Chapter 8 - Fetch API

Asst.Prof. Dr. Umaporn Supasitthimethee
ผศ.ดร.อุมาพร สุภสิทธิเมธี

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Synchronous/Asynchronous

Asynchronous vs. Synchronous

- **Synchronous** tasks are performed one at a time and only **when one is completed, the following is unblocked**. In other words, you need to wait for a task to finish to move to the next one.

```
console.log('hello ')
console.log('world, ')
console.log('bye’)
```

hello
World
Bye

- **Asynchronous** software design expands upon the concept by building code that allows a program to ask that a task be performed alongside the original task (or tasks), **without stopping to wait for the task to complete**. When the secondary task is completed, the original task is notified using an agreed-upon mechanism so that it knows the work is done, and that the result, if any, is available.

```
console.log('hello ')
setTimeout(() => console.log('world, '), 3000)
console.log('bye’)
```

hello
Bye
world

https://developer.mozilla.org/en-US/docs/Glossary/Asynchronous

Asynchronous Functions

In JavaScript, a callback function is a function that is passed into another function as an argument.

This function can then be invoked during the execution of that higher order function.

Since, in JavaScript, functions are objects, functions can be passed as arguments.

```
console.log('Hello')
//setTimeout is an
//asynchronous function
setTimeout(function () {
  console.log('JS')
}, 5000)

console.log('Bye bye')
```

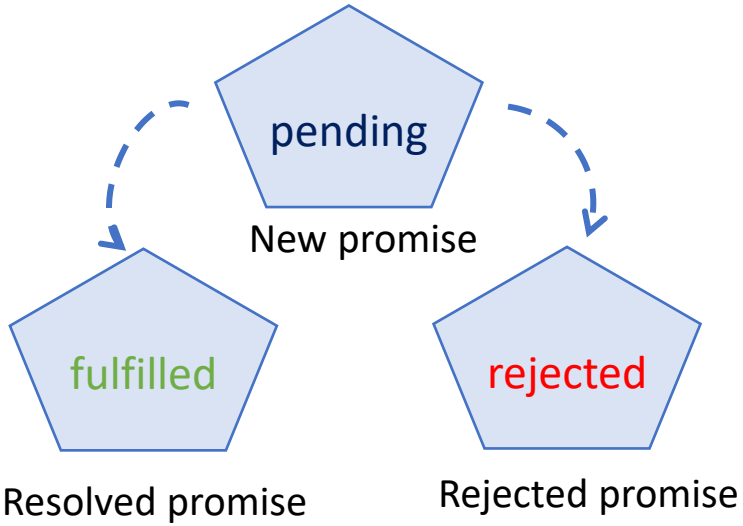
```
//Console

Hello
Bye bye

//until 5 seconds
JS
```

setTimeout() executes a particular block of code once after a specified time has elapsed.

`setTimeout()` is an asynchronous function, meaning that the timer function will not pause execution of other functions in the functions stack.

Promise

Promises



- Promises are ***asynchronous***.
- *"I **promise** to do this **whenever** that is true. If it isn't true, then I won't."*
- With Promises, we can defer execution of a code block until an async request is completed. This way, other operations can keep running without interruption.
- Promises have three states:
  - ***pending***: initial state, neither fulfilled nor rejected.
  - ***fulfilled***: meaning that the operation was completed successfully.
  - ***rejected***: meaning that the operation failed.

Creating a Promise

- The **Promise** object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- The **Promise** object is created using the `new` keyword and contains the promise;
- This is an executor function which has a `resolve` and a `reject` callback

```
const promise = new Promise(function(resolve, reject)
{
        // promise description
})
```

Using Promises `.then()`

`.then()` receives a function with an argument which is the resolve value of our promise.
`.catch()` returns the reject value of our promise

```
async function doSomething(done) {
 return new Promise((resolve, reject) => {
   console.log('waiting...')
   setTimeout(() => {
     done ? resolve('success') : reject('failure')
   }, 5000)
 })
}
```

```
async function thenDoSomething(result) {
  doSomething(result)
    .then((x) => {
      console.log(x + ', your activity is success')
    })
    .catch((error) => {
      console.log(`${error}, your activity fails`)
    })
}

thenDoSomething(false)
```

waiting...
failure, your activity fails

Using Promises `async/await`

- `async` function returns a promise -- if the function returns a value, the promise will be resolved with the value, but if the `async` function throws an error, the promise is rejected with that value.
- `await` keyword is only used in an `async` function to ensure that all promises returned in the `async` function are synchronized
- `await` eliminates the use of callbacks in `.then()` and `.catch()`

```
async function doSomething(done) {
 return new Promise((resolve, reject) => {
   console.log('waiting...')
   setTimeout(() => {
     done ? resolve('success') : reject('failure')
   }, 5000)
 })
}
```

```
async function waitingSomething(result) {
  try {
    const x = await doSomething(result)
    console.log(x + ', your activity is success')
  } catch (e) {
    console.log(`${error}, your activity fails`)
  }
}

waitingSomething(true)
```

```
waiting...
success, your activity is success
```

