Outline

1. Built-in variables and functions
2. Built-in modules: fs, os, path, url, querystring, net, http
3. Events and Streams in node.js
4. Sending a web request using the http module
5. Building a basic web server

Built-in variables and functions

Built-in variables and functions

- Node.js provides a set of built-in variables and functions that are essential for developing applications
- **Global variables**: global, process, __dirname, __filename, exports, console, module
- **Global functions**: setTimeout(), clearTimeout(), setInterval(), clearInterval(), require()

The global object

- The global object is the root object of the JavaScript execution environment in Node.js.
- It contains all of the built-in objects and functions that are available in Node.js.
- The global object can be accessed using the global keyword.

```
function sampleFunction() {
    let a = 10
    global.b = 20
    console.log(a + b) // 30
}

sampleFunction()
console.log(a) // ReferenceError: a is not defined
console.log(b) // 20
```


The global object

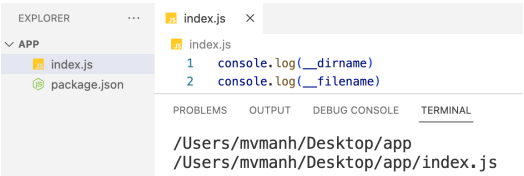
```
function sampleFunction() {
    let a = 10
    global.b = 20
    console.log(a + b) // 30
}

sampleFunction()
console.log(a) // ReferenceError: a is not defined
console.log(b) // 20

console.log(global.b) // 20 (same as above)
global.console.log(b) // 'console' is also a member of the global object
```


__dirname and __filename

- __dirname: It's a string that represents the directory name of the current module.
- __filename: It's a string that represents the file name of the current module.



The module object

- Modules are a way of organizing code in Node.js, it is usually presented by a .js file.
- The module object is available in every JavaScript file in a Node.js application.
- It provides information about the current module, such as its name, path, and exports.
- Here are some of the properties and methods of the module object:
  - module.exports: An object that contains the exports of the module.
  - module.filename: The path of the file that contains the module.
  - module.name: The name of the module.
  - module.require: A function that loads other modules.
  - module.exports: A function that exports the module's exports.

The module object

EXPLORER
APP
index.js
package.json

index.js

```
console.log(module)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
{
  id: '.',
  path: '/Users/mvmanh/Desktop/app',
  exports: {},
  filename: '/Users/mvmanh/Desktop/app/index.js',
  loaded: false,
  children: [],
  paths: [
    '/Users/mvmanh/Desktop/app/node_modules',
    '/Users/mvmanh/Desktop/node_modules',
    '/Users/mvmanh/node_modules',
    '/Users/node_modules',
    '/node_modules'
  ]
}
```

The console object

- The console object in Node.js is a global object that provides a simple debugging console similar to the JavaScript console mechanism provided by web browsers.
- **Logging Messages**:
  - `console.log()`: Standard log message.
  - `console.error()`: Error message.
  - `console.warn()`: Warning message.
  - `console.info()`: Info message.

The console object

- If the default Terminal does not apply distinct colors to different types of console messages. You can manually add color and styling to your messages to make them stand out.

```
const RESET = "\x1b[0m";
const RED = "\x1b[31m";
const YELLOW = "\x1b[33m";
const BLUE = "\x1b[34m";
const GREEN = "\x1b[32m";

console.log(`${GREEN}This is a general log message.${RESET}`);
console.error(`${RED}An error occurred: File not found.${RESET}`);
console.warn(`${YELLOW}Warning: Data may be outdated.${RESET}`);
console.info(`${BLUE}Info: Application started successfully.${RESET}`);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
This is a general log message.
An error occurred: File not found.
Warning: Data may be outdated.
Info: Application started successfully.
```

The console object

- **Formatted Output:**
  - The console object allows you to format the output using placeholders. You can use %s for strings, %d for numbers, %j for JSON, and others.

```
const name = "Alice";
const age = 30;
console.log("Name: %s, Age: %d", name, age);

// Name: Alice, Age: 30
```

The console object

- **Time Measurements:**
  - You can use `console.time()` and `console.timeEnd()` to measure the time taken by a specific operation.

index.js > ...

```
console.time("operation");
let s = 0
for (let i = 1; i < 10 ** 9; i++)
{
  s += 1
}
console.timeEnd("operation");
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
mvmanh@Mais-MacBook-Pro app % node index.js
operation: 630.599ms
```


The setTimeout functions

- `setTimeout()` is a function that schedules a function to be called after a specified number of milliseconds.
- `clearTimeout()` is a function that cancels a timer that was scheduled with `setTimeout()`.

```
setTimeout(() => console.log('Time up!'), 3000) // run console.log after 3 seconds
console.log('Scheduled')

// Scheduled
   .
   .
   .
// Time up!
```


The setInterval functions

- `setInterval()` is a function that schedules a function to be called repeatedly after a specified number of milliseconds.
- `clearInterval()` is a function that cancels a timer that was scheduled with `setInterval()`.

```
let counter = 0;
const id = setInterval(() => {
   counter++;
   console.log(counter)
   if (counter === 5) {
      clearInterval(id);
   }
}, 1000);
```

1
2
3
4
5

Modules in Node.js

Module in Node.js

- In Node.js, modules are an essential part of structuring your code. They allow you to split your code into smaller, more manageable pieces, making it easier to maintain and reuse.
- **Core Modules**: These are built-in modules provided by Node.js itself. Examples include fs for file system operations and http for creating web servers.
- **User-Defined Modules**: These are modules created by developers like you to encapsulate specific functionality or code for reusability.

