Lecture 7

Node.js (part 2)

Today’s Contents

- Handling POST requests in Express.js
- Express.js Middleware
- Express.js File Uploading
- Cookies in Express.js
- CommonJS Modules
- `nodemon` package
- File I/O in Node.js
    - Reading/writing text files
    - Saving data to text files in JSON format
- Fetch API Review

Handling POST requests in Node.js

- Handling different POST requests with Node.js
- Testing web services with Postman

POST Parameters

With GET endpoints, we've used `req.params` and `req.query` to get endpoint parameters passed in the request URL.

But remember that POST requests carry data in the Request body!

```
app.post("/contact", (req, res) => {
  let name = req.params.name; // req.params isn't POST body
  /* ... */
});
```

Q: What is a disadvantage of sending parameters in a URL?

- Not secure, limit to the length of data we can send in a URL

So, how can we get POST body parameters sent by a client?

Handling Different POST Requests

POST requests can be sent with different data types:

- `application/x-www-form-urlencoded`
- `application/json`
- `multipart/form-data`

In Express, we have to use middleware to extract the POST parameters from `req.body`. For the first two, there are built-in middlewares, and we don't need middleware for `text/plain`.

With `fetch`, we use the `FormData` object to send POST body, which has the Content-Type of `multipart/form-data`.

There is no built-in middleware to access the `req.body` params for multipart content.

➢ We need another module!

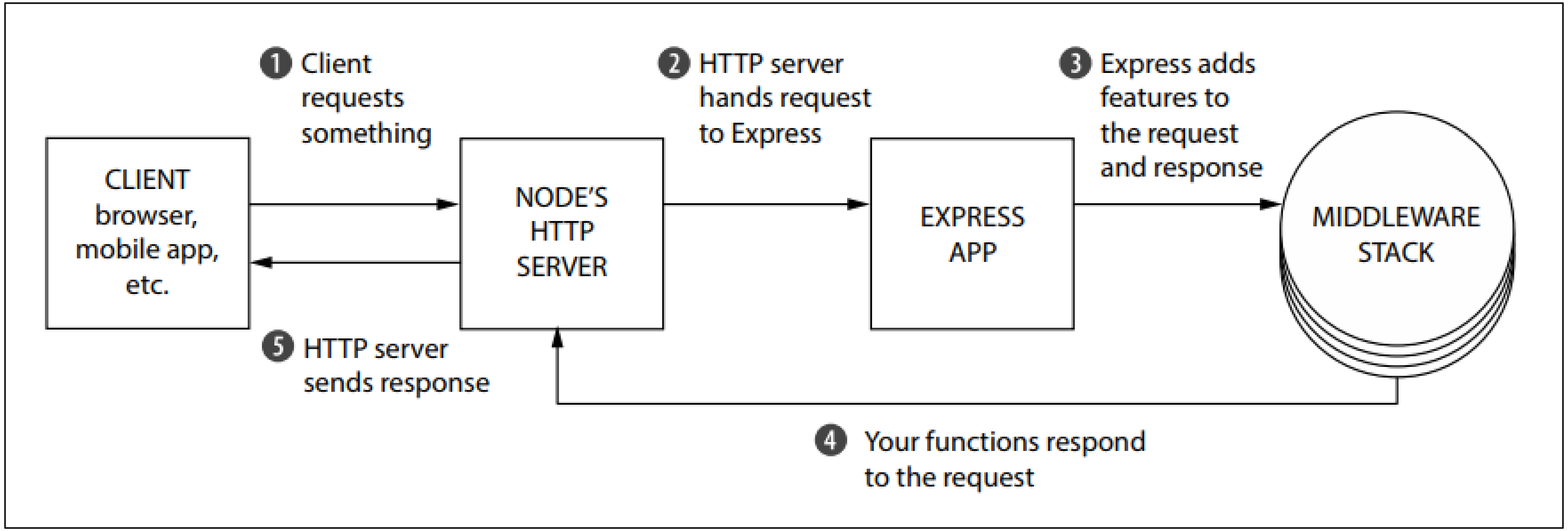
Postman demo

Use Postman to send a post request of each type to the following URL and see result:

https://hanustartup.org/wpr/PostTypes.php

Also inspect the request that Postman sends using Postman's **console**.