REST API

REST APIs

- Representational State Transfer (REST) APIs communicate via HTTP requests to perform standard functions like creating, reading, updating, and deleting records (also known as CRUD) within a resource.
- Request headers and parameters are also important in REST API calls because they include important information such as metadata, authorizations, uniform resource identifiers (URIs), cookies and more.
- The representation of a respond resource most often be JSON, HTML or XML. JSON is popular because it’s readable by both humans and machines.

JSON structure

- **JSON is a string** whose format very much resembles JavaScript object literal format.
- JSON **requires double quotes** to be used around strings and property names. **Single quotes are not valid** other than surrounding the entire JSON string.
- You can include the same basic data types inside JSON as you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals.
- JSON is purely a string with a specified data format — **it contains only properties, no methods**.
- We can also convert arrays to/from JSON.

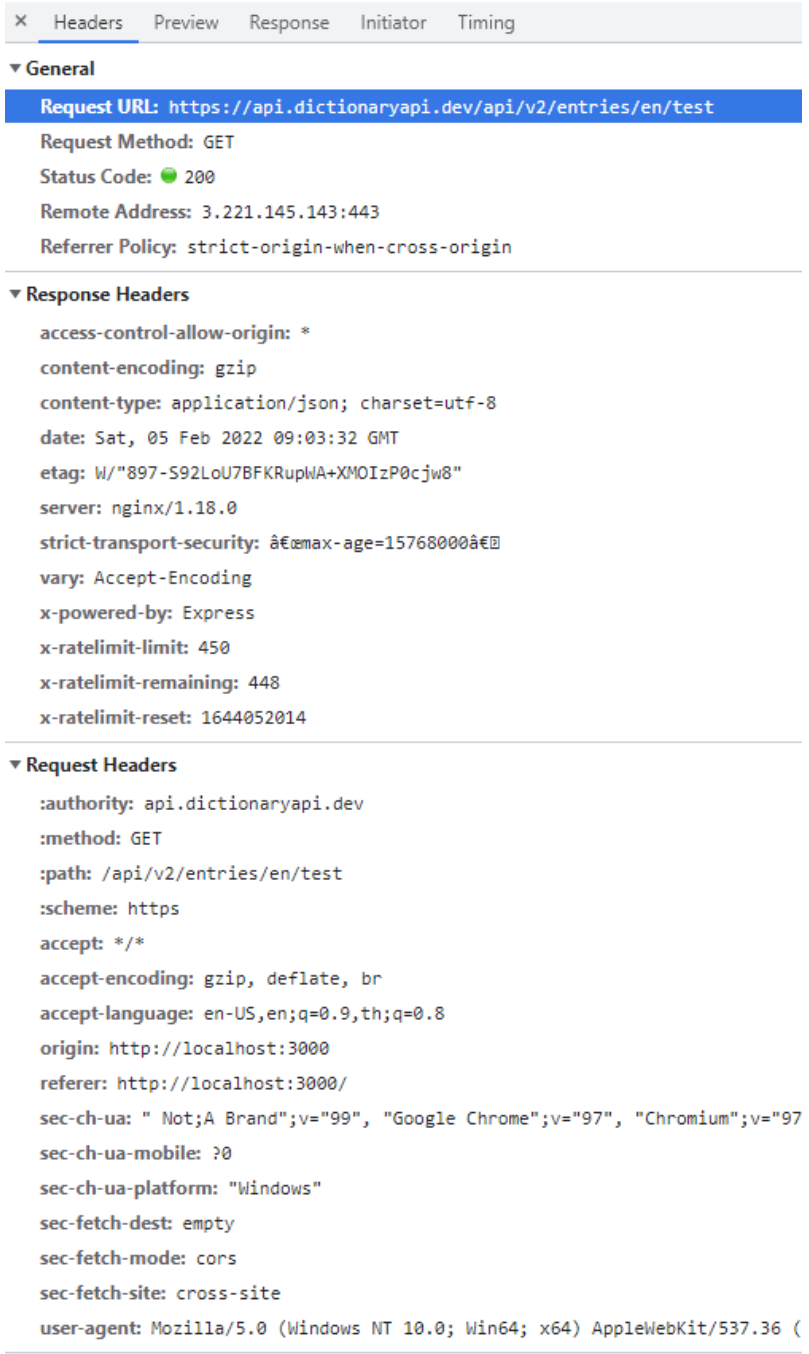
```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}
```

```
[
  {
    "name": "Molecule Man",
    "age": 29,
    "secretIdentity": "Dan Jukes",
    "powers": [
      "Radiation resistance",
      "Turning tiny",
      "Radiation blast"
    ]
  },
  {
    "name": "Madame Uppercut",
    "age": 39,
    "secretIdentity": "Jane Wilson",
    "powers": [
      "Million tonne punch",
      "Damage resistance",
      "Superhuman reflexes"
    ]
  }
]
```

