Session 12, 13 & 14:

• Introduction to Node.js

• Node modules

• Selectors

• Syntax

• Developing node.js web application

• Event-driven I/O server-side JavaScript

NodeJS is a runtime environment for executing **JavaScript** outside the **browser**, built on the **V8 JavaScript engine**. It enables **server-side development**, supports **asynchronous**, **event-driven programming**, and efficiently handles scalable network applications.

- NodeJS is **single-threaded**, utilizing an **event loop** to handle multiple tasks concurrently.
- It is **asynchronous** and **non-blocking**, meaning operations do not wait for execution to complete.
- The **V8 engine** compiles **JavaScript** to machine code, making **NodeJS** fast and efficient.

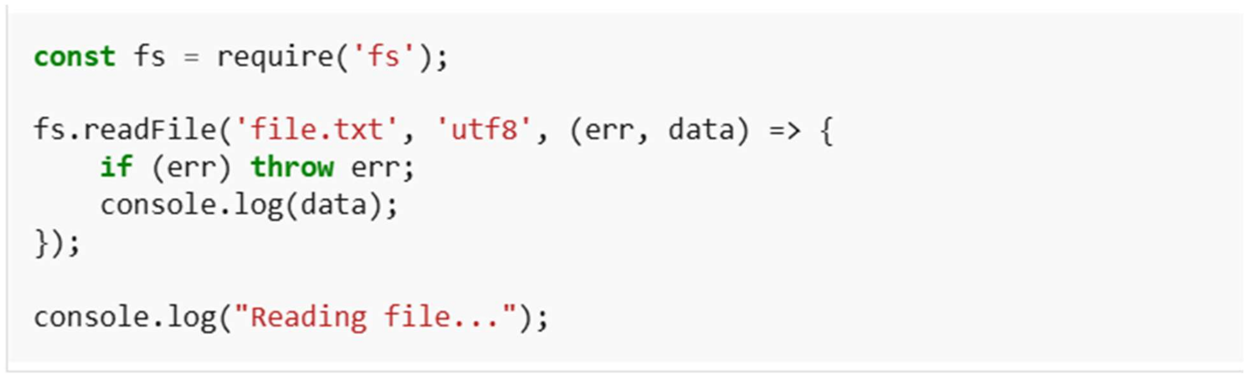
Key Features of NodeJS

- **Server-Side JavaScript:** NodeJS allows JavaScript to run outside the browser, enabling backend development.
- **Asynchronous & Non-Blocking:** Uses an event-driven architecture to handle multiple requests without waiting, improving performance.
- **Single-Threaded Event Loop:** Efficiently manages concurrent tasks using a single thread, avoiding thread overhead.
- **Fast Execution:** Powered by the V8 JavaScript Engine, NodeJS compiles code directly to machine code for faster execution.
- **Scalable & Lightweight:** Ideal for building microservices and handling high-traffic applications efficiently.
- **Rich NPM Ecosystem:** Access to thousands of open-source libraries through Node Package Manager (NPM) for faster development.

Single-Threaded Event Loop Model

NodeJS operates on a single thread but efficiently handles multiple concurrent requests using an event loop.

- **Client Sends a Request:** The request can be for data retrieval, file access, or database queries.
- **NodeJS Places the Request in the Event Loop:** If the request is non-blocking (e.g., database fetch), it is sent to a worker thread without blocking execution.
- **Asynchronous Operations Continue in Background:** While waiting for a response, NodeJS processes other tasks.
- **Callback Execution:** Once the operation completes, the callback function executes, and the response is sent back to the client.



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

fs.readFile('file.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
});