Middleware & Request/Response Pipeline



❶ Client requests something
❷ HTTP server hands request to Express
❸ Express adds features to the request and response
CLIENT browser, mobile app, etc.
NODE'S HTTP SERVER
EXPRESS APP
MIDDLEWARE STACK
❺ HTTP server sends response
❹ Your functions respond to the request

The `multer` Module

A module for extracting POST parameters sent through multipart POST requests like those sent with `FormData`.

Has a lot of functionality to support file uploading, but we will just use it to access the body of a POST request sent through `FormData`, which we can't get with just `req.body`.

To use, we'll need to set an option to ignore upload features with `multer().none()`

```
const multer = require("multer");

app.use(multer().none());
```

(*) Remember to run `npm install multer` in any project that uses it.

Supporting all POST requests

We often don't want to make assumptions about what method a client uses to POST data. You are required to support all three with the appropriate middleware on all assignments in this class.

```
// other required modules ...
const multer = require("multer");

// for application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true })); // built-in middleware
// for application/json
app.use(express.json()); // built-in middleware
// for multipart/form-data (required with FormData)
app.use(multer().none()); // requires the "multer" module
```

Postman demo #2

Use Node.js to handle three types of Post request (normal, form-data, json) at:

`http://localhost/post_target`

In Postman, create different Post requests to test the implementation.

Summary of Handling a POST Request

1. Use `app.post` instead of `app.get`
2. Use `req.body.paramname`
   - (instead of `req.params.paramname` or `req.query.paramname`)
3. Require the `multer` (non-core) module with the rest of your modules
4. Use the three middleware functions to support the three different types of POST requests from different possible clients (see previous slide)
5. You can use `fetch` (and `FormData`) to send POST requests in client-side code and use Postman to test your POST request handler. Remember you can't test POST requests in the URL!

Express.js middleware

Example: Your first middleware

```
const express = require('express');
const app = express();

app.use((req, res, next) => {
    req.requestTime = Date.now();
    next();
});

app.get('/', (req, res) => {
    res.send(`Hello World!<br>Requested at: ${req.requestTime}`);
});

app.listen(8000);
```

- The above example adds the `requestTime` property to the `req` object for every request to this Express web server.
- A middleware function is any function which receives 3 parameters: `req, res, next`
- `next` is a Function which should be invoked at the end of the middleware's body

Express.js middleware

- An Express application is essentially a series of middleware function calls.
- Middleware functions are functions that have access to:
  - The request object (`req`)
  - The response object (`res`)
  - The `next` middleware function in the request-response cycle
- Middleware functions can perform the following tasks:
  - Execute any code
  - Make changes to the `req` and the `res` objects
  - End the request-response cycle
  - Call the next middleware function in the stack
- If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

Types of middleware

An Express application can use the following types of middleware:

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

Third-party middleware

- Third-party middlewares are middleware functions that are available in other packages.
- One of the most noticeable third-party middlewares is `cookie-parser`:

```
const express = require('express')
const app = express()
const cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```

- Another one is `multer`.

Built-in middleware

- Express has the following built-in middleware functions:
  - `express.static` serves static assets (HTML, CSS, images...)
  - `express.json` parses incoming POST requests with `application/json` content type
  - `express.urlencoded` parses incoming POST requests with `x-www-form-urlencoded` content type

Application-level middleware

- Bind application-level middleware to an instance of the Express `app` object by using `app.use()`:

```
const express = require('express')
const app = express()

app.use((req, res, next) => {
    console.log('Time:', Date.now());
    next();
});
```

Error-handling middleware

- Error-handling middleware functions always takes ***four*** arguments.
- The middleware below handles any error that may be generated by any endpoint in an Express application.