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

Three elements in REST API

- **Request**—This is the data you send to the API across HTTP Protocol
- **Response**—Any data you get back from the server after a successful / failed request.
- **Headers**—Additional metadata passed to the API to help the server understand what type of request it is dealing with, for example “content-type”



Using HTTP Methods for RESTful Services

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

https://www.restapitutorial.com/lessons/httpmethods.html

fetch()

```
fetch(resource [, init])
```

resource
This defines the resource that you wish to fetch including a URL object — that provides the URL of the resource you want to fetch.

init Optional
An object containing any custom settings that you want to apply to the request. The possible options are:

- **method**:
  The request method, e.g., GET, POST.
- **headers:**
  Any headers you want to add to your request, contained within a Headers object or an object literal with String values.
- **body:**
  Any body that you want to add to your request. Note that a request using the GET method cannot have a body.

return value

- A Promise that resolves to a Response object.

Using Fetch API

- The `fetch()` function **returns a Promise** which is fulfilled with a Response object representing the server's response. You can then check the request status and extract the body of the response in various formats, including text and JSON, by calling the appropriate method on the response.
- The `fetch()` method takes one mandatory argument, the path to the resource you want to fetch **asynchronously across the network**.
- It returns a promise that resolves to the response to that request — as soon as the server responds with headers — even if the server response is an HTTP error status.

```
function fetch(url) {
  return new Promise(function (resolve, reject) {
  // promise description
  }
  )
}
```

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

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

json()

- The `json()` method of the Response interface takes a response stream and reads it to completion. It **returns a promise which resolves with the result of parsing the body text as JSON**.

- Note that despite the method being named `json()`, **the result is not JSON but is instead the result of taking JSON as input and parsing it to produce a JavaScript object**.

```
async function fetchData(endpoint) {
  try {
    const res = await fetch(endpoint);
    const data = await res.json();
    return data;
  } catch (error) {
    throw new Error(res.status);
  }
}
```

JSON-Server

README.md

JSON Server build passing npm package 0.16.3

Get a full fake REST API with **zero coding** in **less than 30 seconds** (seriously)

Created with <3 for front-end developers who need a quick back-end for prototyping and mocking.

- Egghead.io free video tutorial - Creating demo APIs with json-server
- JSONPlaceholder - Live running version
- **My JSON Server** - no installation required, use your own data

See also:

- husky - Git hooks made easy
- hotel - developer tool with local .localhost domain and https out of the box

https://www.npmjs.com/package/json-server/v/1.0.0-alpha.23

JSON-Server Steps

1. Install JSON Server

```
npm install json-server
```

หมายเหตุ ถ้าเพิ่ม –g (หมายถึง global ใช้ได้ทุก project)

2. Create a " ./data/db.json " file with some data

```
{
 "posts": [
   { "id": "1", "title": "a title", "views": 100 },
   { "id": "2", "title": "another title", "views": 200 }
  ]
}
```

3. Create script for running json-server backend

```
"scripts": {
      "backend": "json-server --watch ./data/db.json --port 5000"
}
```

Create script to run JSON-Server *(package.json)*

package.json

```
"scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "backend": "json-server --watch ./data/db.json --port 5000"
  },
  "dependencies": {
    "json-server": "^0.17.2",
    "vue": "^3.2.47"
  },
```

npm run backend

```
> json-server --watch ./data/db.json --port 5000


  \{^_^}/ hi!

  Loading ./data/db.json
  Done

  Resources
  http://localhost:5000/photos

  Home
  http://localhost:5000
```