console.log("Reading file...");
```

Applications of NodeJS

Web Development: NodeJS powers backend services for web applications, handling HTTP requests and managing APIs efficiently.

- **Real-Time Applications:** Used in chat applications, **online gaming, and live streaming services** due to its **event-driven, non-blocking architecture.**
- **Server-Side Applications:** Enables full-stack JavaScript development, handling database operations, **authentication, and server logic.**
- **Microservices Architecture:** Helps build scalable, independent microservices for modern web applications.
- **API Development:** Ideal for creating RESTful and GraphQL APIs that interact with databases and client applications.
- **IoT Applications:** NodeJS efficiently handles real-time data processing for IoT devices like sensors and smart home systems.

Limitations of NodeJS

Security Risks – Being open-source and widely used, NodeJS applications are prone to security vulnerabilities like Cross-Site Scripting (XSS) and SQL Injection if not handled properly.

- **Single-Threaded Limitations:** While the event loop manages concurrency efficiently, CPU-intensive tasks can block the thread, affecting performance.
- **Performance Issues with Heavy Computation:** NodeJS is not ideal for CPU-bound tasks like machine learning or video processing, as it lacks multi-threading for heavy computations.
- **Callback Hell:** Older asynchronous code heavily relied on nested callbacks, making it difficult to read and maintain, though Promises and Async/Await help mitigate this.
- **Weak Type Checking:** Since NodeJS runs JavaScript, dynamic typing can lead to runtime errors and unpredictable behavior without strict type enforcement.

Node.js Modules

What are User-Defined Modules?