Node.js built-in modules

1. **fs** : File system module for reading, writing, and manipulating files.
2. **os** : Operating system module for retrieving information about the operating system and its environment.
3. **path** : Path module for manipulating file paths.
4. **querystring** : Querystring module for parsing and manipulating query strings.
5. **net** : Networking module for creating and managing network sockets.
6. **http/https** : HTTP(s) module for creating and managing HTTP servers and clients.
7. **url** : Module for parsing and manipulating URLs.
8. **stream** : Abstract interface for working with streaming data.
9. **event** : Module for managing events.

File System

File System

- Node.js **fs** is a module that provides access to the file system on the local machine.
- It can be used to read, write, create, delete, and list files and directories.
- Node.js File System is asynchronous, which means that file operations do not block the main thread.
- This makes it ideal for high-performance applications.

The fs module

- To use the `fs` module, you first need to `require` it. You can do this by using the `require()` function

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

- Once you have required the `fs` module, you can use its functions to interact with the file system

```
const contents = fs.readFileSync('myfile.txt', 'utf8');
```


The fs module

- The **fs** module provides a number of common functions for interacting with the file system:
  - `readFile()`: Reads the contents of a file.
  - `writeFile()`: Writes the contents of a file.
  - `createFile()`: Creates a new file.
  - `deleteFile()`: Deletes a file.
  - `readdir()`: Lists the contents of a directory.
  - `mkdir()`: Creates a new directory.
  - `rmdir()`: Deletes a directory.

Reading a text file synchronously

- To read a text file synchronously, you can use the readFileSync() function from the fs module.

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

try {
    const contents = fs.readFileSync('data.txt', 'utf8');
    console.log(contents);
} catch (err) {
    console.error(err);
}
console.log('finish reading the file')

// "Hello! This is the content of data.txt"
// finish
```

A Blocking Call
A function or operation that stops program execution until it finishes.

Reading a text file without encoding

- Reading a text file without providing a encoding by default will result in a Buffer array

```
const fs = require('fs');
try {
    const contents = fs.readFileSync('data.txt');
    console.log(contents);
} catch (err) {
    console.error(err);
}
```

data.txt
1 ABC

```
mvmanh@Mais-MacBook-Pro node_app % node index.js
<Buffer 41 42 43>
```

Reading a text file using a Callback

- The `readFile()` function takes three arguments: the path to the file, the encoding, and a callback function.

```
fs.readFile('data.txt', 'utf8', (err, contents) => {
    if (err) {
       console.error(err);
    } else {
       console.log(contents);
    }
});
console.log('finish’);

// finish
// “Hello! This is the content of data.txt”
```


Reading a text file using a Callback

- The `readFile()` function takes three arguments: the path to the file, the encoding, and a callback function.

```
fs.readFile('data.txt', 'utf8', (err, contents) => {
    if (err) {
       console.error(err);
    } else {
       console.log(contents);
    }
});
console.log('finish’);

// finish
// “Hello! This is the content of data.txt”
```

Using a calback
Prevents blocking and maintains responsiveness. Program continues executing while waiting for file read to finish. But it can lead to "callback hell" with deeply nested code, reducing readability.

Reading a text file Asynchronously

- **Asynchronous** code is code that does not block the main thread while it is executing.

```
async function fileReadingExample() {
    try {
        const contents = await fs.promises.readFile('data.txt', 'utf8');
        console.log(contents);
    } catch (err) {
        console.error(err);
    }
}

fileReadingExample()
console.log('finish reading the file')

// finish
// “Hello! This is the content of data.txt”
```

An Asynchronous, Non-Blocking Call
an operation that allows the program to continue running while waiting for the call to complete, enabling other tasks to be processed simultaneously.

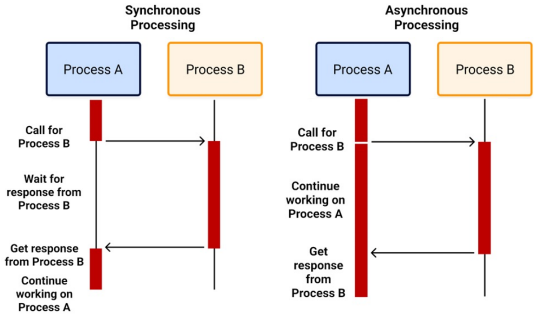
Reading a text file Asynchronously