```
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Something broke!');
});
```

Cookies in Express.js

Cookies

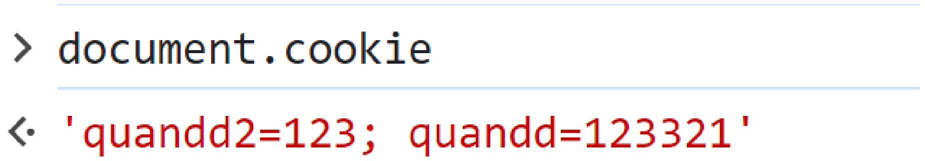
- Cookies are data, stored in small text files on your browser's temporary storage.
  - Temporary storage is the place where your browser caches html, css, js, images, etc.
  - Temporary storage is meant to speed-up page loads
- Cookies were originally invented to remember user information (to be used in future visits). Today, cookies are used for authenticating users, storing user preferences and tracking user's activities.
- Cookies are attached to every HTTP request from the client.
  - The more cookies, the bigger the request size → takes more time to send a request
  - In every request, the server-side application can read cookies from client
- Server-side application can send cookies to client.

Using Cookies on client-side

- A cookie is a name-value pair. E.g.

  `username=quandd`

- Client-side JS supports basic capabilities with cookies.
- Accessing `document.cookie` returns a string which contains all cookies.
  - Cookies are separated by semicolons (`;`)



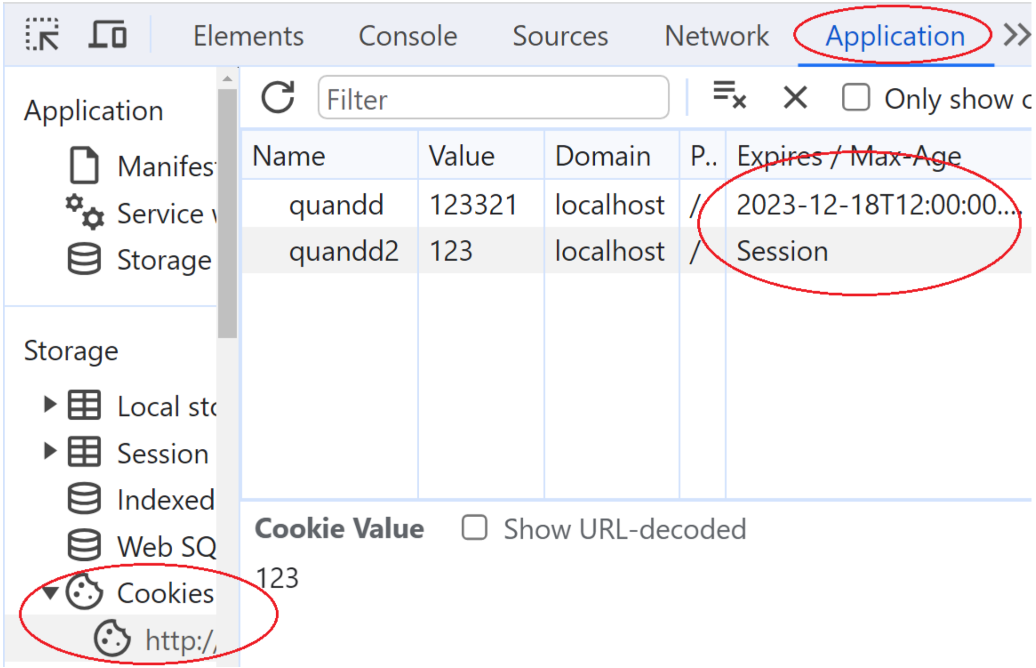
```
> document.cookie
<· 'quandd2=123; quandd=123321'
```

Using Cookies on client-side

- To create a cookie, assign it to document.cookie

