Web Application Development

Node.js Fundamentals

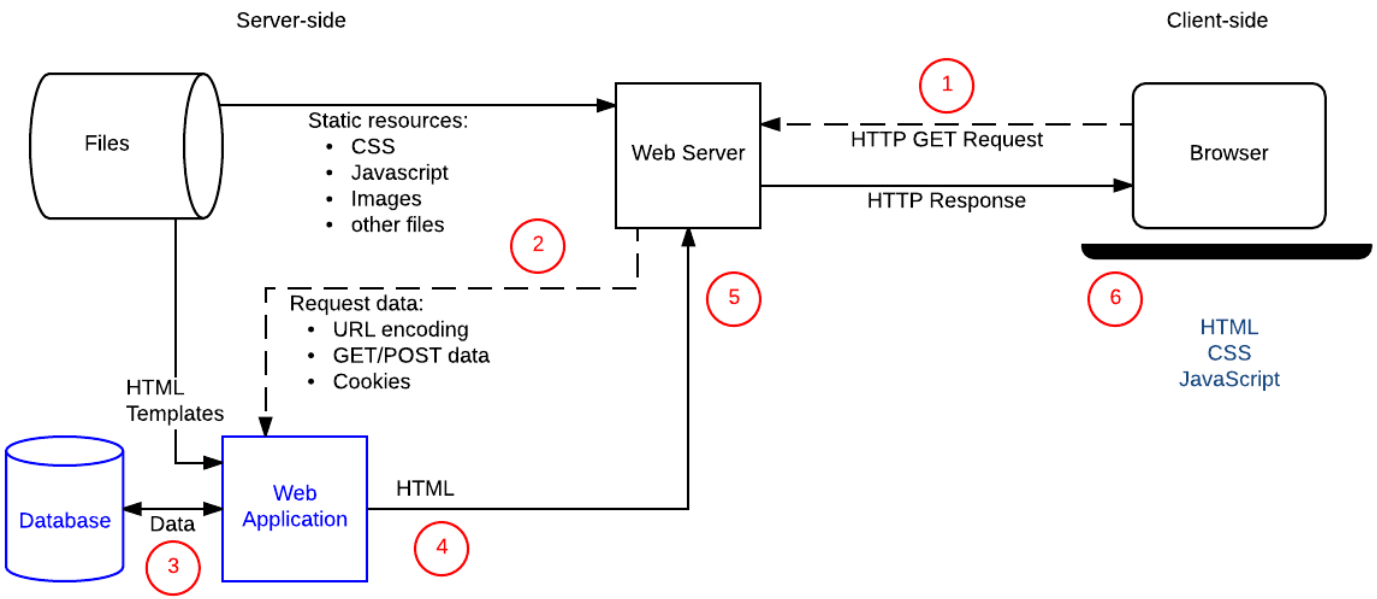
KIEU Quoc Viet HUYNH Vinh Nam

Information and Communication Technology Laboratory (ICTLab),
University of Science and Technology of Hanoi

Hanoi, Sept 2024

Table of Contents

Node.js Fundamentals



What is Node.js?

- **Definition**:
  - Node.js is a JavaScript runtime built on Chrome's V8 engine that allows JavaScript to be executed on the server.
  - Enables server-side programming with JavaScript.
- **Key Points**:
  - Uses non-blocking, event-driven architecture.
  - Primarily used for building scalable and fast network applications.

What is Node.js? (cont.)