```
async function readFile() {
    try {
        const contents = await fs.promises.readFile('data.txt', 'utf8');
        console.log(contents);
    } catch (err) {
        console.error(err);
    }
}

readFile()
console.log('finish reading the file')
```

async and await
The **await** keyword pauses the execution of the async function until the Promise is resolved. While the Promise is pending, the thread is free to run other code.

Synchronous vs. Asynchronous



Technical Examples

- **Synchronous Processing**
  - User Interfaces: Humans expect an immediate response when they interact with a computer!
  - HTTP APIs: HTTP APIs pass requests and responses in a synchronous fashion
- **Asynchronous Processing**
  - Batch-processing: is a data-processing method to handle large amounts of data asynchronously.
  - Long-running tasks: such as fulfilling an order placed on an e-commerce site are best handled asynchronously

The OS Module

The OS module

- The **os** module provides information about the computer operating system.
- It provides properties and methods for interacting with the operating system, such as:
  - The os.hostname() method returns the hostname of the computer.
  - The os.freemem() method returns the amount of free memory on the system.
  - The os.totalmem() method returns the total amount of memory on the system.
  - The os.type() method returns the type of operating system (e.g., "Linux", "Windows", "macOS").
  - The os.platform() method returns the platform of the operating system (e.g., "linux", "darwin", "win32").
  - The os.userInfo(): Returns information about the currently effective user.

The OS module

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

console.log('Architecture:', os.arch()); // arm64
console.log('Platform:', os.platform()); // drawin
console.log('CPU Info:', os.cpus()); // { model: 'Apple M1 Ultra', speed: 2400 }
console.log('Total Memory:', os.totalmem() / (1024 * 1024), 'MB');
console.log('Free Memory:', os.freemem() / (1024 * 1024), 'MB');
console.log('Hostname:', os.hostname()); // '/Users/mvmanh'

console.log('Home Directory:', os.homedir());
console.log('Network Interfaces:', os.networkInterfaces());
console.log('Load Averages:', os.loadavg());
```


The Path Module

The Path module

- The **path** module provides utilities for working with file paths.
- It provides methods for splitting paths into their components, joining paths together, and manipulating paths in other ways.
  - `path.join()`: Joins two or more paths together.
  - `path.basename()`: Returns the filename from a path.
  - `path.dirname()`: Returns the directory name from a path.
  - `path.extname()`: Returns the file extension from a path.
  - `path.normalize()`: Normalizes a path by removing redundant separators, ensuring that it is absolute.
  - `path.isAbsolute()`: Returns a boolean indicating whether a path is absolute.

The Path module

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

const filePath = '/user/documents/file.txt';

console.log(path.extname(filePath)); // .txt
console.log(path.basename(filePath)); // file.txt
console.log(path.dirname(filePath)); // /user/documents
console.log(path.basename(path.dirname(filePath))); // documents

const joinedPath = path.join('/user', 'documents', 'file.txt');
console.log(joinedPath); // /user/documents/file.txt
```

The URL Module

The URL module

- The **url** module provides utilities for parsing and manipulating URLs.
- It provides methods for extracting the different components of a URL, such as the protocol, hostname, port, path, and query string.
- It also provides methods for creating new URLs, appending query parameters, and resolving relative URLs against a base URL.

The URL module

- `url.parse()`: Parses a URL string and returns a URL object.
- `url.format()`: Formats a URL object and returns a URL string.
- `url.searchParams()`: Returns a URLSearchParams object representing the query parameters.
- `url.hostname()`: Returns the hostname of a URL.
- `url.port()`: Returns the port of a URL.
- `url.pathname()`: Returns the pathname of a URL.
- `url.search()`: Returns the query string of a URL.

The URL module

```
const url = require('url')
const obj = url.parse('https://www.example.com:8080/path/page?query=value#section', true);
console.log(obj);
```

```
Url {
    protocol: 'https:',
    slashes: true,
    auth: null,
    host: 'www.example.com:8080',
    port: '8080',
    hostname: 'www.example.com',
    hash: '#section',
    search: '?query=value',
    query: { query: 'value' },
    pathname: '/path/page',
    path: '/path/page?query=value',
    href: 'https://www.example.com:8080/path/page?query=value#section'
}
```

The URL module

```
const url = require('url')

const obj = url.parse('https://www.example.com:8080/path/page?query=value#section', true);
const {hostname, query} = obj // destructing assignment

console.log(hostname) // www.example.com
console.log(query) // { query: 'value' }
```


The QueryString Module

Query String Example

https://example.com/product?category=electronics&brand=apple&uid=admin%40gmail.com

category=electronics&brand=apple&uid=admin%40gmail.com

Variable Name	Variable Value
category	electronics
brand	apple
uid	admin@gmail.com

The QueryString module

- The **querystring** module provides utilities for parsing and formatting URL query strings.
- It provides two main methods: `parse()` and `stringify()`.
- The `parse()` method parses a URL query string into an object, where the keys are the query parameter names and the values are the query parameter values.
- The `stringify()` method formats an object into a URL query string.
- The QueryString module is a useful tool for working with URL query strings in Node.js applications.

The QueryString module

- Here is an example of how to use the `parse()` method to parse a URL query string:

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

const query = "username=admin&password=123abc";
const data = querystring.parse(query);

console.log(data); // { username: 'admin', password: '123abc' }
```


The QueryString module

- Here is an example of how to use the `stringify()` method to format an object into a URL query string:

```
const qs = require("querystring");

const account = {
    username: "admin",
    email: "admin@gmail.com",
    password: "123456"
};
const query = qs.stringify(account);

console.log(query); // username=admin&email=admin%40gmail.com&password=123456
```


A quick review of URL Encoding

- URL encoding converts characters into a format that can be safely transmitted via URLs.
- Special characters like spaces, symbols, and non-ASCII characters are encoded into **%** followed by **two hexadecimal digits**.