```
document.cookie = "quandd2=123"
```

- By default, cookie has the type of `Session` and will be deleted when the browser is closed.



Name	Value	Domain	P..	Expires / Max-Age
quandd	123321	localhost	/	2023-12-18T12:00:00...
quandd2	123	localhost	/	Session

Cookie Value ☐ Show URL-decoded

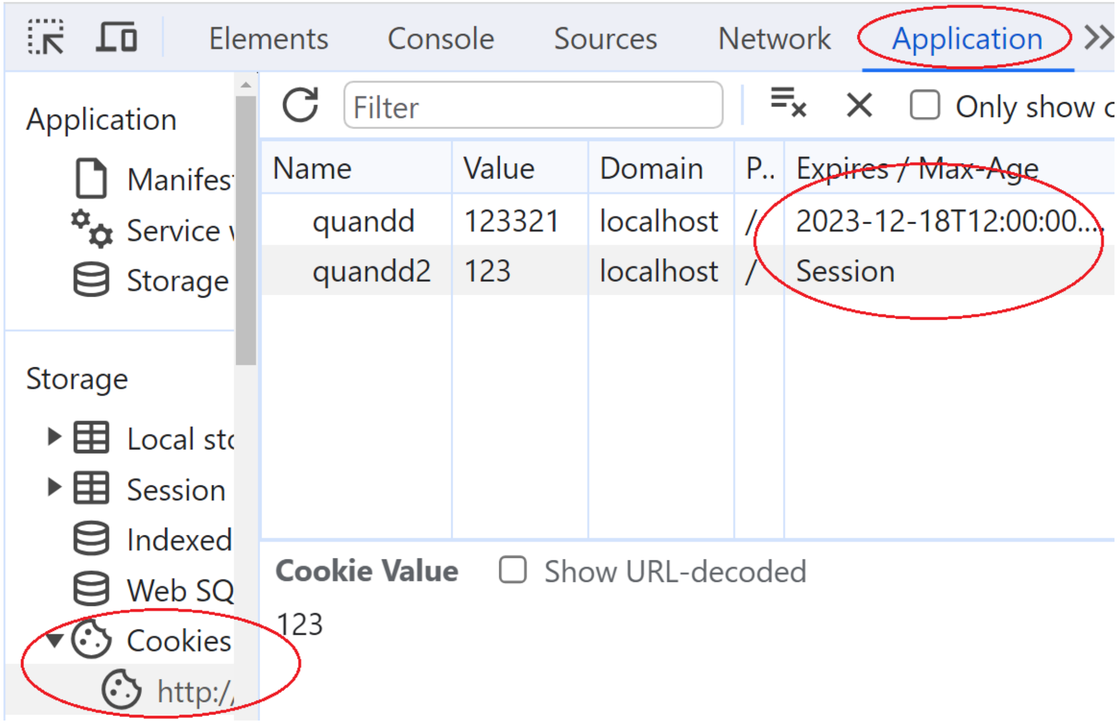
123

Using Cookies on client-side

- You can set the expire time and path when creating a cookie

```
document.cookie = "quandd=123321; expires=18 Dec 2023
12:00:00 UTC; path=/";
```

- Setting a cookie again using the same name and a different value will overwrite the existing one.
- Cookies can be viewed in the *Application* tab of the Developer Tools.
- Expired cookies will simply disappear



Name	Value	Domain	P..	Expires / Max-Age
quandd	123321	localhost	/	2023-12-18T12:00:00...
quandd2	123	localhost	/	Session

Cookie Value Show URL-decoded

123

Reading cookies from client on server-side

- We have to use the cookie-parser module. First, install it:

```
npm install cookie-parser --save
```

- Then, import and use it with the Express.js server:

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());
```

- You can read cookies from `req.cookies`. For example:

```
app.get('/', (req, res) => {
    res.send('Welcome ' + req.cookies.username);
});
```

Sending cookies to client from server-side

- You can set cookies using res.cookie. For example:

```
app.get('/', (req, res) => {
    res.cookie('cookie_name', 'Cookie value');
    res.send('Cookie has been set!');
});
```