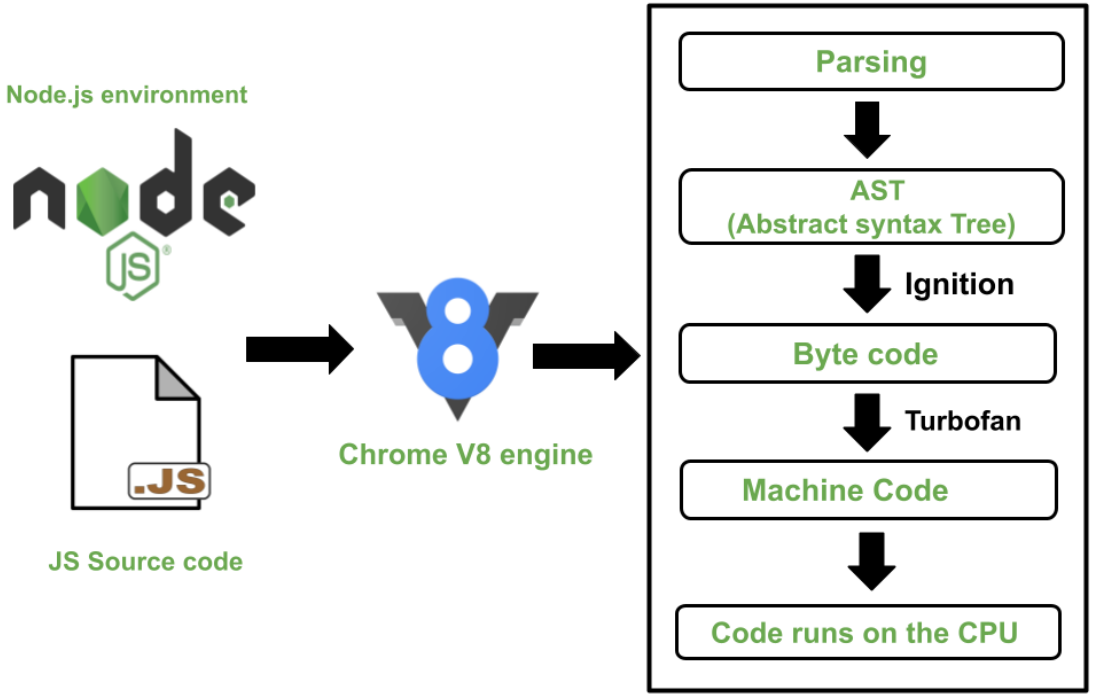
- **Why is it important?**:
  - JavaScript can now be used end-to-end (front-end and back-end), reducing the need for multiple programming languages across an application.

Node.js Architecture

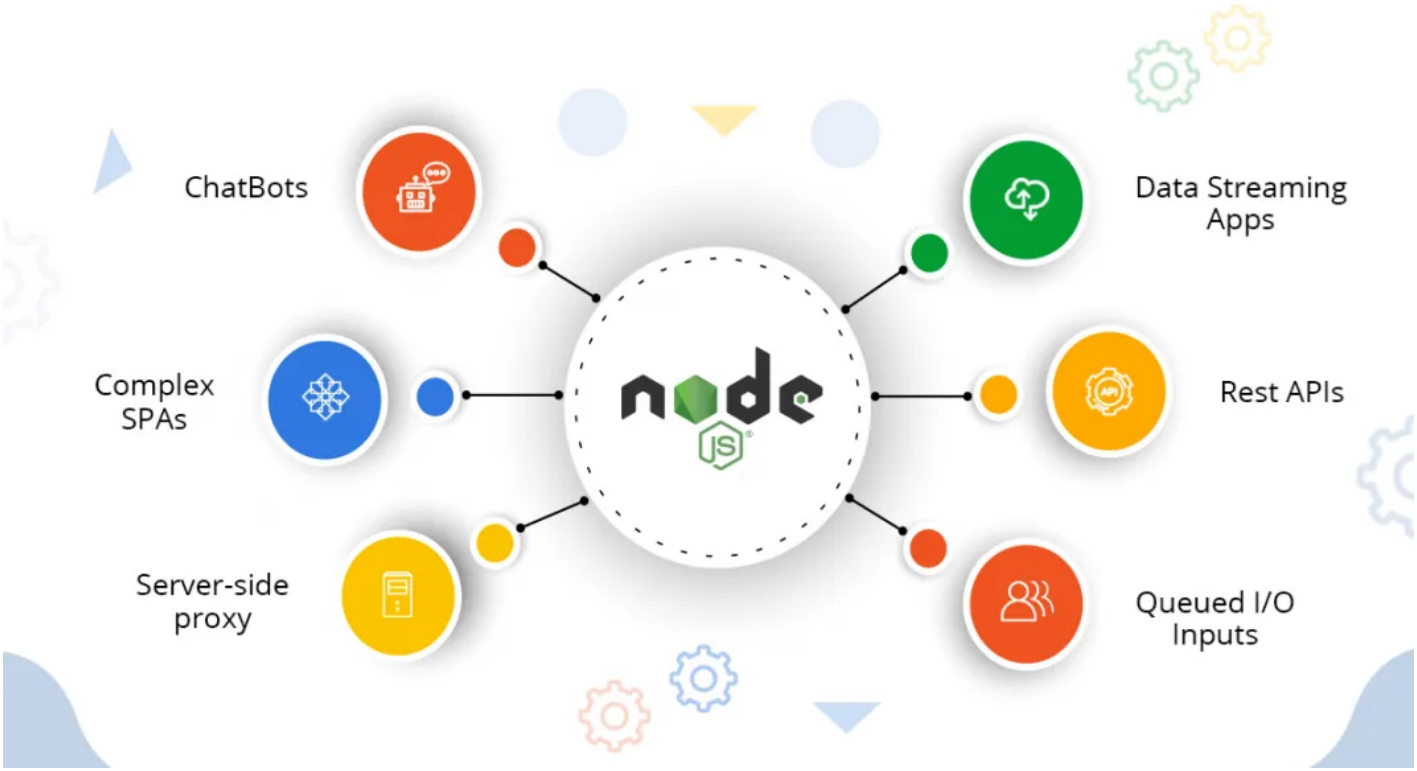
- **Single-threaded, Event-Driven**:
  - Node.js operates on a single thread using the event loop to handle multiple connections concurrently.
  - Unlike traditional multithreaded servers, Node.js is non-blocking, allowing it to efficiently manage multiple requests without creating new threads for each.
- **Event Loop**:
  - Explain the event loop: The core of Node.js' non-blocking architecture.
  - Handles I/O operations asynchronously, queuing up operations and processing them once they're complete.