Modules are reusable blocks of code that can be exported from one file and imported into another.

Helps in organizing code, improving maintainability, and avoiding repetition.

Node.js uses CommonJS modules by default (require & module.exports).

Key Concepts

1. module.exports → Exports a module for use in other files.

2. require() → Imports an external module.

3. Types of Exports:

Single Export (Function, Object, Variable)

Multiple Exports (Using an Object)

Best Practices

1. Keep modules small & focused (Single Responsibility Principle).

2. Use descriptive names (userService.js, authUtils.js).

3. Prefer module.exports for clarity (Avoid mixing with exports).

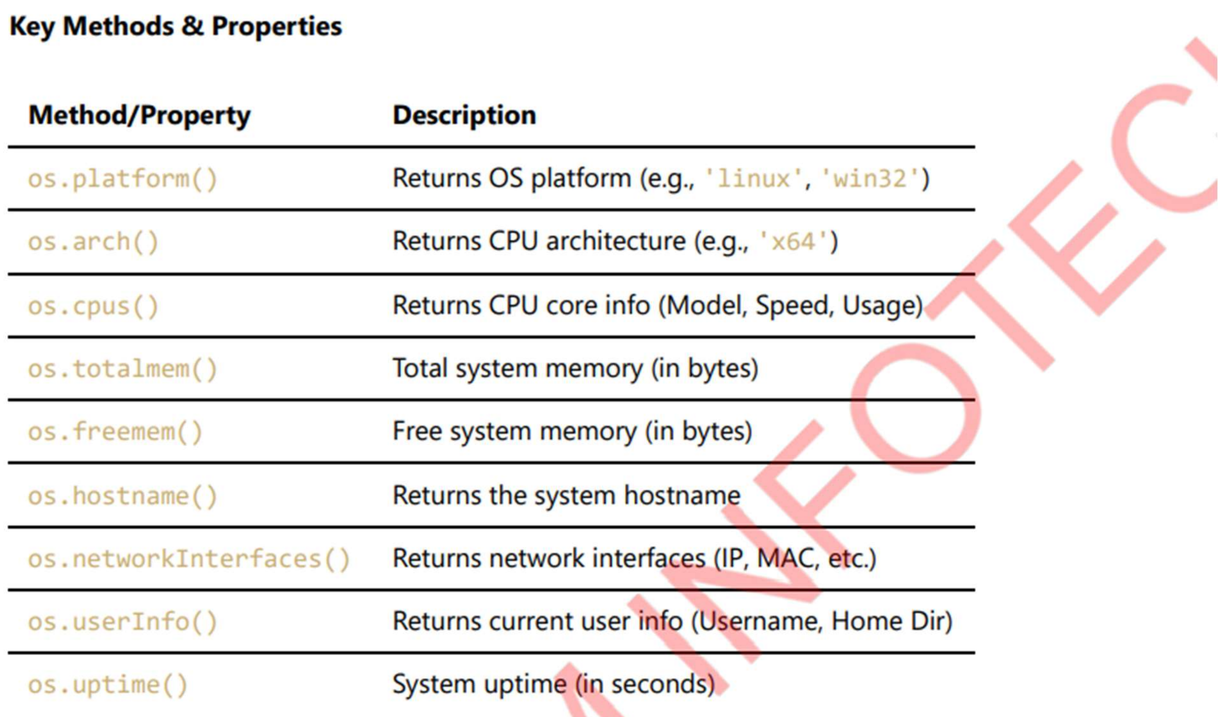
4. Use ES Modules (import/export) if using modern Node.js (v12+).

Some in Built Node Modules

Node.js "os" Module

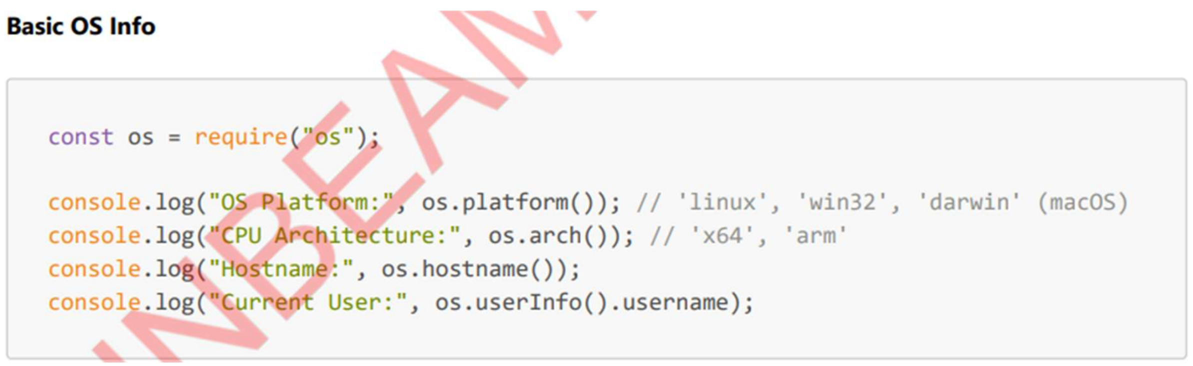
What is the os Module?

A built-in Node.js module that provides utilities for interacting with the operating system.



Key Methods & Properties

Method/Property	Description
`os.platform()`	Returns OS platform (e.g., `'linux'`, `'win32'`)
`os.arch()`	Returns CPU architecture (e.g., `'x64'`)
`os.cpus()`	Returns CPU core info (Model, Speed, Usage)
`os.totalmem()`	Total system memory (in bytes)
`os.freemem()`	Free system memory (in bytes)
`os.hostname()`	Returns the system hostname
`os.networkInterfaces()`	Returns network interfaces (IP, MAC, etc.)
`os.userInfo()`	Returns current user info (Username, Home Dir)
`os.uptime()`	System uptime (in seconds)



Basic OS Info

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

console.log("OS Platform:", os.platform()); // 'linux', 'win32', 'darwin' (macOS)
console.log("CPU Architecture:", os.arch()); // 'x64', 'arm'
console.log("Hostname:", os.hostname());
console.log("Current User:", os.userInfo().username);
```