- You can set a cookie with parameters such as `maxAge` or `expires`:

```
res.cookie('quandd', '123123', {
    maxAge: 5000,
    // expires works the same as the maxAge
    expires: new Date('01 12 2023')
});
res.send('Cookie has been set!');
```

The CommonJS module

- In Node, each file is treated as a separate module. Consider the following code:

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

- On the first line, the module `circle.js` that is in the same directory is imported (loaded). Here are the contents of `circle.js`:

```
const { PI } = Math;
exports.area = (r) => PI * r ** 2;
exports.circumference = (r) => 2 * PI * r;
```

- The module `circle.js` has exported the functions `area()` and `circumference()`.
- Functions, objects, variables are added to the root of a module by adding properties to the special `exports` object.

Loading a module

- We use the require function to load a CommonJS module:

```
const circle = require('./circle.js');
```

- A required module prefixed with `'./'` is relative to the file calling `require()`.

- If the exact filename is not found, Node.js will attempt to load the required file with the added extensions: `.js`, `.json`, and finally `.node`. Therefore, the above module can also be imported like the following:

```
const circle = require('./circle');
```

- If the module identifier passed to `require()` is not a core module, and does not begin with `'/'`, `'../'`, or `'./'`, Node.js will attempt to load the module from the `./node_modules` directory.

The module wrapper

- Before a module's code is executed, Node.js will wrap it with a function wrapper that looks like the following:

```
(function (exports, require, module, __filename, __dirname) {
    // Module code actually lives in here
});
```

- Top-level variables have module scope rather than global scope.
- It provides some global-looking variables that are actually specific to the module, such as:
  - The `module` and `exports` objects that you can use to export values from the module.
  - The convenience variables `__filename` and `__dirname`, containing the module's absolute filename and directory path.

See the difference between `exports` and `module.exports` here

nodemon package

`nodemon` is a tool that helps develop Node.js based applications by automatically restarting the node application when file changes in the directory are detected.

Installation

- Either through cloning with git or by using **`npm`** (the recommended way):

```
npm install -g nodemon
```

```
  # or using yarn: yarn global add nodemon
```

  And nodemon will be installed globally to your system path.

- You can also install nodemon as a development dependency:

```
npm install --save-dev nodemon
```

```
  # or using yarn: yarn add nodemon -D
```

  With a local installation, nodemon will not be available in your system path or you can't use it directly from the command line. Instead, the local installation of nodemon can be run by calling it from within an npm script (such as `npm start`) or using `npx nodemon`.

nodemon package

Usage

- nodemon wraps your application, so you can pass all the arguments you would normally pass to your app:

```
nodemon [your node app]
```

- For CLI options, use the -h (or --help) argument:

```
nodemon -h
```

- Using nodemon is simple, if my application accepted a host and port as the arguments, I would start it as so:

```
nodemon ./server.js localhost 8080
```

File I/O in Node.js

Unlike the browser, we have access to the file system when running Node.js.

We can read all kinds of files, as well as write new files.

The `fs` Core Module

We saw `express` as our first module in Node.js to create an API

... And `multer` for parsing POST request bodies

Another useful module: `fs` (file system)

- This is a “core" module, meaning we don't have to `npm install` anything.

There are many functions in the `fs` module (with excellent documentation)

Most functions rely on error-first callbacks, but we're going to use the Promise versions!

Reading Files and Writing Files

`fs.readFile(fileName, encodingType)`

- `fileName:` (string) file name
- `encodingType:` file encoding (usually `"utf8"`)
- Returns: Promise with a value holding the file's contents

`fs.writeFile(fileName, contents)`

- `fileName:` (string) file name
- `contents:` (string) contents to write to file
- Returns: Promise without any resolved value

Reading a file with error-first callbacks

To read and write files, we first need the FileSystem package
`const fs = require('fs');`

```
fs.readFile('example.txt', 'utf8', (err, contents) => {
  if (err) {
    handleError(err);
  } else {
    printContents(contents);
  }
});
```

As soon as `fs.readFile` finishes reading the file (or there's an error), it will invoke the callback function that you provide.

