Week 2 - Web Fundamentals & Core Modules - Part 1

Amr A Khllaf

August 15, 2025

Contents

1 Web Fundamentals (Protocols)

1.1 What are the Protocols?

- Protocols are a **set of rules** that define how data is transmitted over the web.

- Examples of protocols include **HTTP**, **HTTPS**, **FTP**, and **WebSocket**.
  1. **HTTP (HyperText Transfer Protocol)**: The foundation of data communication on the web.
  2. **HTTPS (HyperText Transfer Protocol Secure)**: An extension of HTTP that adds a layer of security by using SSL/TLS to encrypt data.
     - **SSL:** Secure Sockets Layer, a standard security technology for establishing an encrypted link between a server and a client.
     - **TLS:** Transport Layer Security, the successor to SSL, providing improved security and performance.
  3. **FTP (File Transfer Protocol)**: A standard network protocol used to transfer files from one host to another over a TCP-based network.
  4. **WebSocket**: A protocol that provides full-duplex communication channels over a single TCP connection, commonly used in real-time web applications.

1.2 Comparison between HTTP and HTTPS

Feature	HTTP	HTTPS
Security	No encryption	Encrypted using SSL/TLS
Port	Default port 80	Default port 443
Performance	Faster due to no encryption overhead	Slightly slower due to encryption
SEO	Not a ranking factor	Considered a ranking factor by Google
Trust	Less trustworthy	More trustworthy due to encryption
Use Cases	General web browsing	Secure transactions, sensitive data

1.2.1 What is the HTTP Request Response Full Cycle and DNS?

- **DNS Resolution**: Before the HTTP request is sent, the domain name is resolved to an IP address using DNS (Domain Name System).
- **Client Sends Request**: The client (browser) sends an HTTP request to the server.
- **Server Processes Request**: The server processes the request and prepares a response.
- **Server Sends Response**: The server sends an HTTP response back to the client.
- **Client Receives Response**: The client receives the response and renders the content.
- **Caching**: The client may cache the response for future requests to improve performance.

1.2.2 What is the Types of IP Addresses?

- **IPv4**: The most widely used IP version, consisting of 32 bits and usually represented as four decimal numbers separated by dots (e.g., 192.168.1.1).
- **IPv6**: The successor to IPv4, designed to address the shortage of available IP addresses. It consists of 128 bits and is represented as eight groups of hexadecimal numbers separated by colons (e.g., 2001:0db8:85a3:0000:0000:8a2e:0370:7334).

1.2.3 IPv4 Vs IPv6

Feature	IPv4	IPv6
Address Length	32 bits	128 bits
Address Format	Decimal (e.g., 192.168.1.1)	Hexadecimal (e.g., 2001:0db8:85a3:0000:0000:8a2e:0370:7334)
Number of Addresses	~4.3 billion	~340 undecillion
Header Size	20-60 bytes	40 bytes
Fragmentation	Handled by routers	Handled by the sender
Security	No built-in security	IPsec support
Use Cases	General web traffic	IoT, mobile devices, and future-proofing

1.3 What is the DNS Server ?

- A DNS (Domain Name System) server is a server that translates human-readable domain names (e.g., www.example.com) into IP addresses (e.g., 192.0.2.1) that computers use to identify each other on the network.
- DNS servers are a critical part of the internet infrastructure, enabling users to access websites using easy-to-remember domain names instead of numerical IP addresses.
- When a user enters a domain name in their web browser, the DNS server resolves the domain name to its corresponding IP address, allowing the browser to connect to the correct server and retrieve the requested content.
- DNS servers also provide additional features such as load balancing, redundancy, and security through DNSSEC (DNS Security Extensions).

1.4 What is Web Server ?

- A web server is a software or hardware that serves content to the web.
- It processes incoming network requests over HTTP and HTTPS and serves the requested content, which can be static (HTML files, images) or dynamic (generated by server-side scripts).
- Web servers play a crucial role in delivering web pages to users' browsers, handling requests, and managing resources.
- Popular web server software includes **Apache, Nginx, Microsoft Internet Information Services (IIS), and LiteSpeed.**

1.4.1 How Web Server Works?

- When a user enters a URL in their browser, the browser sends an HTTP request to the web server hosting the website.
- The web server receives the request and processes it, which may involve retrieving files from the file system, querying a database, or executing server-side scripts.
- Once the server has the requested content, it sends an HTTP response back to the browser, which includes the content (e.g., HTML, CSS, JavaScript) and a status code indicating the result of the request.
- The browser receives the response and renders the web page for the user to view.

- Web servers can also handle additional tasks such as logging requests, managing sessions, and implementing security measures like HTTPS.
- **How web server handles if we upload to it many Applications?**
  - Web servers can efficiently host multiple applications using several strategies:
    - **Virtual Hosting:** Serve multiple domain names or websites from a single server, allowing each application to have its own domain or subdomain.
    - **Containerization:** Use technologies like Docker to isolate applications, ensuring they run independently and securely on the same server.
    - **Load Balancing:** Distribute incoming traffic across multiple servers or application instances to prevent overload and improve reliability.
    - **Reverse Proxy:** Employ a reverse proxy to route requests to the appropriate application, cache content, and enhance performance.
    - **Subdomains and Ports:** Assign different subdomains (e.g., app1.example.com, app2.example.com) or ports (e.g., example.com:3000, example.com:4000) to separate applications on the same server.
- So the Steps of Request - Response are:
  1. User Enters URL in Browser
  2. DNS Server Translates Domain Name to IP Address
  3. Browser Sends HTTP Request to Web Server (Nginx or Apache)
  4. Web Server Processes Request
  5. Web Server Sends HTTP Response
  6. Browser Renders Web Page