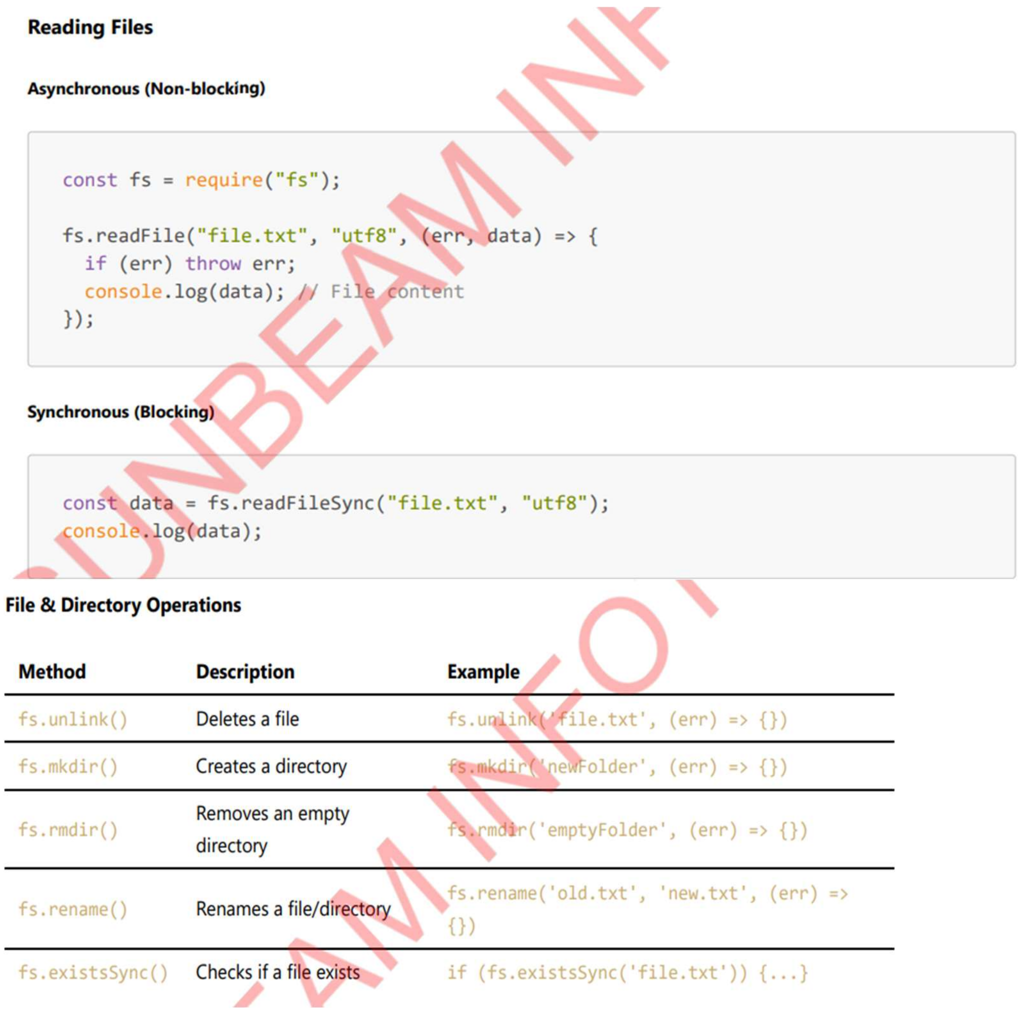
Node.js "fs" Module

What is the fs Module?

A built-in Node.js module for interacting with the file system.Supports synchronous (blocking) and asynchronous (non-blocking) operations.

Used for:

- **Reading/Writing files**
- **Creating/Deleting files & directories**
- **File permissions & stat**



Reading Files

Asynchronous (Non-blocking)

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

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

Synchronous (Blocking)

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

File & Directory Operations

Method	Description	Example
`fs.unlink()`	Deletes a file	`fs.unlink('file.txt', (err) => {})`
`fs.mkdir()`	Creates a directory	`fs.mkdir('newFolder', (err) => {})`
`fs.rmdir()`	Removes an empty directory	`fs.rmdir('emptyFolder', (err) => {})`
`fs.rename()`	Renames a file/directory	`fs.rename('old.txt', 'new.txt', (err) => {})`
`fs.existsSync()`	Checks if a file exists	`if (fs.existsSync('file.txt')) {...}`

Best Practices

- **Prefer async methods (Avoid blocking the event loop).**
- **Use fs.promises for Promise-based operations (Modern approach).**
- **Handle errors properly (Avoid crashes).**
- **Use path.join() for cross-platform paths (Avoid \ vs / issues).**

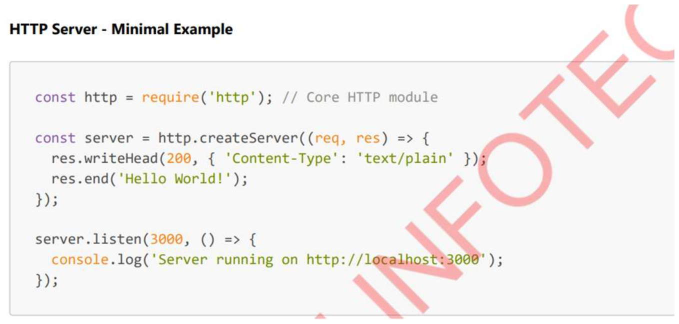
HTTP Server

- **Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).**

- **The HTTP module can create an HTTP server that listens to server ports and gives a response back tothe client.**

Key Components

- **http.createServer() - Creates server instance**
- **req (Request) - Incoming message object**
- **res (Response) - Server response object**
- **server.listen() - Starts server on specified port**