Writing to files

Writing to files is the same, but with slightly different arguments:

```
fs.writeFile('example.txt', 'Sample Text', (err) => {
  if (err) {
    handleError(err);
  } else {
    console.log('File successfully written to!');
  }
});
```

Reading files with the FileSystem package

We can use the *Promisified* version of the fs module so that its functions return Promises instead of taking a callback.

```
const fs = require('fs').promises;
```

Using .then/.catch

```
let contents = fs.readFile('example.txt', 'utf8');
contents
   .then(printContents)
   .then(() => {console.log('Done reading file!');})
   .catch(handleError);
```

Using async/await

```
let contents = await fs.readFile('example.txt', 'utf8');
printContents(contents);
console.log('Done reading file!');
```

Basic File I/O

```
const fs = require('fs').promises;
// ...

async function readFile() {
  let contents = await fs.readFile('posts.txt', 'utf8');
  return contents.split('\n');
}

async function writeFile(post) {
  await fs.writeFile('posts.txt', post + '\n');
}

// ...
```

Error handling with `async/await`

- For error-handling with `async/await`, you must use `try/catch` instead of `.then/.catch`
- The `catch` statement will catch any errors that occur in the `then` block (whether it’s in a Promise or a syntax error in the function), similar to the `.catch` in a `fetch` promise chain
- Remember that if you are using these in a web service, you should handle client-specific (400) errors differently than server-specific (500) errors (such as those caught by a `fs` function).

Example of Error Handling

```
// Example Error Handling
try {
  let contents = await fs.readFile('posts.txt', 'utf8');
  // Do something with contents.
} catch (err) {
  // in reality, your error handling should be better than this 😉
  console.error(err);
}
```

Parsing as JSON

You can read any file, including JSON. To parse JSON file contents and use as a JS object, use `JSON.parse()`.

```
// Example JSON Parsing
let contents = await fs.readFile("something.json", "utf8");
let obj = JSON.parse(contents);
```

Writing and Reading JSON

- JSON files are text representations of JSON objects. When you read from them, you will get a big string. When you write to them, you need to write a string. Remember to use `JSON.parse(jsonString)` to turn a JSON string into an object, and `JSON.stringify(jsonObj)` to turn a JSON object into a string.
- Being able to store a bunch of organized data in a file is convenient!

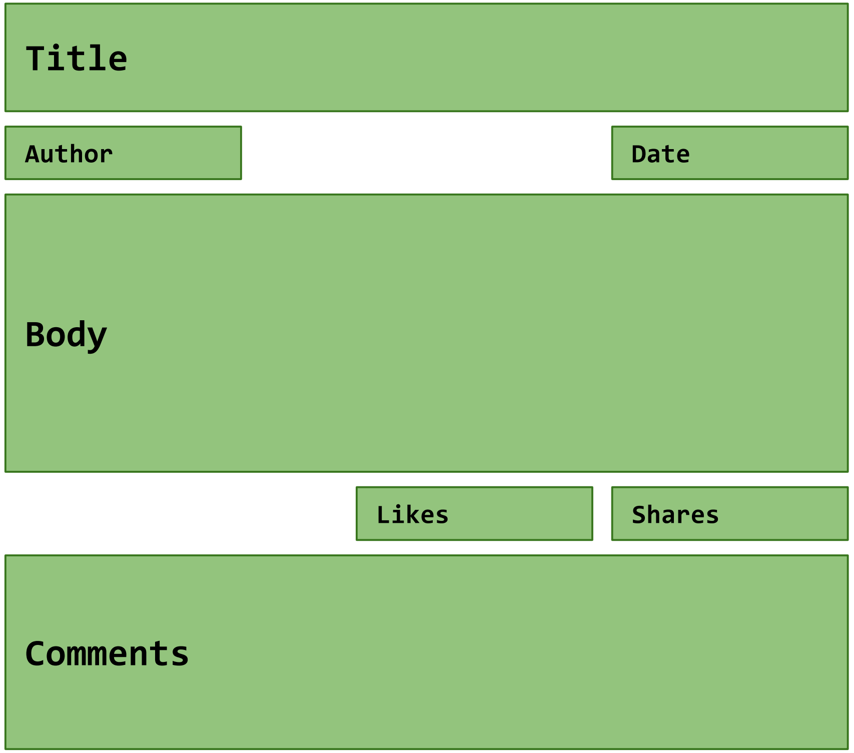
```
let data = await fs.readFile('example.json', 'utf8');
data = JSON.parse(data);
data['exploration-session'] = 'React';
await fs.writeFile('example.json', JSON.stringify(data));
```