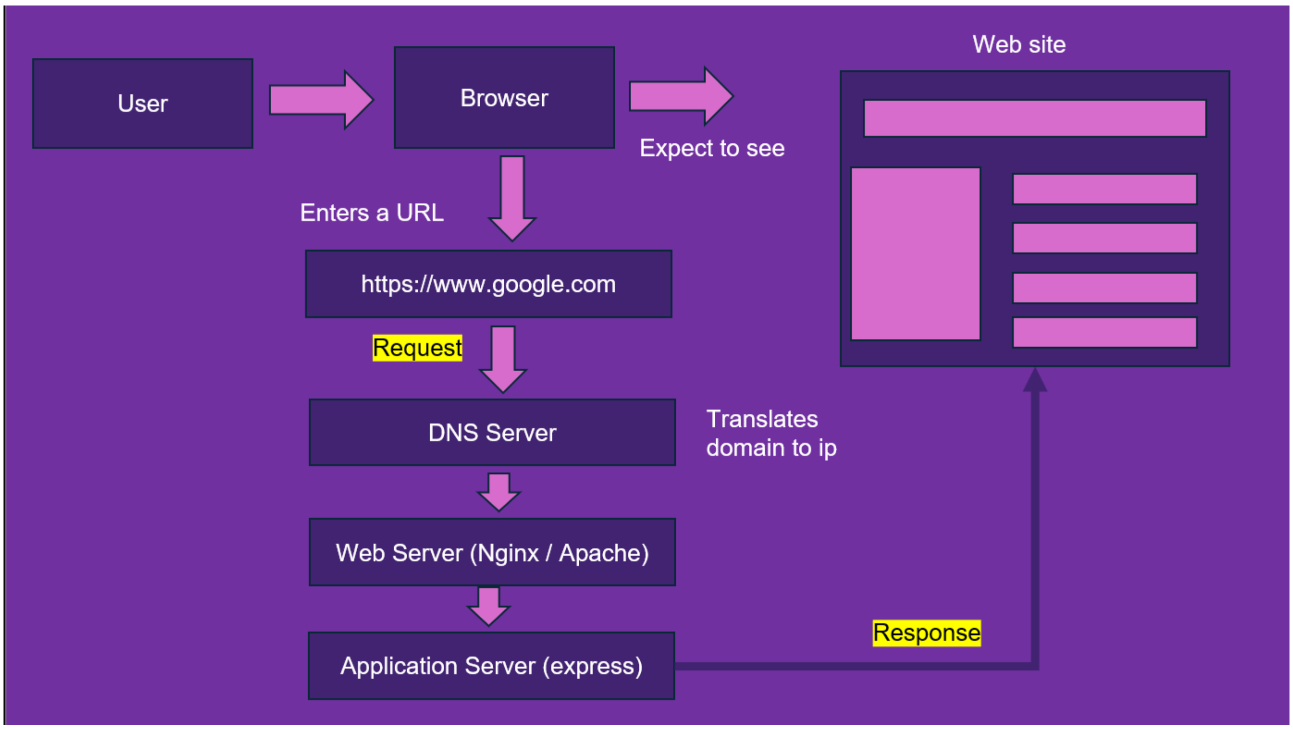


Figure 1: Steps Of Request - Response

2 URL (Uniform Resource Locator)

- A URL (Uniform Resource Locator) is the address used to access resources on the internet.
- It consists of several components, including the protocol (e.g., HTTP, HTTPS), domain name (e.g., www.example.com), port (optional), path (e.g., /path/to/resource), and query parameters (optional).
- URLs are essential for web navigation, allowing users to locate and retrieve specific resources from web servers.
- Example of a URL: `https://www.example.com:443/path/to/resource?query=param`

2.1 URL Structure

- **Protocol:** The communication protocol used (e.g., HTTP, HTTPS).
- **Domain Name:** The human-readable address of the server (e.g., www.example.com).
- **Port:** The port number on the server (optional, default is 80 for HTTP and 443 for HTTPS).
- **Path:** The specific location of the resource on the server (e.g., /path/to/resource).
- **Query Parameters:** Additional parameters for the request (optional, e.g., ?query=param).
- **Fragment(Anchor - a):** A specific section within the resource (optional, e.g., #section).

2.1.1 Example of a URL Breakdown

Consider the following URL:

`https://www.example.com:443/path/to/resource?query=param#section`

- **Protocol:** `https` — Specifies the communication protocol (HTTPS).
- **Domain Name:** `www.example.com` — The server's human-readable address.
- **Port:** `443` — The port number used for the connection (default for HTTPS).
- **Path:** `/path/to/resource` — The specific location of the resource on the server.
- **Query Parameters:** `?query=param` — Additional data sent to the server.
- **Fragment:** `#section` — Refers to a specific part within the resource.

This structure allows precise identification and access to resources on the web.

2.2 What is Difference between URL and URI

- **URL (Uniform Resource Locator):** A specific type of URI that provides the means to access a resource on the internet, including the protocol and address.
- **URI (Uniform Resource Identifier):** A broader term that includes both URLs and URNs (Uniform Resource Names). A URI identifies a resource either by location (URL) or by name (URN).
- **URN (Uniform Resource Name):** A specific type of URI that identifies a resource by name in a particular namespace, without providing its location. URNs are used for resources that are not tied to a specific location, such as ISBNs for books.
- In a URL, you should specify the complete path to the resource, similar to how anchors (`<a href="...">`) in HTML link to specific locations.

- A URI acts like a routing concept in React, identifying resources by path or name, enabling navigation and resource management within applications.

2.3 HTTP Request & Response Headers and Body

- **HTTP Request Headers:** Metadata sent by the client to the server, providing information about the request. Common headers include `User-Agent`, `Accept`, and `Authorization`.
- **HTTP Response Headers:** Metadata sent by the server to the client, providing information about the response. Common headers include `Content-Type`, `Content-Length`, and `Set-Cookie`.

Header Name	Description
Content-Type	The media type of the resource (e.g., text/html)
Content-Length	The size of the response body in bytes
Set-Cookie	Used to send cookies from the server to the client

- **HTTP Request Body:** The data sent by the client to the server in an HTTP request. This is typically used with POST and PUT requests to submit form data or upload files.
- **HTTP Response Body:** The data sent by the server to the client in an HTTP response. This is typically the HTML content, JSON data, or other resources requested by the client.