@ → %40

Character	URL Encoding Code
Space	%20
@	%40
#	%23
+	%2B

The Net Module

The Net module

- The **net** module provides an asynchronous network API for creating stream-based TCP or IPC servers and clients.
- It provides methods for creating sockets, connecting to servers, sending and receiving data, and handling errors.
  - `net.createServer()`: Creates a new TCP or IPC server.
  - `net.createConnection()`: Creates a new TCP or IPC connection to a server.
  - `socket.write()`: Writes data to a socket.
  - `socket.read()`: Reads data from a socket.
  - `socket.on()`: Attaches a listener to an event.
  - `socket.error()`: Returns an error object if an error occurs.

The Net module

- Here are some specific examples of when you might use the net module:
  - To create a web server that can handle HTTP requests.
  - To create a chat app that allows users to send messages to each other in real time.
  - To create a multiplayer game that allows players to compete against each other online.
  - To transfer files from one computer to another over the network.
  - To access a DNS server to resolve a domain name to an IP address.

The HTTP Module

http module

- The **http** module is a built-in module in Node.js that allows you to create HTTP servers and clients.
- It provides a number of functions for creating and managing HTTP servers and clients.
- Some of the most commonly used functions in the HTTP module include `http.createServer()`, `http.request()`, and `http.get()`.
- The HTTP module is a powerful tool that can be used to create a variety of HTTP-based applications.

Sending a GET request

1. The **http** module provides functions for making HTTP requests and responses.

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

2. Create a variable for the URL you want to make a request to.

```
const URL = 'https://jsonplaceholder.typicode.com/users';
```

3. Use the `http.get()` method to make the request.

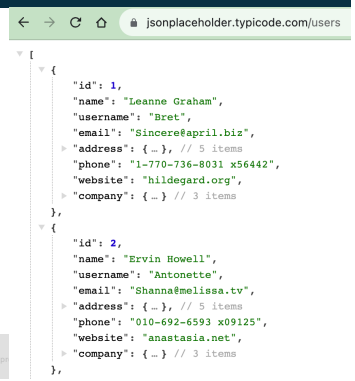
```
const request = http.get(URL, handleResponse);
```

4. Use `request.on('error')` method to add a callback that handle any error

```
request.on('error', (err) => {
    console.error(err);
});
```


Sending a GET request

https://jsonplaceholder.typicode.com/users



Sending a GET request

5. Create a function to handle the response from the request:

```
const handleResponse = (res) => {
    // step 6,7 and 8 are placed here
};
```

6. Call the `res.on()` method to attach the response handler function to the request.

```
const handleResponse = (res) => {
    let data = [];
    res.on('data', ...);
    res.on('end', ...);
};
```


Sending a GET request

7. Use the `res.on('data')` event handler to listen for the data chunks of the response.

```
...
let data = [];
res.on('data', (chunk) => {
    data.push(chunk);
});
...
```

Sending a GET request

8. Use the `res.on('end')` event handler to listen for the end of the response.

```
...
let data = [];
res.on('end', () => {
    const response = Buffer.concat(data).toString();
    const users = JSON.parse(response);
    // process the received data here
    console.log(users)
});
...
```



```
const http = require('http');
const URL = 'https://jsonplaceholder.typicode.com/users';

const handleResponse = (res) => {
    let data = [];
    res.on('data', (chunk) => {
        data.push(chunk);
    });

    res.on('end', () => {
        const response = Buffer.concat(data).toString();
        const users = JSON.parse(response);

        console.log(users)
    });
}; // handleResponse
```

```
const request = http.get(URL, handleResponse);
request.on('error', (err) => {
    console.error(err);
});
```

```
mvmanh@Mac-Studio app % node index.js
[
  {
    id: 1,
    name: 'Leanne Graham',
    username: 'Bret',
    email: 'Sincere@april.biz',
    address: {
      street: 'Kulas Light',
      suite: 'Apt. 556',
      city: 'Gwenborough',
      zipcode: '92998-3874',
      geo: [Object]
    },
```


Stream

Stream in Node.js

- Streams are objects that let you read data from a source or write data to a destination in continuous fashion. There are four types of streams:
  - Readable – Stream which is used for read operation.
  - Writable – Stream which is used for write operation.
  - Duplex – Stream which can be used for both read and write operation.
  - Transform – A type of duplex stream where the output is computed based on input.