Is File-Processing the Best Way?

- We have seen how to store JSON data in a file
- On the server, we have a lot of functionality to access the file system
- But, processing files can get a bit tedious and it's easy to accidentally overwrite data.

What if we want to update the file?

Consider a Blog, which has "posts", where each post is structured like what's on the right →



Could end up with JSON like:

```
{
  posts: [
    {
      title: "This is my first blog post!",
      body: "I don't have much to say, but here it is.",
      time: "2021-05-17T23:37:22Z"
    },
    {
      title: "This is my second blog post!",
      body: "I still don't have a lot to say because I'm limited to 100 characters per line.",
      time: "2021-05-18T23:37:22Z"
    }
  ]
}
```

What about accessing the posts?

What if we want to filter our data?

By ....

- All posts, most recent post, first, last, last week's, last months, that one post on May 17th, all posts containing a word in a title, posts with a certain word in the body, within a time range, having so many comments, have comments with certain words, ...

Question: How might you write the code to access posts in different ways?

Like this, maybe?

```
app.get('/blogs', async function(req, res) {
    let title = req.query.title;
    let post = req.query.post;
    let date = req.query.date;
    let sort = req.query.sort;

    let posts = await getPosts();
    let filteredPosts = [];
    for (let i = 0; i < posts.length; i++) {
        if (posts[i].title.includes(title)
            && posts[i].body.includes(post)
            && (new Date(posts[i].time)) >= (new Date(date))) {

            filteredPosts.push(posts[i]);
        }
    }

    if (sort === 'asc') {
        // Sort filteredPosts w/ oldest first
    } else if (sort === 'desc') {
        // Sort filteredPosts w/ newest first
    }

    res.json(filteredPosts);
});
```

Discussion

What are some limitations of filtering out the data in JSON?

- Efficiency: Expensive looping for large datasets, various if/else checks, and extra loops for sorting.
- Long, complex code just to get a subset of the desired data - this code also will need to be manually edited to a good degree when you want a different subset/sorting of data. This also adds more room for programming errors.
- Some filtering code written on the client-side (JS) - could take a relatively long time to finish when the page is also working on other tasks (e.g. user events, other AJAX requests).

Express.js File Uploading

How to send `multipart/form-data` POST Request?

- Files can be attached to a multipart/form-data POST request.
- This form is often used to upload files on the web.
- Method 1: using an HTML form

```
<form action="/profile" method="post" enctype="multipart/form-data">
  <label>Fullname <input type="text" name="fullname" /></label>
  <label>Username <input type="text" name="uname" /></label>
  <label>Avatar <input type="file" name="avatar" /></label>
  <button>Submit Form</button>
</form>
```

- Don't forget the `enctype="multipart/form-data"` in your form.

How to send `multipart/form-data` POST Request?