HTTP Server - Minimal Example

```
const http = require('http'); // Core HTTP module

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World!');
});

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

Callback Hell

What is Callback Hell?

Callback Hell (a.k.a. "Pyramid of Doom") occurs when multiple nested callbacks make code:

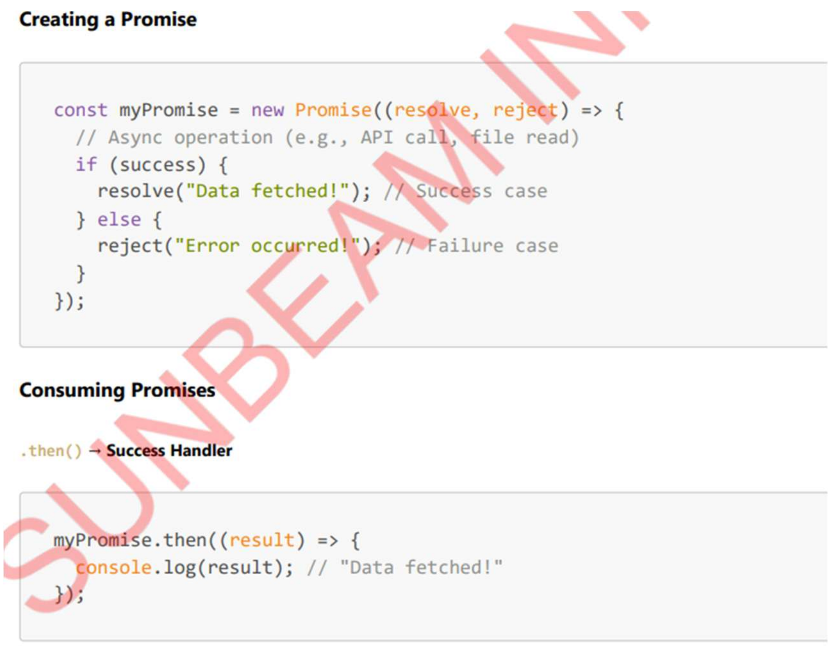
- Hard to read (Deep indentation)
- Difficult to maintain
- Prone to errors (Error handling becomes messy)

JS Promises

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation.

Key States of a Promise

- **pending → Initial state (not yet resolved/rejected)**
- **fulfilled → Operation completed successfully**
- **rejected → Operation failed**



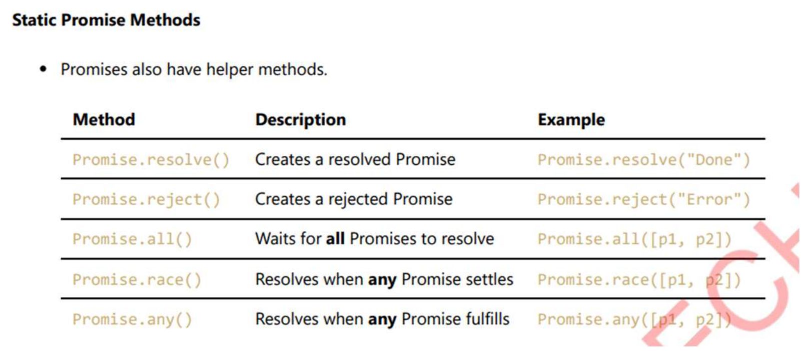
Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {
  // Async operation (e.g., API call, file read)
  if (success) {
    resolve("Data fetched!"); // Success case
  } else {
    reject("Error occurred!"); // Failure case
  }
});
```

Consuming Promises

`.then()` → **Success Handler**

```
myPromise.then((result) => {
  console.log(result); // "Data fetched!"
});
```



Static Promise Methods

- Promises also have helper methods.

Method	Description	Example
`Promise.resolve()`	Creates a resolved Promise	`Promise.resolve("Done")`
`Promise.reject()`	Creates a rejected Promise	`Promise.reject("Error")`
`Promise.all()`	Waits for **all** Promises to resolve	`Promise.all([p1, p2])`
`Promise.race()`	Resolves when **any** Promise settles	`Promise.race([p1, p2])`
`Promise.any()`	Resolves when **any** Promise fulfills	`Promise.any([p1, p2])`

Promise.resolve()

Returns a Promise object that is resolved with the given value (data).

Promise.reject()

Returns a new Promise object that is rejected with the given reason.