Stream in Node.js

- Each type of **Stream** is an **EventEmitter** instance and throws several events at different instance of times.
  - data – This event is fired when there is data is available to read.
  - end – This event is fired when there is no more data to read.
  - error – This event is fired when there is any error receiving or writing data.
  - finish – This event is fired when all the data has been flushed to underlying system.

Reading from a Stream

1. import the **fs** module, which contains the **createReadStream()** function.

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

2. call the **createReadStream()** function to create a readable stream from the file **data.txt**.

```
const stream = fs.createReadStream('data.txt');
```

3. listen for the **data** event on the readable stream

```
stream.on('data', (chunk) => {
    // Do something with the chunk of data.
    console.log(chunk.toString());
});
```


Reading from a Stream

4. listen for the **end** event on the readable stream

```
stream.on('end', () => {
    // The file has been read to completion.
    console.log('The file has been read.');
});
console.log('This is the end of the index.js file')
```


Reading from a Stream

- **The full source code**

```
const stream = fs.createReadStream('data.txt');

stream.on('data', (chunk) => {
    console.log(chunk.toString());
});

stream.on('end', () => {
    console.log('The file has been read.');
});
console.log('This is the end of the index.js file')
```

```
mvmanh@Mac-Studio app % node index.js
This is the end of the index.js file
This is the content of
the file data.txt (2 lines)
The file has been read.
```

Stream vs Promise

```
async function readFile() {
    try {
        const contents = await fs.promises.readFile('data.txt');
        console.log(contents);
    } catch (err) {
        console.error(err);
    }
}
```

Promised-based version

Stream-based version

```
const stream = fs.createReadStream('data.txt');
let chunks = []
stream.on('data', (chunk) => {
    chunks.push(chunk)
});
stream.on('end', () => {
    console.log(chunks.toString()) // result
});
```


Aspect	fs.createReadStream (Stream)	fs.promises.readFile (Promise)
Reading Approach	Streaming, **read in chunks**	Buffering, **reads entire file at once**
Memory Usage	Memory-efficient for large files	Loads entire file into memory
Performance	Better for large files, efficient streaming	May have performance issues with large files
Use Cases	Large files, log files, multimedia processing	Smaller files, config files, JSON handling
Code Complexity	Requires event listeners, slightly complex	Simple Promise-based approach

Sending a POST request

Sending a POST request

1. The http module provides functions for making HTTP requests and responses.
2. Create a JavaScript object representing the data you want to send as JSON.

```
const account = {
    name: 'John Doe',
    email: 'johndoe@example.com',
    password: '123456'
};
```

3. Convert the JavaScript object to a JSON string using `JSON.stringify()`.

```
const data = JSON.stringify(account);
```

Sending a POST request

3. Define the options for the POST request, including the URL, headers, and method.

```
const data = JSON.stringify(account);
const options = {
    hostname: 'api.example.com',
    path: '/students',
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        'Content-Length': data.length
    }
};
// will be sent to: http://api.example.com/students/
```


Sending a POST request

4. Create a request using `http.request()` and write the JSON data to the request body.

```
const request = http.request(options, (res) => {
    let buffer = '';
    res.on('data', (chunk) => {
        buffer += chunk;
    });
    res.on('end', () => {
        // Handle the response data
        const result = JSON.parse(buffer);
        console.log(result); // {success: true, message: 'Add success'}
    });
});
```


Sending a POST request

5. Set up an event listener to handle errors that might occur during the request

```
request.on('error', (err) => {
    console.error('Error occured:', err);
});
```

6. Proceed to write the JSON data to the request body and end the request

```
request.write(data); // json string
request.end();
```



```
const account = {
   name: 'John Doe',
   email: 'johndoe@example.com',
   password: '123456'
};
const data = JSON.stringify(account);

const options = {
   hostname: 'api.example.com',
   path: '/students',
   method: 'POST',
   headers: {
      'Content-Type': 'application/json',
      'Content-Length': data.length
   }
};
```

```
const request = http.request(options, (res) => {
   let buffer = '';
   res.on('data', (chunk) => {
      buffer += chunk;
   });
   res.on('end', () => {
      const result = JSON.parse(buffer);
      console.log(result);
   });
});
request.on('error', (err) => {
   console.error('Error occured:', err);
});

request.write(data); // can be called multiple times
request.end();
```

Creating a web server

Creating a web server

- Create an HTTP server using the createServer method of the http module:

```
const server = http.createServer((req, res) => {
    // Handle incoming requests here
});
```

- Inside the callback function, you can handle incoming requests and send responses:

```
const server = http.createServer((req, res) => {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.write('Hello, World!\n'); // can call multiple times
    res.end('Welcome to Node.js\n');
});
```


Creating a web server

- To make your server listen on a specific port, use the listen method:

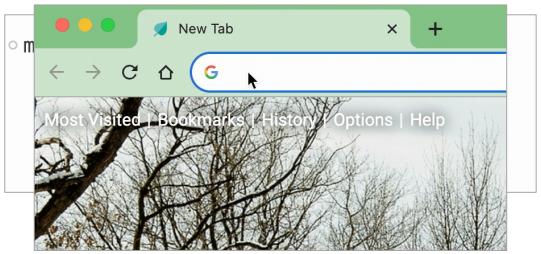
```
const port = 3000;
server.listen(port, () => {
    console.log(`Server is running on http://localhost:${port}`);
});
```

- In your terminal, run the following command to start your Node.js server:

```
node server.js
```

- Access Your Server: **http://localhost:3000**

Creating a web server



Serving static contents

Serving a html file

1. Create an HTML file that you want to serve.
2. Import the http and fs modules.
3. Create a new http server.
4. Use the `fs.readFile()` method to read the HTML file into a buffer.
5. Use the `res.sendFile()` method to send the HTML file to the client.
6. Start the server.

Serving a html file

```
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
    const filePath = './index.html';
    const html = fs.readFileSync(filePath);
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(html);
});
server.listen(3000);
```

Using the `readFileSync()` function can lead to performance problems if the file is large or if there are a lot of requests coming in.

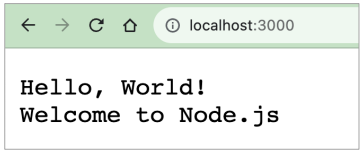
MIME Types

- Multipurpose Internet Mail Extensions are used to identify the type of content in a file.
- They are used by email clients, web browsers, and other applications to determine how to handle a file.
- The type specifies the general category of the content, such as text, image, or audio.
- The subtype specifies the specific format of the content, such as HTML, JPEG, or MP3.

MIME Types

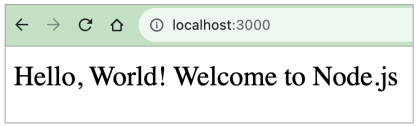
- Here are some examples of MIME Types:
  - text/plain: This is the MIME Type for plain text files.
  - text/html: This is the MIME Type for plain text files.
  - image/jpeg: This is the MIME Type for JPEG images.
  - image/png: This is the MIME Type for PNG images.
  - audio/mp3: This is the MIME Type for MP3 audio files.
  - application/pdf: This is the MIME Type for PDF files.
  - video/mpeg: This is the MIME Type for MPEG video files.

text/plain vs text/html



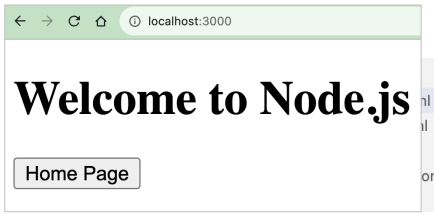
text/plain

text/html

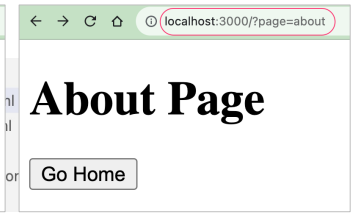


Serving multiple files

Serving HTML Files Based on Query Strings



index.html



about.html

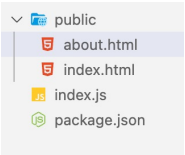
Handling Query String

- To parse and extract query string parameters from the request URL, we can use the url module.

```
const server = http.createServer((req, res) => {
    console.log(req.url)
       http://localhost:3000/resources/student?type=international&page=2
       /resources/student?type=local&page=2
    const parsedUrl = url.parse(req.url, true);
    console.log(parsedUrl.query) // { type: 'local', page: '2' }
    console.log(parsedUrl.query.type) // local
    res.end()
});
```


Serving HTML Files Based on Query Strings

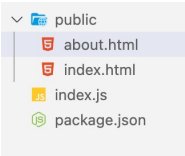
```
async function loadHtml(fileName) {
    let filePath = path.join(__dirname, 'public', 'index.html');
    if (fileName === 'about') {
        filePath = path.join(__dirname, 'public', 'about.html');
    }
    return await fs.promises.readFile(filePath, 'utf8')
}
```



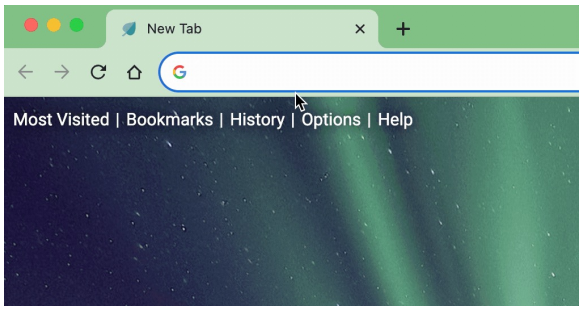
Serving HTML Files Based on Query Strings

```
const server = http.createServer(async (req, res) => {
    const parsedUrl = url.parse(req.url, true);
    const {page} = parsedUrl.query;
    const html = await loadHtml(page)

    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(html);
});
```



Serving HTML Files Based on Query Strings



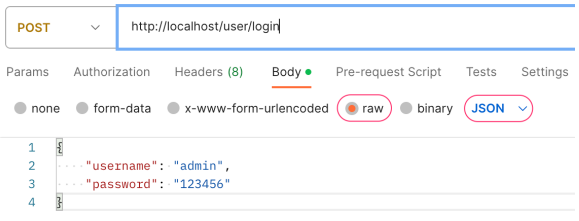
Handling incoming POST request

Handling Incoming POST Request

- Handling POST requests is essential when you need to receive data from clients, such as form submissions or JSON data.
- Two common types of incoming request messages: JSON vs. Form UrlEncoded.