- Method 2: using the `fetch` function in client-side JavaScript

```
let uname = document.querySelector("#username").value;
let fileInput = document.querySelector("#avatar");
let data = new FormData();
data.append('user', uname);
data.append('avatar', fileInput.files[0]);
let response = await fetch(API_URL, { method: "POST", body: data });
```

- This method still requires an HTML form, but will submit the form using JavaScript instead of the default behavior of the form's submit button.
    - **Note:** you'll need to call the `preventDefault()` function to prevent the default behavior.

How to handle file upload on server?

Example of using `multer` middleware on the form-handling endpoint to receive a single uploaded file:

```
const multer = require('multer');

app.post(
    '/upload',
    multer({ dest: 'tmp/' }).single('avatar'),
    async (req, res) => {
        let ava = req.file;
        await fs.rename(ava.path, 'public/images/' + ava.originalname);
        res.send("Upload complete!");
    }
);
```

The uploaded file object

```javascript
const multer = require('multer');
const upload = multer({ dest: 'tmp/' });

app.post('/upload', upload.single('avatar'), async (req, res) => {
    res.json(req.file); // let's see what's inside this object
});
```

Browser output (formatted for readability):

```
{
  fieldname: 'avatar',
  originalname: '640px-UML_logo.svg.png',
  encoding: '7bit',
  mimetype: 'image/png',
  destination: 'tmp/',
  filename: 'a74b35aeddd727f9c2f480948fcc9256',
  path: 'tmp\\a74b35aeddd727f9c2f480948fcc9256',
  size: 28733
}
```

How to handle file upload on server?

```
app.post(
    '/upload',
    multer({ dest: 'tmp/' }).single('avatar'),
    async (req, res) => {
        let ava = req.file;
        await fs.rename(ava.path, 'public/images/' +
ava.originalname);
        res.send("Upload complete!");
    }
);
```

- The `dest` property specifies the directory where the file will be uploaded.
- The uploaded file on the server has a different name (random, without extension)
- That's why you often need to move the file to somewhere else.

Uploading multiple files - HTML Form

```html
<form action="/upload" method="post" enctype="multipart/form-data">
    <label>
        Photos:
        <input type="file" name="photos" multiple />
    </label>
    <input type="submit" value="Upload photos" />
</form>
```

- The `multiple` attribute lets user select multiple files for this `file` input.

Uploading multiple files - Server

```
const upload = multer({ dest: 'tmp/' });

app.post(
    '/photos/upload',
    upload.array('photos', 12),
    function (req, res) {
        // req.files is an array of `photos` files
        // req.body will contain other non-file fields, if any
    }
);
```

- Use `upload.array` instead of `upload.single` for handling a multiple file input.
- It's also possible to handle a form with multiple `file` input fields, each `file` input has the `multiple` attribute. Visit multer documentation to learn more use cases.

JS Fetch API Review

Self-Reading Section

Review the Fetch API

- **What is the Fetch API?**

✓ The Fetch API provides a modern, flexible way to make network requests in client-side JavaScript.
✓ Replaces the older XMLHttpRequest API.

- **Key Features:**

✓ Returns a Promise.
✓ Supports various HTTP methods like GET, POST, PUT, DELETE.
✓ Easy to use with async/await.

Basic Syntax

- **Fetching Data with GET**

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

- `fetch(url)`: Initiates the request.
- `response.json()`: Converts the response to JSON.
- Handles errors with `.catch()`.

Fetch API

- Using **asyn/await**

```
async function fetchData() {
  try {
    const response = await 
fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
```

Fetch API

Making a POST Request

```
fetch('https://api.example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ key: 'value' })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

Method: method: 'POST' to specify the request type.

Headers: Specify content type using headers.

Body: Include data to be sent, converted to a JSON string.

Fetch API

Handling Responses

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Response Properties:

`response.ok:` Checks if the request was successful (status in the range 200-299).

`response.status:` HTTP status code of the response.

`response.headers:` Access response headers.

Fetch API

Error Handling

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Fetch Error:', error);
  }
}
```

Common Errors:

- Network failures.
- Invalid URLs.
- Issues with CORS (Cross-Origin Resource Sharing).