Promise.all()

Takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises fulfill (including when an empty iterable is passed), with an array of the fulfillment values. It rejects when any of the input's promises reject, with this first rejection reason

Promise.any()
Takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when any of the input's promises fulfill, with this first fulfillment value. It rejects when all of the input's promises

reject (including when an empty iterable is passed), with an AggregateError containing an array of rejection reasons

Promise.allSettled()
Takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises settle (including when an empty iterable is passed), with an array of objects that describe the outcome of each promise.

Promise use cases

- API calls (fetch, Axios).
- File operations (fs.promises)
- . 3. Database queries (Mongoose, Sequelize).

Async/Await in JavaScript

What is Async/Await?

Async/await is syntactic sugar built on top of Promises that:

- Makes asynchronous code look synchronous
- Eliminates .then() chains
- Uses try/catch for error handling
- Requires the async keyword before functions

Key Characteristics

- async functions always return Promises
- await pauses execution until Promise settles
- Works with any Promise-based API
- Errors are caught with try/catch

Key Improvements with Async/Await

1. Flatter Structure

- No more .then() chains
- Sequential code flow

2. Natural Error Handling

- Uses standard try/catch

3. Variable Scope

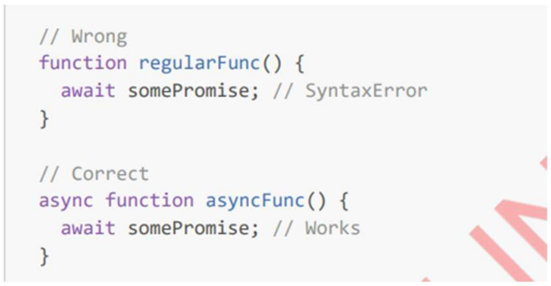
- Variables available in same scope (no nested closures)

4. Readability

- Looks like synchronous code

Important Rules

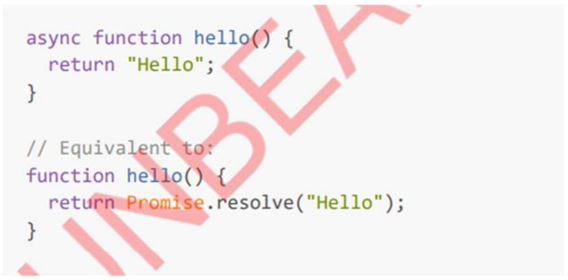
Rule 1: Await Only Works in Async Functions



```javascript
// Wrong
function regularFunc() {
  await somePromise; // SyntaxError
}

// Correct
async function asyncFunc() {
  await somePromise; // Works
}
```

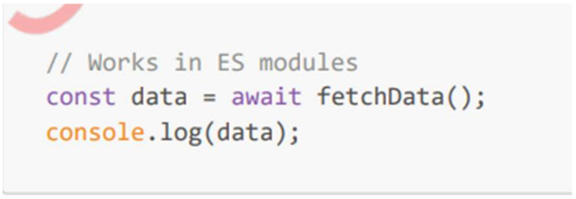
Rule 2: Async Functions Return Promises



```javascript
async function hello() {
  return "Hello";
}

// Equivalent to:
function hello() {
  return Promise.resolve("Hello");
}
```

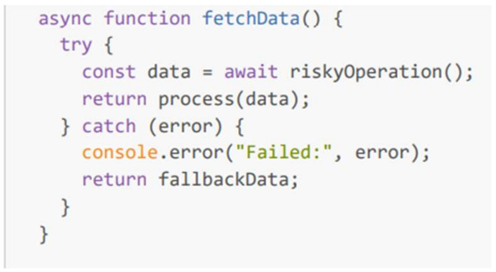
Rule 3: Top-Level Await (Modern JS v14.8.0+)



```javascript
// Works in ES modules
const data = await fetchData();
console.log(data);
```