JSON Data

- JSON is a lightweight data interchange format that is easy for humans to read and write.
- JSON data is typically sent with the Content-Type: application/json header.

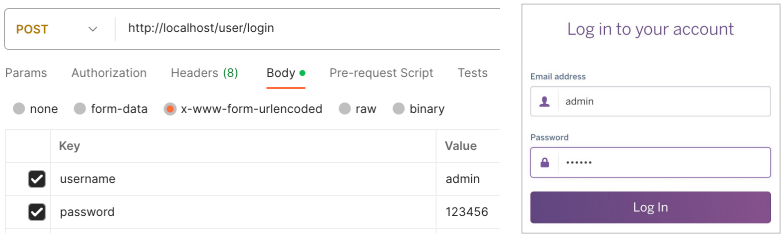


Form-UrlEncoded

- This format is commonly used when sending data from HTML forms to the server.
- This data is typically sent with the Content-Type: application/x-www-form-urlencoded header.
- Data is encoded as key-value pairs separated by **&**, and keys and values are URL-encoded.
- Sample payload sending from client to userver: user=admin%40gmail.com&pass=123456

Form-UrlEncoded

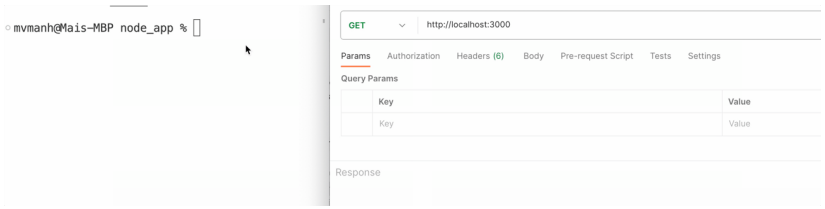
- This data is typically sent with the Content-Type: application/x-www-form-urlencoded header.
- Sample payload sending from client to userver: user=admin%40gmail.com&pass=123456



Incoming Form UrlEncoded POST request


```
const formUrl = 'application/x-www-form-urlencoded'
const server = http.createServer((req, res) => {
    if (req.method === 'POST' && req.headers['content-type'] === formUrl) {
       let data = '';
       req.on('data', chunk => {
          data += chunk;
       });
       req.on('end', () => {
           const {username, password} = querystring.parse(data);
           console.log(username, password);
           res.writeHead(200, { 'Content-Type': 'text/plain' });
           res.end('Login Success');
       });
    } // end if
}); // end createServer
```

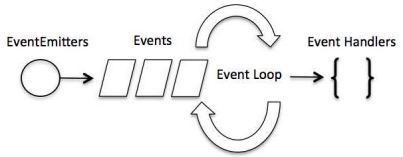

Incoming JSON Request



Events

Events

- In Node.js, events are a core part of the event-driven programming paradigm.
- Events are essentially signals that something has happened in your application, such as a user clicking a button, a file being read, or a network request completing.



Event Emitter

- The **EventEmitter** class in Node.js is a fundamental building block for working with events. It provides a way to create, emit, and handle custom events.
  - To listen for an event, you use the **on()** method of the EventEmitter object.
  - When an event is emitted, all of the functions that are listening for that event are called.
  - You can also remove listeners using the **off()** method.

Event Emitter

1. To use the EventEmitter class, you first need to import the event module to your application:

```
const EventEmitter = require('events');
```

2. Then, you can create an instance of EventEmitter to work with:

```
const mySource = new EventEmitter();
```

3. You can define custom events by using the **on** method:

```
mySource.on('finish', (data) => {
    console.log('Handle the "finish" event with data: ', data);
});
```

Event Emitter

4. To trigger (emit) a custom event, you can use the **emit** method:

```
mySource.emit('finish', { message: 'Hello, world!' });
```

5. This will call all the event handlers that have been registered for the 'finish' event.

```
mvmanh@Mais-MacBook-Pro node_app % node index.js
Handle the "finish" event with data:  { message: 'Hello, world!' }
```

6. You can also remove event listeners using the removeListener method:

```
mySource.removeListener('finish', myEventHandler);
```


Event Emitter

- Full source code example

```
const EventEmitter = require('events');
const mySource = new EventEmitter();
mySource.on('finish', (data) => {.   // register for event handler
    console.log('Handle the "finish" event with data: ', data);
});
mySource.emit('finish', { message: 'Hello, world!' }); // trigger event
```