---

3 HTTP Methods & HTTP Status Codes

3.1 HTTP Methods

- **GET:** Retrieve data from the server (e.g., a web page).
  - **GET:** The only HTTP method you can use directly in a browser to view and retrieve content from a server. Browsers send GET requests when you enter a URL or click a link, allowing you to see the response (such as a web page) in the browser window.
- **POST:** Submit data to the server (e.g., form data).
- **PUT:** Update existing data on the server.
- **DELETE:** Remove data from the server.

3.2 HTTP Status Codes

HTTP status codes are standardized response codes sent by web servers to indicate the result of an HTTP request. These codes are grouped into five categories:

3.2.1 2xx - Success

Status codes in the 2xx range indicate that the client's request was successfully received, understood, and accepted.

- **200 OK:** The request succeeded and the response contains the requested data
- **201 Created:** The request succeeded and a new resource was created

- **204 No Content:** The request succeeded but there is no content to return

3.2.2 3xx - Redirection

Status codes in the 3xx range indicate that further action needs to be taken to complete the request.

- **301 Moved Permanently:** The requested resource has been permanently moved to a new URL
- **302 Found:** The requested resource temporarily resides at a different URL
- **304 Not Modified:** The resource hasn't changed since the last request (used for caching)

3.2.3 4xx - Client Error

Status codes in the 4xx range indicate that the client made an error in their request.

- **400 Bad Request:** The server cannot process the request due to a client error
- **401 Unauthorized:** Authentication is required and has failed or not been provided
- **403 Forbidden:** The server understood the request but refuses to authorize it
- **404 Not Found:** The requested resource could not be found on the server
- **422 Unprocessable Entity:** The request was well-formed but contains semantic errors

3.2.4 5xx - Server Error

Status codes in the 5xx range indicate that the server failed to fulfill a valid request.

- **500 Internal Server Error:** A generic error occurred on the server
- **502 Bad Gateway:** The server was acting as a gateway and received an invalid response
- **503 Service Unavailable:** The server is temporarily unable to handle the request
- You can see visual representations of these status codes at HTTP Cats

---

4 HTTP Request and Response Messages

HTTP communication consists of two key components: requests sent by clients and responses returned by servers. Understanding these messages is fundamental to web development.

4.1 HTTP Request Structure

An HTTP request is composed of:

1. **Request Line**: Contains the HTTP method, URL, and protocol version

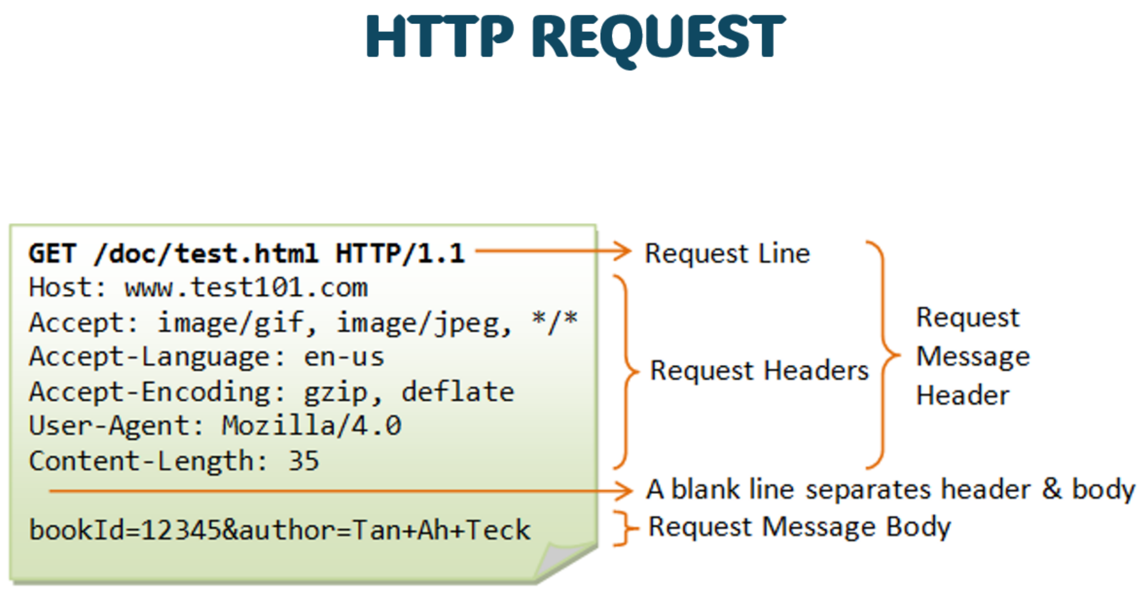
```
GET /index.html HTTP/1.1
```

2. **Request Headers**: Key-value pairs providing metadata about the request

```
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
Authorization: Bearer token123
```

3. **Empty Line**: Separates headers from body
4. **Request Body**: Optional data sent to the server (primarily with POST, PUT methods)

```
{
  "username": "johndoe",
  "email": "john@example.com"
}
```



```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

Figure 2: HTTP Request Structure

4.2 HTTP Response Structure

An HTTP response contains:

1. **Status Line**: Includes protocol version, status code, and reason phrase