Node.js Architecture (cont.)

- **V8 Engine**:
  - Node.js uses Google's V8 JavaScript engine to convert JavaScript code into machine code.



Use Cases of Node.js



Use Cases of Node.js (cont.)

- **Real-Time Applications**: Suitable for chat applications, online collaboration tools, and multiplayer games.
- **API Servers**: Commonly used to build RESTful APIs and microservices.
- **Single-Page Applications (SPAs)**: Works well with frameworks like React or Vue.js to handle dynamic content.
- **Data Streaming Applications**: Ideal for audio/video streaming due to its event-driven nature.
- **IoT Applications**: Handles large amounts of data from IoT devices efficiently.

Recap: Why Node.js?

- Node.js allows JavaScript to run on the server.
- Uses a **single-threaded, event-driven architecture** to handle multiple connections concurrently.
- Powered by the **V8 JavaScript engine** for fast execution.
- Ideal for real-time, API-based, and data-heavy applications.

Key takeaway: Node.js excels in non-blocking, asynchronous operations.

Installing Node.js & NPM

Steps to Install:

1. Go to `https://nodejs.org/` and download the appropriate installer for your OS.
2. Install Node.js along with npm (Node Package Manager).
3. Verify installation:

```
node -v
npm -v
```

Installing Node.js & NPM (cont.)

Command Prompt

```
Microsoft Windows [Version 10.0.22631.4169]
(c) Microsoft Corporation. All rights reserved.

C:\Users\USTH>node -v
v18.12.1

C:\Users\USTH>npm -v
8.19.2

C:\Users\USTH>
```

Setting Up a Basic Node.js Project

Steps to Set Up:

1. Create a new project directory:

```
mkdir nodejs-fundamentals
cd nodejs-fundamentals
```

2. Initialize the project using npm:

```
npm init -y
```

3. This creates a `package.json` file to manage the project.

Setting Up a Basic Node.js Project (cont.)

Command Prompt

```
C:\Users\USTH\nodejs-fundamentals>npm init -y
Wrote to C:\Users\USTH\nodejs-fundamentals\package.json:

{
  "name": "nodejs-fundamentals",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Understanding `package.json`

What is `package.json`?

- Contains metadata about your project.
- Tracks dependencies for libraries and packages.
- Defines custom **scripts** (e.g., commands to start a server).

Key Fields:

- `"name"`: Name of the project.
- `"version"`: Current version of the project.
- `"dependencies"`: Lists installed npm packages.
- `"scripts"`: Commands such as `"start": "node app.js"`.

Writing and Running the First Node.js Program

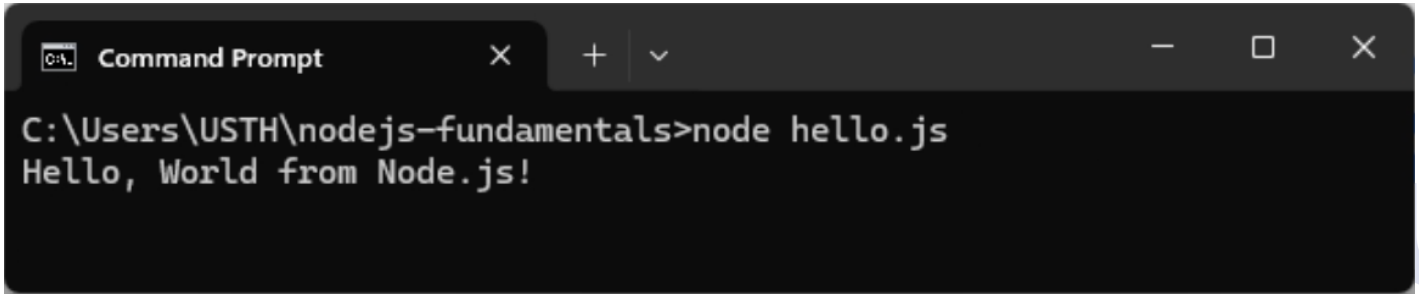
Create a file `hello.js`:

```
console.log('Hello, World from Node.js!');
```

Run the script in the terminal:

```
node hello.js
```

Output: `Hello, World from Node.js!`



Node.js as a JavaScript Runtime

What is Node.js?

- Node.js is a **JavaScript runtime** that allows JS to be executed on the server-side.
- Built on Chrome's V8 JavaScript engine.
- Provides an environment for building network applications, using non-blocking I/O operations.

Overview of Modules and `require()` Function

What are Modules?

- Modules in Node.js are independent blocks of reusable code.
- Node.js uses the `require()` function to load modules.

Example: Loading the `fs` Module

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

Built-in Modules: `fs`, `http`, `url`, etc.

Example: Using the fs (File System) Module

Reading a File Synchronously:

```
const fs = require('fs');
let data = fs.readFileSync('file.txt', 'utf8');
console.log(data);
```

Explanation:

- `fs.readFileSync()` reads a file synchronously, blocking execution until reading is complete.
- `console.log()` outputs the file content.

Creating Custom Modules

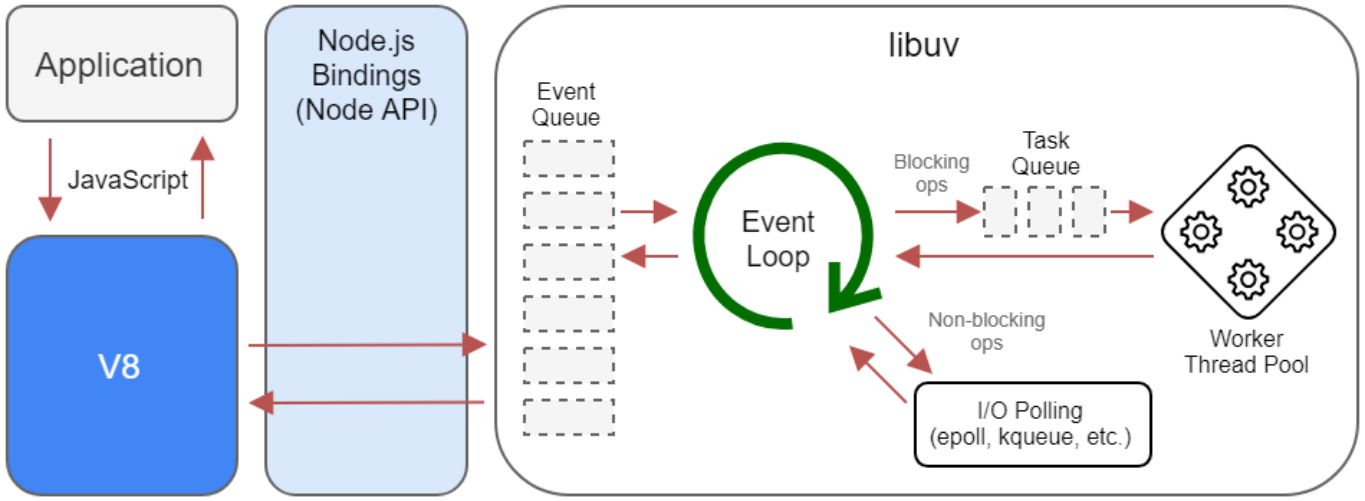
Creating and Exporting a Module:

```
// file: myModule.js
module.exports.sayHello = function() {
    console.log('Hello from myModule!');
};
```

Importing and Using the Custom Module:

```
const myModule = require('./myModule');
myModule.sayHello(); // Outputs: Hello from myModule!
```

Event-Driven Architecture in Node.js



What is Event-Driven Programming?

- Node.js uses an event-driven architecture, where actions trigger events.
- The **Event Loop** processes events asynchronously without blocking.

Example: Using EventEmitter

Creating an Event Listener:

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

// Register a listener
emitter.on('event', () => {
    console.log('An event occurred!');
});

// Emit the event
emitter.emit('event'); // Outputs: An event occurred!
```

Explanation:

- `emitter.on()` registers an event listener.
- `emitter.emit()` triggers the event.