Event Emitter in Real-life Example

- In this example, we'll create a DownloadManager class that handles the download process and emits events for start, progress, and completion.
- An instance of the class is then exported as a module (line 32)

downloader.js > ...

```
1   const EventEmitter = require('events');
2   const url = require('url')
3   const path = require('path')
4
5   class DownloadManager extends EventEmitter {
6     constructor() {…
10    }
11
12    startDownload(link) {…
29    }
30  }
31
32  module.exports = new DownloadManager()
```


Event Emitter in Real-life Example

- In the index.js, we then import the local downloader.js module, register some event handlers and finally call the startDownload() method, passing a url for downloading a file

EXPLORER

NODE_APP
- downloader.js
- index.js
- package.json

index.js | downloader.js

index.js > ...

```
1   const downloader = require('./downloader')
2
3   downloader.on('start', (size) => console.log(`Start downloading a file: ${size} MB`))
4   downloader.on('finish', (file) => console.log('file saved at: ' + file))
5   downloader.on('progress', (percent) => console.log(`Downloading ${percent}%`))
6
7   downloader.startDownload('http://example.com/video.mp4')
```

Running the app in action

index.js | downloader.js

index.js > ...

```
1  const downloader = require('./downloader')
2
3  downloader.on('start', (size) => console.log(`Start downloading a file: ${size} MB`))
4  downloader.on('finish', (file) => console.log('file saved at: ' + file))
5  downloader.on('progress', (percent) => console.log(`Downloading ${percent}%`))
6
7  downloader.startDownload('http://example.com/video.mp4')
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
mvmanh@Mais-MacBook-Pro node_app %
```

```
class DownloadManager extends EventEmitter {
    constructor() {
        super();
        this.totalSize = 100;
        this.downloadedSize = 0;
    }
    startDownload(link) {
        // perform download logic
    }
}
```

```
startDownload(link) {
    this.emit('start', this.totalSize); // start event

    const id = setInterval(() => {
        this.downloadedSize += 10;
        this.emit('progress', this.downloadedSize); // Progress completed
        if (this.downloadedSize === this.totalSize) {
            clearInterval(id);
            const fileName = path.basename(url.parse(link).pathname)
            this.emit('finish', 'c:/user/admin/' + fileName); // Finish event
        }
    }, 500);
}
```

User-Defined Modules

User-defined Modules

- User-defined modules are JavaScript files that contain reusable code.
- They can be used to encapsulate code and make it easier to maintain and reuse.
- To create a user-defined module, simply create a JavaScript file with the .js extension.
- The module's code should be placed in a function called module.exports.
- To use a user-defined module, you can use the require() function.
- The require() function takes the name of the module as its argument and returns the module's exported code.

Creating a new module

1. **Create a new JavaScript file** for your module, e.g. mathUtils.js
2. **Write your code**: Define functions, variables, or classes that you want to export from your module.

```
// mathUtil.js
function add(a, b) {
    return a + b;
}
function subtract(a, b) {
    return a - b;
}
```

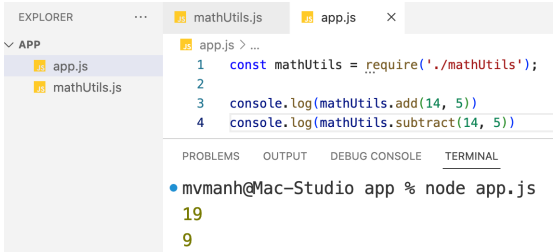

Creating a new module

3. **Exporting** Functions and Variables

```
function add(a, b) {
    return a + b;
}
function subtract(a, b) {
    return a - b;
}
// Make them accessible in other files
module.exports = {add, subtract};
```

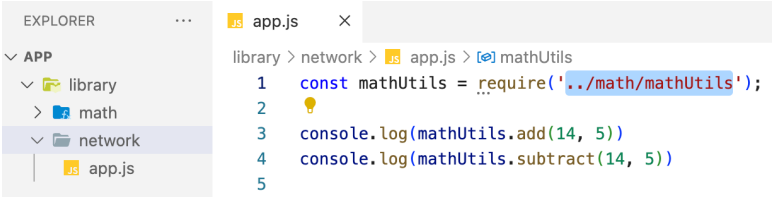

Creating a new module

4. In a different file, use the **require()** function to import the module you created.



Creating a new module

- If your files are placed in different directories, you need to modify the require path accordingly



Using destructing assignment with require()

- When requiring a module in Node.js, you can use destructuring assignment to extract specific functions, variables, or objects from the imported module

```
const {add: sum, subtract} = require('./mathUtils');

console.log(sum(14, 5))
console.log(subtract(14, 5))
```


Single Export

- You can use single exports, also known as default exports, to export a single function, class, or value as the primary export from a module.
- This approach is useful when you want to provide a clear and straightforward way to access the main functionality of your module.

```
function add(a, b) {
    return a + b;
}
//module.exports = {add, subtract};
module.exports = add
```

```
const addFunction = require('./mathUtils');
const sum = require('./mathUtils');

console.log(addFunction(14, 5))
console.log(sum(14, 5))
```