```
HTTP/1.1 200 OK
```

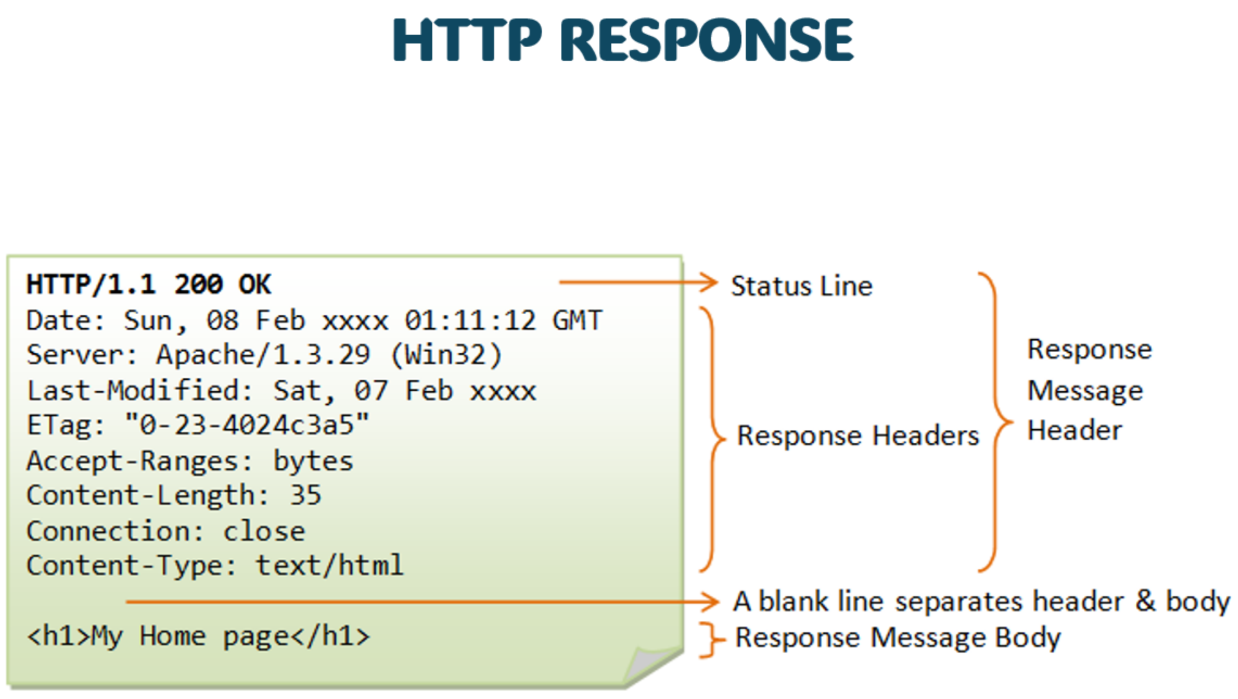
2. **Response Headers**: Key-value pairs with information about the response

```
Content-Type: text/html; charset=UTF-8
Content-Length: 1234
Set-Cookie: session=abc123; Path=/
Cache-Control: max-age=3600
```

3. **Empty Line**: Separates headers from body
4. **Response Body**: Contains the requested resource or error details

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example Page</title>
  </head>
  <body>
```

```
    <h1>Hello World!</h1>
  </body>
</html>
```



```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```

Figure 3: HTTP Response Structure

4.3 Common Request Headers

Header	Description	Example
`Host`	Domain name of the server	`Host: api.example.com`
`User-Agent`	Client application information	`User-Agent: Mozilla/5.0`
`Accept`	Media types the client can process	`Accept: application/json`
`Content-Type`	Format of the data in the request body	`Content-Type: application/json`
`Authorization`	Authentication credentials	`Authorization: Bearer token123`
`Cookie`	Contains stored cookies	`Cookie: sessionId=abc123`

4.4 Common Response Headers

Header	Description	Example
`Content-Type`	Format of the response body	`Content-Type: text/html; charset=UTF-8`
`Content-Length`	Size of the response body in bytes	`Content-Length: 2048`

Header	Description	Example
`Set-Cookie`	Sets a cookie on the client	`Set-Cookie: userId=123; Path=/`
`Cache-Control`	Directives for caching mechanisms	`Cache-Control: max-age=3600`
`Access-Control-Allow-Origin`	CORS permission	`Access-Control-Allow-Origin: *`

4.5 Inspecting HTTP Messages

Developers can examine HTTP requests and responses using:

- Browser Developer Tools (Network tab)
- Tools like Postman or Insomnia
- Command-line utilities like `curl` or `wget`
- HTTP analyzers like Wireshark

Understanding these message structures is essential for debugging web applications, implementing APIs, and ensuring secure communication between clients and servers.

---

5 Intro to Node.js

Node.js is a JavaScript runtime Environment built on Chrome's V8 JavaScript engine that enables developers to execute JavaScript code outside of a web browser. It has revolutionized web development by allowing JavaScript to be used for server-side programming.

5.1 Key Features

- **JavaScript Runtime**: Built on Chrome's V8 JavaScript engine
- **Event-Driven Architecture**: Non-blocking I/O model makes it lightweight and efficient
- **Cross-Platform**: Runs on Windows, macOS, Linux, and other platforms
- **Free and Open Source**: Available under the MIT license
- **Rich Ecosystem**: Vast collection of libraries and packages via npm (Node Package Manager)

5.2 History of Node.js

Node.js was created by Ryan Dahl in 2009 and was first introduced at the European JSConf on November 8, 2009. Dahl developed Node.js out of frustration with the limitations of traditional web server software, particularly Apache HTTP Server's inability to handle concurrent connections efficiently.

5.2.1 Timeline

- **2009**: Initial release by Ryan Dahl
- **2010**: npm (Node Package Manager) was introduced by Isaac Schlueter

- **2011**: LinkedIn became one of the first major companies to adopt Node.js
- **2012**: Node.js reached version 0.8 with improved stability
- **2015**: The Node.js Foundation was formed to oversee the project's development
- **2015**: io.js (a fork of Node.js) merged back into Node.js
- **2019**: Node.js 12.0 released with improved performance and security features
- **Present**: Node.js continues to evolve with regular updates and widespread adoption

Today, Node.js is widely used by major companies including Netflix, PayPal, Uber, LinkedIn, and NASA, powering everything from web applications to IoT devices.

---

6 Node REPL (Read-Eval-Print Loop)

The Node.js REPL (Read-Eval-Print Loop) provides an interactive programming environment for executing JavaScript code in real-time.

6.1 Key Features

- **Read**: Reads user input (JavaScript code)
- **Eval**: Evaluates the input code
- **Print**: Displays the result
- **Loop**: Repeats the process

6.2 Using the REPL

To start the Node.js REPL:

```
$ node
>
```

Once in the REPL, you can:

- Execute JavaScript expressions directly
- Access all built-in modules and global objects
- Define variables and functions
- Explore object properties with tab completion

6.3 Advanced Features

- **Multi-line editing**: Use the `.editor` command for writing multi-line code blocks
- **Command history**: Navigate through previous commands with up/down arrow keys
- **Tab completion**: Press Tab to autocomplete variables, properties, and methods
- **Special commands**: Type `.help` to see all available commands

6.4 Example Usage

