
•Create a response and send data back to the browser
Browser
•Process the returned data using JavaScript
•Update page content
Internet

XMLHttpRequest GET

The general way of using the object:

```
const resElement = document.getElementById("test");

function loadDoc() {
    const xhttp = new XMLHttpRequest();
    xhttp.onload = function() {
        resElement.innerHTML = this.responseText;
    }
    xhttp.open("GET", "test.txt");
    xhttp.send();
}

button.addEventListener("click", loadDoc);
```

XMLHttpRequest POST

```
const resElement = document.getElementById("test");

function loadDoc() {
    const xhttp = new XMLHttpRequest();
    xhttp.setRequestHeader("Content-Type", "application/json");
    xhttp.onload = function() {
        const test = this.getResponseHeader("Last-Modified");
        resElement.innerHTML = this.responseText;
    }
    xhttp.open("POST", "/api/test");
    xhttp.send(JSON.stringify({ name: "me" }));
}

button.addEventListener("click", loadDoc);
```

Ready State

The onreadystatechange function runs 4 times for each ready state.

We should use the onload and onerror functions instead.

```
// XMLHttpRequest.UNSENT == 0
// XMLHttpRequest.OPENED == 1
// XMLHttpRequest.HEADERS_RECEIVED == 2
// XMLHttpRequest.LOADING == 3
// XMLHttpRequest.DONE == 4
function loadDoc() {
    const xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {...}
    }
    // ...
}
```

Fetch API

Fetch API

The Fetch API introduces the **window.fetch** method which is a clean interface for making HTTP requests.
It is a replacement for XMLHttpRequest.

The fetch() call returns a Promise which is fulfilled with a Response object.
Asynchronous JavaScript programming is required for this.

We can use the much easier async/await syntax from ECMAScript 2017 instead of working with Promises.

Fetch GET

```
const url = "https://example.org/products.json";

async function getData() {
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Response status: ${response.status}`);
    }
    const json = await response.json();
    console.log(json);
  }
  catch (error) {
    console.error(error.message);
  }
}
```

Fetch POST

```
async function postData(data) {
  try {
    const response = await fetch(url, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(data)
    });
    if (!response.ok) {
      throw new Error(`Response status: ${response.status}`);
    }
    const json = await response.json();
    console.log(json);
  }
  catch (error) {
    console.error(error.message);
  }
}
```

Fetching Data in React

Combining useEffect and useState Hooks to effectively fetch data is explained in code.

```
CODE!
```

Thank you for your attention.