JSON-Server Resources and EndPoints

```
{
      "posts": [
        { "id": "1", "title": "a title", "views": 100 },
        { "id": "2", "title": "another title", "views": 200 }
      ],
      "comments": [
        { "id": "1", "text": "a comment about post 1", "postId": "1" },
        { "id": "2", "text": "another comment about post 1", "postId": "1" }
       ]
}
```

Resources
http://localhost:5000/posts
http://localhost:5000/comments

JSON-Server Endpoints

http://localhost:5000/posts

```
[
    { "id": "1", "title": "a title", "views": 100 },
    { "id": "2", "title": "another title", "views": 200 }
]
```

http://localhost:5000/posts/1

```
{ "id": "1", "title": "a title", "views": 100 }
```

Fetch API
CRUD

Prepare your JSON

```
[
  {
    "id": "1",
    "category": "home",
    "description": "Buy groceries"
  },
  {
    "id": "2",
    "category": "work",
    "description": "Finish project report"
  },
  {
    "id": "3",
    "category": "home",
    "description": "Clean the house"
  },
```

https://www.npmjs.com/package/json-server/v/1.0.0-alpha.23

GET method

async-await

```
async function getTodos() {
  try {
    const res = await
    fetch('https://localhost:5000/todos')
    const data = await res.json()
    console.log(data)
  } catch (error) {
    console.error('Error:', error)
  }
}
```

promise.then()

```
fetch('https://localhost:5000/todos')
.then(res => response.json())
.then(data => console.log(data))
.catch((error) => console.error('Error:',
error))
```

Fetch contains HTTP response, not the actual JSON. This includes headers, status code, etc. To extract the JSON body content from the response, we use the json() method.

POST (Add) method

async-await

```
async function createTodo() {
  try {
    const res = await fetch('https://
localhost:5000/todos', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        category: 'Home',
        description: 'Buy milk',
      }),
    })
    const addedTodo = await res.json()
  } catch (error) {
    console.error('Error:', error)
  }
}
```

promise.then()

```
fetch('https://localhost:5000/todos', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      category: 'Home',
      description: 'Buy milk',
    }),
})
.then(res => res.json())
.then(addedTodo => //update your todo array here…)
.catch((error) => console.error('Error:', error))
```

The **JSON.stringify() method converts a JavaScript object or value to a JSON string.
console.log(JSON.stringify({ x: 5, y: 6 })); // expected output: "{"x":5,"y":6}

*The fetch first parameter should always be the URL, a second JSON object with options like method, headers, request body, and so on.

DELETE method

async-await

```
async function deleteTodo() {
  try {
    const res = await fetch('https://localhost:5000/todos/1', {
      method: 'DELETE'
    })
    if(res.status === 200){
        console.log('Deleted')
            }
  } catch (error) {
    console.error('Error:', error)
  }
}
```

promise.then()

```
fetch('https://localhost:5000/todos/1', {
    method: 'DELETE',
})
.then((res) => {
   if(res.status === 200)
     console.log('Deleted')
})
.catch((error) => console.error('Error:', error))
```

PUT (Update with Replace) method

replaces an existing entire item or creates new if it does not exist.

Async-Await

```
async function updateTodo() {
  try {
    const res = await fetch('https://localhost:5000/todos/1', {
      method: 'PUT',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        category: 'Work',
        description: 'Prepare meeting notes',
      }),
    })
    const editedTodo = await res.json()
} catch (error) {
    console.error('Error:', error)
  }
}
```

PUT (Update with Replace) method

promise.then()

```
fetch('https://localhost:5000/todos/1', {
    method: 'PUT',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      category: 'Work',
      description: 'Prepare meeting notes',
    }),
})
.then(res => {
       const editedTodo = res.json()
})
.catch((error) => console.error('Error:', error))
```

PATCH (Update with Modify) method

partially updates an existing item without modifying the other field.

Async-Await

```
async function patchTodo() {
  try {
    const res = await fetch('https://localhost:5000/todos/1', {
      method: 'PATCH',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        description: 'Prepare meeting notes and presentation',
      }),
    })
    const editedTodo = await res.json()
  } catch (error) {
    console.error('Error:', error)
  }
}
```

PATCH (Update with Modify) method

promise.then()

```
fetch('https://localhost:5000/todos/1', {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      description: 'Prepare meeting notes and presentation',
    }),
})
.then(res => {
        const editedTodo = res.json()
})
.catch((error) => console.error('Error:', error))
```

Practice
Quote Manager