```
$ node
> const greeting = "Hello, Node.js!"
undefined
```

```
> greeting
'Hello, Node.js!'
> greeting.toUpperCase()
'HELLO, NODE.JS!'
> .help
# Shows all available commands
```

The REPL is invaluable for testing code snippets, exploring APIs, and learning Node.js without needing to create and run files.

6.5 `this` keyword in Node.js REPL

The `this` keyword behaves differently in the Node.js REPL environment compared to regular JavaScript files:

- In the REPL, `this` refers to the global object (similar to `global`)
- This gives you direct access to all globally available properties and functions

```
// In Node.js REPL
> this === global
true

> this.setTimeout === setTimeout
true

// Assigning to this creates global properties
> this.message = "Hello from REPL"
'Hello from REPL'
> global.message
'Hello from REPL'
```

This behavior differs from regular Node.js modules, where `this` inside a module refers to that module's exports object, not the global object. The REPL's global binding for `this` makes it convenient for interactive exploration and testing.

---

7 Modules In Node.js

A module in Node.js is a reusable block of code whose functionality can be easily imported into other files. The modular approach helps in organizing code, maintaining separation of concerns, and enabling code reuse.

7.1 Types of Modules in Node.js

Node.js supports three types of modules:

1. **Core Modules**: Built-in modules that come with Node.js installation
2. **Local Modules**: Custom modules created within your application
3. **Third-party Modules**: Modules installed via npm (Node Package Manager)

7.1.1 Core Modules

Core modules are built into Node.js and can be used without installation:

```
// Importing core modules
const fs = require("fs"); // File System operations
const http = require("http"); // HTTP server/client functionality
const path = require("path"); // Path manipulation utilities
const os = require("os"); // Operating system related utilities
```

7.1.2 Local Modules

Custom modules created for specific application needs:

```
// math.js (Export)
function add(a, b) {
  return a + b;
}

module.exports = { add };

// app.js (Import)
const math = require("./math");
console.log(math.add(5, 3)); // Output: 8
```

7.1.3 Third-party Modules

External modules from npm:

```
# Installing a third-party module
npm install express
```

```
// Using the installed module
const express = require("express");
const app = express();
```

7.2 Module Systems in Node.js

Node.js supports two module systems:

7.2.1 CommonJS (Traditional)

The original module system used in Node.js:

```
// Exporting
module.exports = { function1, function2 };

// Importing
const module = require("./module-name");
```

7.2.2 ES Modules (Modern)

ECMAScript modules, supported in newer versions of Node.js:

```
// Exporting
export function function1() {}
export function function2() {}

// Importing
import { function1, function2 } from "./module-name.js";
```

To use ES modules, either:

- Use the `.mjs` extension
- Add `"type": "module"` to your package.json

7.3 Module Caching

Node.js caches modules after the first import, making subsequent requires faster:

```
// First require loads and executes the module
const module1 = require("./myModule");

// Second require returns the cached module without re-executing code
const module2 = require("./myModule");

// Both variables reference the same instance
console.log(module1 === module2); // true
```

7.4 Best Practices

- Keep modules focused on a single responsibility
- Use descriptive naming conventions
- Structure your project with logical module organization
- Consider using named exports for better code clarity
- Document your module's public API

The module system is a fundamental aspect of Node.js that enables developers to build maintainable, scalable applications by breaking code into manageable, reusable components.

---

8 OS Module in Node.js

- It's a core module in Node.js.
- The `os` module provides operating system-related utility methods and properties.
- It can be used to get information about the current operating system, such as the platform, CPU architecture, and memory usage.

8.1 Using node: Prefix in require()

- The code you're looking at demonstrates an alternative way to import Node.js built-in modules:

Why use the node: prefix?

- Module Resolution Clarity: The node: prefix explicitly indicates that you're importing a built-in Node.js module, not a third-party module from node_modules.
- **Preventing Conflicts:** It ensures that you're getting the official Node.js module even if a package with the same name exists in your node_modules folder.
- **Performance:** Node.js can optimize the loading of core modules when using this explicit syntax.
- **Future-proofing:** This syntax was introduced in Node.js v14.18.0 and is the recommended way to import core modules in newer Node.js applications.
- **Security:** Without this prefix, if someone published a malicious package named "os" to npm and you accidentally installed it, your code might use that package instead of the built-in module.
- This is particularly important for security-sensitive applications where you want to ensure you're using official Node.js APIs.

```javascript
const os = require("os"); // OS module

const os = require("node:os"); // Node.js built-in OS module

// Get the operating system platform
console.log(os.platform()); // e.g., "win32", "linux"

// Get the CPU architecture
console.log(os.arch()); // e.g., "x64"

// Get the total system memory
console.log(os.totalmem()); // in bytes

// Get the free system memory
console.log(os.freemem()); // in bytes

// Get the home directory
console.log(os.homedir()); // e.g., "/Users/username"

// Get the system uptime
console.log(os.uptime()); // in seconds

// Get the hostname
console.log(os.hostname());

// Get the system's network interfaces
console.log(os.networkInterfaces());
```

---

9 FS (File System) Module in Node.js

The File System module provides an API for interacting with files and directories.

9.1 Usage

```javascript
const fs = require("node:fs"); // Recommended import
```

9.2 Reading and Writing Files

```javascript
// Read file (async)
fs.readFile("file.txt", "utf8", (err, data) => {
  if (err) throw err;
  console.log(data);
});

// Read file (sync)
const data = fs.readFileSync("file.txt", "utf8");

// Write file (async)
fs.writeFile("output.txt", "Hello World!", (err) => {
  if (err) throw err;
  console.log("File written");
});
```

9.3 Directory Operations

```javascript
// Create directory
fs.mkdir("newDir", (err) => {
  if (err) throw err;
});

// Read directory contents
fs.readdir("myDir", (err, files) => {
  if (err) throw err;
  console.log(files);
});
```

9.4 Streams

```javascript
// Read stream for large files
const readStream = fs.createReadStream("largefile.txt");
readStream.on("data", (chunk) => {
  console.log(`Received ${chunk.length} bytes`);
});