Blocking vs Non-Blocking I/O

Blocking I/O:

- Operations execute in sequence, one after the other.
- Blocks further execution until the current operation is complete.

Non-Blocking I/O:

- Operations can be initiated without waiting for the previous operation to complete.
- Node.js excels in handling non-blocking, asynchronous operations.

Example of Blocking Code

Synchronous File Reading:

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

let data = fs.readFileSync('file.txt');
console.log(data);
console.log('File reading complete.');
```

Output Order:

- File content is printed first.
- "File reading complete" is printed only after file is fully read.

Example of Non-Blocking Code

Asynchronous File Reading:

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

fs.readFile('file.txt', (err, data) => {
    if (err) throw err;
    console.log(data);
});
console.log('File reading initiated.');
```

Output Order:

- "File reading initiated" is printed first.
- File content is printed later when reading is complete.

What are Callbacks?

Definition:

- A **callback** is a function passed as an argument to another function.
- It is executed after an asynchronous operation completes.

Callback Syntax:

```
function fetchData(callback) {
    // Simulate async operation
    setTimeout(() => {
        console.log('Data fetched!');
        callback();
    }, 1000);
}

fetchData(() => {
    console.log('Callback executed.');
});
```

Example: Callback Function

Asynchronous Operation with Callback:

```
function doSomething(callback) {
    console.log('Doing something...');
    callback();
}

function onComplete() {
    console.log('Task complete.');
}

doSomething(onComplete); // Outputs: Doing something... Task complete.
```

Error Handling in Callbacks

Handling Errors in Asynchronous Functions:

```
fs.readFile('file.txt', (err, data) => {
    if (err) {
        console.error('Error reading file:', err);
        return;
    }
    console.log(data);
});
```

Explanation:

- Check if `err` exists before proceeding.
- Use `return` to stop further execution if an error occurs.

Recap of Asynchronous Programming

Key Takeaways:

- Node.js uses non-blocking I/O operations to handle multiple requests concurrently.
- Callbacks are functions passed to asynchronous operations, executed when the task is complete.
- Proper error handling in callbacks is crucial for robust code.

Introducing the HTTP Module

The HTTP Module in Node.js:

- The built-in `http` module allows Node.js to create a basic web server.
- Provides functionality for handling HTTP requests and responses.
- No need for external libraries—just import and use.

Key Features:

- Create an HTTP server with `http.createServer()`.
- Listen for requests on a specific port.

Code Example: Basic HTTP Server in Node.js

Example Code:

```javascript
const http = require('http');
const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello, World!\n');
});

server.listen(3000, () => {
    console.log('Server running at http://localhost:3000/');
});
```

Explanation: This code creates a simple HTTP server that listens on port 3000.

Running the Server

Steps to Run the Server:

1. Save the code as `server.js` in your project folder.
2. In the terminal, navigate to the folder and run the server with:

```
node server.js
```

3. You should see a message like:

```
Server running at http://localhost:3000/
```

4. Open your browser and navigate to `http://localhost:3000/` to see the output.

Breakdown of the HTTP Server Code

Key Sections of the Code:

- `http.createServer()`: Creates the HTTP server and handles incoming requests.
- `res.statusCode = 200`: Sets the status code to 200, indicating a successful request.
- `res.setHeader('Content-Type', 'text/plain')`: Specifies the content type of the response as plain text.
- `res.end('Hello, World!')`: Ends the response and sends the data to the client.
- `server.listen(3000)`: Instructs the server to listen on port 3000.

Console Output: Displays when the server starts successfully.

Recap of Key Concepts

What We've Covered:

- Introduction to the Node.js HTTP module and its role in building a server.
- Writing a simple Node.js server to handle HTTP requests.
- Running and testing the server on `localhost`.
- Understanding the key components of an HTTP server in Node.js.

Next Steps: Preparing to integrate front-end with back-end using HTTP endpoints.

Questions and Next Steps

Any Questions?

Next Steps:

- Review the code examples we covered in class.
- Begin thinking about how to integrate front-end components with this HTTP server.
- The hand-on assignment will help solidify your understanding.

Hand-on Assignment

- Build a simple Node.js server that handles different types of requests:
  - Serve different types of content (e.g., HTML, JSON).
  - Set up different routes to respond to GET and POST requests.
- Use the concepts covered in class to break down the HTTP server code.
- Submit the project by the end of the week.

Tip: Start by expanding the basic server example and adding routes for different paths (e.g., `/`, `/about`, `/data`).

Thank you for listening!