// Write stream
const writeStream = fs.createWriteStream("output.txt");
writeStream.write("Hello world!");
writeStream.end();
```

9.5 Modern Approach (Promises)

```javascript
const fsPromises = require("fs/promises");
```

```
async function example() {
  try {
    const data = await fsPromises.readFile("file.txt", "utf8");
    await fsPromises.writeFile("output.txt", data.toUpperCase());
  } catch (err) {
    console.error(err);
  }
}
```

9.6 Best Practices

- Use async methods to avoid blocking the event loop
- Implement proper error handling
- Use the path module for cross-platform path handling
- Use streams for large files

---

10 HTTP Module in Node.js

The HTTP module in Node.js allows you to create HTTP servers and clients. It is a core module and can be imported using the following syntax:

```
const http = require("node:http"); // Import using recommended node:
↪  prefix
```

10.1 Creating HTTP Servers

```
// Create a basic HTTP server
const server = http.createServer((req, res) => {
  // Request handler function
  res.statusCode = 200;
  res.setHeader("Content-Type", "text/plain");
  res.end("Hello, World!");
});

// Start the server on port 3000
server.listen(3000, () => {
  console.log("Server running at http://localhost:3000/");
});
```

10.2 HTTP Response Methods

10.2.1 res.end()

- The `res.end()` method is a crucial HTTP response method that signals to the server that all response headers and body have been sent
- It indicates that the response is complete and ready to be sent to the client
- It takes an optional argument which can be:
  - `String`: Text content to send as response body
  - `Buffer`: Binary data to send as response body

  - No argument: Only ends the response without sending additional content

```
// Simple text response
res.end("Hello World");

// Buffer response (binary data)
const buffer = Buffer.from("Binary content");
res.end(buffer);

// Empty response (headers only)
res.statusCode = 204; // No Content
res.end();
```

- After `res.end()` is called, any attempt to send more data to the client will result in an error
- Always ensure you call `res.end()` to properly complete the HTTP response cycle

10.3 Making HTTP Requests

```
const http = require("node:http");

const options = {
  hostname: "www.example.com",
  port: 80,
  path: "/",
  method: "GET",
};

const req = http.request(options, (res) => {
  console.log(`Status Code: ${res.statusCode}`);

  let data = "";

  // Collect data chunks
  res.on("data", (chunk) => {
    data += chunk;
  });

  // Process complete response
  res.on("end", () => {
    console.log("Response body:", data);
  });
});

// Handle request errors
req.on("error", (error) => {
  console.error(`Request error: ${error.message}`);
});

// End the request
req.end();
```

10.4 Simplified GET Requests

```
http
  .get("http://www.example.com", (res) => {
    let data = "";

    res.on("data", (chunk) => {
      data += chunk;
    });

    res.on("end", () => {
      console.log("Data received:", data);
    });
  })
  .on("error", (error) => {
    console.error(`Error: ${error.message}`);
  });
```

- In this example, we create an HTTP request to “www.example.com” and log the response status and body.
- We also handle any errors that may occur during the request.
- Finally, we end the request.

10.5 Best Practices

- Always handle errors in your HTTP requests and responses.
- Use environment variables to store sensitive information like API keys.
- Consider using a framework like Express for more complex applications.

---

11 Send Files To Client

The HTTP module in Node.js allows you to send various file types to clients by properly setting the Content-Type header.

12 Send Files To Client

The HTTP module in Node.js allows you to send various file types to clients by properly setting the Content-Type header.

12.1 Steps to send files:

1. Import the required modules (http, fs, path).
2. Create an HTTP server.
3. Read the file using fs.readFile or create a read stream.
4. Set the appropriate Content-Type header.
5. Send the file data in the response.

12.2 Sending Text Files

```
const http = require("node:http");
const fs = require("node:fs");

const server = http.createServer((req, res) => {
  fs.readFile("example.txt", (err, data) => {
    if (err) {
      res.statusCode = 500;
      res.end("Error loading file");
      return;
    }

    res.statusCode = 200;
    res.setHeader("Content-Type", "text/plain");
    res.end(data);
  });
});

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

12.3 Sending HTML Files

```
const http = require("node:http");
const fs = require("node:fs");

const server = http.createServer((req, res) => {
  fs.readFile("index.html", (err, data) => {
    if (err) {
      res.statusCode = 500;
      res.end("Error loading file");
      return;
    }

    res.statusCode = 200;
    res.setHeader("Content-Type", "text/html");
    res.end(data);
  });
});

server.listen(3000);
```

12.4 Sending Images

```
const http = require("node:http");
const fs = require("node:fs");
const path = require("node:path");

const server = http.createServer((req, res) => {
  const imagePath = path.join(__dirname, "images", "example.jpg");

  fs.readFile(imagePath, (err, data) => {
    if (err) {
      res.statusCode = 500;
      res.end("Error loading image");
      return;
    }

    // Set Content-Type based on image format
    const extension = path.extname(imagePath).toLowerCase();
    let contentType = "image/jpeg"; // default

    if (extension === ".png") {
      contentType = "image/png";
    } else if (extension === ".gif") {
      contentType = "image/gif";
    } else if (extension === ".svg") {
      contentType = "image/svg+xml";
    } else if (extension === ".webp") {
      contentType = "image/webp";
    }

    res.statusCode = 200;
    res.setHeader("Content-Type", contentType);
    res.end(data);
  });
});

server.listen(3000);
```

12.5 Using Streams for Large Files

For better performance with large files, use streams instead of loading the entire file into memory:

```
const http = require("node:http");
const fs = require("node:fs");
const path = require("node:path");

const server = http.createServer((req, res) => {
  const filePath = path.join(__dirname, "large-image.jpg");

  // Check if file exists
  fs.stat(filePath, (err, stat) => {
    if (err) {
      res.statusCode = 404;
      res.end("File not found");
      return;
    }

    res.statusCode = 200;
    res.setHeader("Content-Type", "image/jpeg");
    res.setHeader("Content-Length", stat.size);

    // Create read stream and pipe to response
    const stream = fs.createReadStream(filePath);
    stream.pipe(res);

    // Handle stream errors
    stream.on("error", () => {
      res.statusCode = 500;
      res.end("Server error");
    });
  });
});

server.listen(3000);
```

12.6 Content-Type for Common File Types

File Type	Content-Type Header
.html	text/html
.css	text/css
.js	application/javascript
.json	application/json
.png	image/png
.jpg/.jpeg	image/jpeg
.gif	image/gif
.svg	image/svg+xml
.pdf	application/pdf
.mp4	video/mp4
.mp3	audio/mpeg

---

13 First API

An API (Application Programming Interface) allows different applications to communicate with each other.

13.1 What is an API?

- An API is a set of rules that define how applications or devices can connect to and communicate with each other
- APIs act as messengers that deliver your request to a system and return the response back to you

13.2 Creating a Simple API with Node.js

Here's how to create a basic API using Node.js:

```
const http = require("node:http");

// Sample data
const users = [
  { id: 1, name: "John", age: 25 },
  { id: 2, name: "Jane", age: 24 },
];

const server = http.createServer((req, res) => {
  // Set response headers
  res.setHeader("Content-Type", "application/json"); // Set Content-Type to
↪   JSON

  // Handle different routes
  if (req.url === "/api/users" && req.method === "GET") {
    // Return all users
    res.statusCode = 200;
    res.end(JSON.stringify(users));
  } else if (req.url.match(/\/api\/users\/([0-9]+)/) && req.method ===
↪   "GET") {
    // Get user by ID
    // Extract ID from URL
    const id = req.url.split("/")[3];
    const user = users.find((u) => u.id === parseInt(id));

    if (user) {
      res.statusCode = 200;
      res.end(JSON.stringify(user)); // User found
    } else {
      res.statusCode = 404;
      res.end(JSON.stringify({ message: "User not found" }));
    }
  } else {
    // Handle 404
    res.statusCode = 404;
    res.end(JSON.stringify({ message: "Route not found" }));
  }
});

const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

13.3 Testing Your API

You can test your API using:

1. A web browser (for GET requests)
2. Tools like Postman or Insomnia
3. Command line with curl:

```
# Get all users
curl http://localhost:3000/api/users

# Get specific user
curl http://localhost:3000/api/users/1
```

13.4 API Response Format

Most modern APIs return data in JSON format:

```
{
  "id": 1,
  "name": "John",
  "age": 25
}
```

- This simple API example demonstrates how to handle different routes and return appropriate responses with correct status codes.

---

14 Routing

Routing determines how an application handles client requests to different endpoints. In Node.js, we can implement basic routing using the HTTP module by examining the request URL and method.

14.1 Simple Routing Implementation

```
const http = require("node:http");

const server = http.createServer((req, res) => {
  // Set default headers
  res.setHeader("Content-Type", "application/json");

  // Parse the URL and method
  const url = req.url; // Get the request URL
  const method = req.method; // Get the request method

  // Handle routes
  if (url === "/api/users" && method === "GET") {
    // Get all users
    res.statusCode = 200;
    res.end(JSON.stringify({ message: "Getting all users" })); // Respond
↪   with all users
  } else if (url === "/api/users" && method === "POST") {
    // Create new user
    res.statusCode = 201;
    res.end(JSON.stringify({ message: "User created" }));
  } else if (url.match(/\/api\/users\/\d+/) && method === "GET") {
    // Get user by ID
    const id = url.split("/")[3];
    res.statusCode = 200;
    res.end(JSON.stringify({ message: `Getting user with ID ${id}` }));
  } else {
    // Route not found
    res.statusCode = 404;
    res.end(JSON.stringify({ message: "Route not found" }));
  }
});

server.listen(3000, () => {
  console.log("Server running on port 3000");
});
```

14.2 Route Structure

A well-designed routing system should:

1. **Be organized**: Group related endpoints logically
2. **Be descriptive**: Use clear URL paths that describe the resource
3. **Follow REST conventions**: Use appropriate HTTP methods for operations

14.3 Common Routing Patterns

HTTP Method	URL Pattern	Purpose
GET	/api/resources	Retrieve all resources
GET	/api/resources/:id	Retrieve a specific resource
POST	/api/resources	Create a new resource
PUT/PATCH	/api/resources/:id	Update a specific resource
DELETE	/api/resources/:id	Delete a specific resource

For complex applications, consider using frameworks like Express.js that provide more sophisticated routing capabilities.

---

15 Third Party Modules (NPM)

NPM (Node Package Manager) provides access to thousands of third-party modules that extend Node.js functionality. These packages save development time by providing pre-built solutions for common tasks.

15.1 Popular NPM Packages

Package	Description	Use Case
Express	Fast, minimalist web framework	Building web servers and APIs
Axios	Promise-based HTTP client	Making API requests
Lodash	Utility library with helper functions	Data manipulation
Mongoose	MongoDB object modeling tool	Database integration
Jest	JavaScript testing framework	Unit and integration testing
Moment	Date parsing and manipulation library	Working with dates and times
Nodemon	Development utility for auto-restarting servers	Development workflow
Debug	Lightweight debugging utility	Application logging

15.2 Using Third-Party Modules

15.2.1 1. Initialize your project

```
npm init -y
```

15.2.2 2. Install packages

```
npm install express mongoose axios
```

15.2.3 3. Import in your code

```
// CommonJS syntax
const express = require("express");
const mongoose = require("mongoose");

// ES Modules syntax (with proper configuration)
import express from "express";
import mongoose from "mongoose";
```

15.2.4 4. Use according to documentation

```
const app = express();
app.get("/", (req, res) => {
  res.send("Hello World!");
});
app.listen(3000);
```

15.3 Nodemon for Development

Nodemon monitors your project files and automatically restarts the server when changes are detected, improving development workflow.

15.3.1 Installation

```
# Install globally
npm install -g nodemon
```

```
# Or as a dev dependency in your project
npm install --save-dev nodemon
```

15.3.2 Usage

```
# Instead of: node server.js
nodemon server.js

# With specific file extensions to watch
nodemon --ext js,json,env server.js
```

15.3.3 Configuration

Create a `nodemon.json` file for custom settings:

```
{
  "watch": ["src"],
  "ext": "js,json",
  "ignore": ["node_modules", "logs"],
  "env": {
    "NODE_ENV": "development"
  }
}
```

- Third-party modules dramatically increase productivity and provide battle-tested solutions for common development challenges.

16 .gitignore for Node.js Projects

To ignore the `node_modules` folder in your Git repository, create a `.gitignore` file in your project root directory and add the following content:

```
# Dependencies
node_modules/
npm-debug.log
yarn-error.log
yarn-debug.log
.pnpm-debug.log

# Environment variables
.env
.env.local
.env.development.local
.env.test.local
.env.production.local

# Build outputs
dist/
build/
out/
```

```
# Coverage directory used by testing tools
coverage/

# IDE and editor folders
.idea/
.vscode/
*.sublime-project
*.sublime-workspace

# Operating System Files
.DS_Store
Thumbs.db
```

- This configuration ensures that dependency directories and other unnecessary files are excluded from your Git repository, keeping it clean and efficient.

---

17 Scripts in Package.json

The `scripts` section in a Node.js project's `package.json` file provides a powerful way to automate common development tasks and standardize workflows across your team.

17.1 Basic Structure

```json
{
  "name": "my-project",
  "version": "1.0.0",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js",
    "test": "jest",
    "build": "webpack"
  }
}
```

17.2 Common Script Commands

Script Name	Common Usage	Example
`start`	Launch the production application	`"start": "node index.js"`
`dev`	Run with development settings	`"dev": "nodemon index.js"`
`test`	Execute test suite	`"test": "jest"`
`build`	Compile/bundle for production	`"build": "webpack --mode=production"`
`lint`	Check code quality	`"lint": "eslint src/**/*.js"`
`format`	Format code according to style guide	`"format": "prettier --write 'src/**/*.js'"`

17.3 Running Scripts

```
# Run the "start" script
npm start

# Run other scripts
npm run dev
npm run test
npm run build
```

17.4 Advanced Usage

17.4.1 Combining Scripts

```
"scripts": {
  "clean": "rimraf dist",
  "build": "npm run clean && webpack",
  "serve": "http-server dist",
  "preview": "npm run build && npm run serve"
}
```

Scripts can be chained together using `&&` to create more complex workflows:

```
"scripts": {
  "lint": "eslint .",
  "test": "jest",
  "validate": "npm run lint && npm run test",
  "prepublish": "npm run validate && npm run build"
}
```

17.4.2 Environment Variables

Set different environments for your scripts:

```
"scripts": {
  "start": "NODE_ENV=production node server.js",
  "dev": "NODE_ENV=development nodemon server.js"
}
```

For cross-platform compatibility, use:

```
"scripts": {
  "dev": "cross-env NODE_ENV=development nodemon server.js"
}
```

17.4.3 Custom Parameters

Pass additional arguments to scripts:

```
npm run test -- --watch
```

In this command, everything after `--` gets passed to the underlying command.

17.4.4 Pre and Post Hooks

Automatically run scripts before or after specified commands:

```
"scripts": {
  "pretest": "npm run lint",
  "test": "jest",
  "posttest": "npm run coverage"
}
```

Running `npm test` automatically executes all three scripts in sequence.

Well-designed scripts can significantly improve developer productivity and ensure consistent build and deployment processes across your team.

---

18 How To Create and Publish Your First NPM Package

Creating and publishing your own NPM package allows you to share your code with the JavaScript community and contribute to the ecosystem of reusable modules.

18.1 Prerequisites

- Node.js and npm installed on your system
- A free npm account (register at npmjs.com)

18.2 Step-by-Step Guide

18.2.1 1. Create Your Package Structure

```
# Create a new directory for your package
mkdir my-awesome-package
cd my-awesome-package

# Initialize a new npm package
npm init
```

Answer the interactive prompts to generate your `package.json` file. Pay special attention to:

- **package name**: Must be unique on npm registry
- **version**: Follow semantic versioning (e.g., 1.0.0)
- **entry point**: Usually `index.js`
- **license**: Choose an appropriate license (e.g., MIT)

18.2.2 2. Create Your Package Code

Create the main file specified in your package.json (usually `index.js`):

```
// index.js
function greet(name) {
  return `Hello, ${name}!`;
}

module.exports = {
  greet,
};
```

18.2.3 3. Add Documentation

Create a README.md file with clear documentation:

~~~
# My Awesome Package

A simple greeting utility for Node.js applications.

## Installation

```bash
npm install my-awesome-package
```
~~~

18.3 Usage

```
const { greet } = require("my-awesome-package");

console.log(greet("World")); // Outputs: Hello, World!
```

18.4 License

MIT

18.4.1 4. Test Your Package Locally

```
# Create a test directory
mkdir test-package
cd test-package

# Link your package for local testing
npm link ../my-awesome-package

# Create a test script
echo "const { greet } = require('my-awesome-package');
↪  console.log(greet('Local Tester'));" > test.js

# Run the test
node test.js
```

18.4.2 5. Login to npm

```
npm login
```

Enter your npm username, password, and email address.

18.4.3 6. Publish Your Package

```
# From your package directory
cd ../my-awesome-package
npm publish
```

18.4.4 7. Updating Your Package

1. Make your changes
2. Update the version in `package.json` following semantic versioning:
   - Patch release (bug fixes): `npm version patch`
   - Minor release (new features): `npm version minor`
   - Major release (breaking changes): `npm version major`
3. Publish the update: `npm publish`

18.5 Best Practices

- **Versioning**: Follow semantic versioning (SemVer)
- **Testing**: Include tests for your package functionality
- **Documentation**: Provide clear usage examples
- **Scope**: Consider using scoped packages (@username/package-name) for personal projects
- **Keywords**: Add relevant keywords to help others discover your package
- **License**: Always include a license

18.6 Additional Configuration Options

18.6.1 Private Packages

For private packages (requires npm paid account):

```
# In package.json
{
  "name": "my-private-package",
  "private": true
  // ...
}
```

18.6.2 Package Files

Control which files get published:

```
// In package.json
{
  "files": ["index.js", "lib/", "README.md"]
}
```

- Publishing your own npm packages not only contributes to the community but also helps you organize and reuse your own code across projects efficiently.