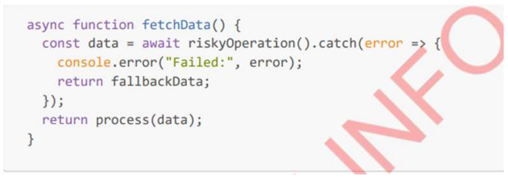
Error Handling

Method 1: Try/Catch

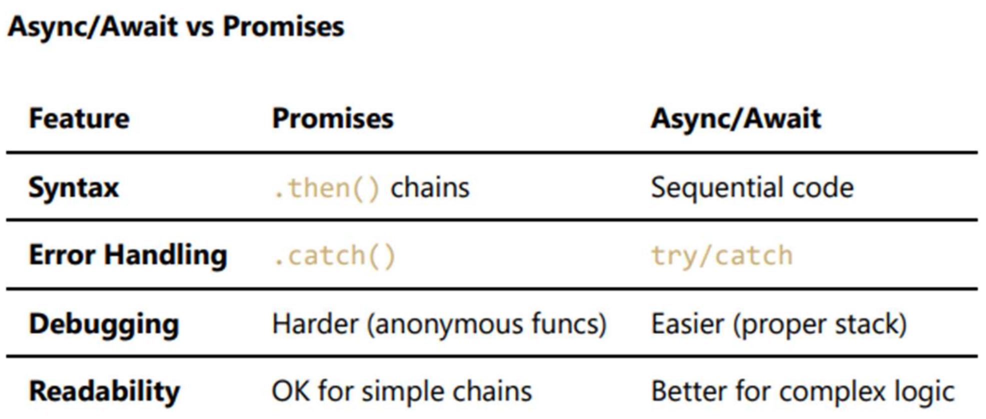


```
async function fetchData() {
  try {
    const data = await riskyOperation();
    return process(data);
  } catch (error) {
    console.error("Failed:", error);
    return fallbackData;
  }
}
```

Method 2: Promise.catch()



```
async function fetchData() {
  const data = await riskyOperation().catch(error => {
    console.error("Failed:", error);
    return fallbackData;
  });
  return process(data);
}
```



Async/Await vs Promises

Feature	Promises	Async/Await
Syntax	`.then()` chains	Sequential code
Error Handling	`.catch()`	`try/catch`
Debugging	Harder (anonymous funcs)	Easier (proper stack)
Readability	OK for simple chains	Better for complex logic

When to Use Async/Await vs Promises

Async/Await

- Most asynchronous code
- Complex promise chains
- Error-heavy operations
- Anywhere you want synchronous-looking code

Raw Promises

- You need Promise.all()/Promise.race()
- Working with existing promise-based libraries
- Performance-critical parallel operations

Node JS Concepts & Internals

Browser Architecture

- Browser: Specialized application for website browsing with parsers (HTML, CSS, XML) to render UI.

Key Components:

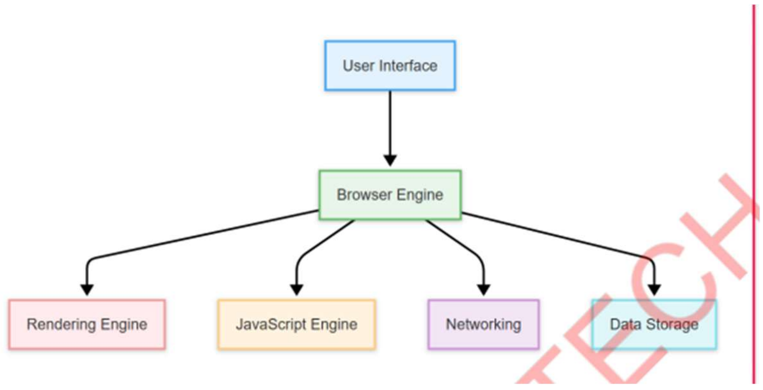
- User Interface: Browser user interface (menus, tabs, etc).
- Browser Engine: Bridge between User interface and Browser core functionality.
- Network: Handles HTTP requests/responses.
- Data Storage: Manage persistent storage like local storage.
- Rendering Engine: Renders HTML/CSS components.
- JavaScript Engine: Executes JS code.

Java Script Engine:

A JavaScript engine is a program or software component that executes JavaScript code. It translates human-readable JavaScript code into machine code that a computer can understand and execute. Modern JavaScript engines use just-in-time (JIT) compilation to improve performance.

Popular JavaScript engines include:

- V8: Developed by Google, used in Chrome and Node.js.
- SpiderMonkey: Developed by Mozilla, used in Firefox. It was the first JavaScript
- engine.
- JavaScriptCore: Used in Safari.
- Chakra: Used in Internet Explorer and Microsoft Edge (legacy).



V8 Engine

Description: Google's open-source, high-performance JS/Wasm engine written in C++.

Used In: Chrome and Node.js.

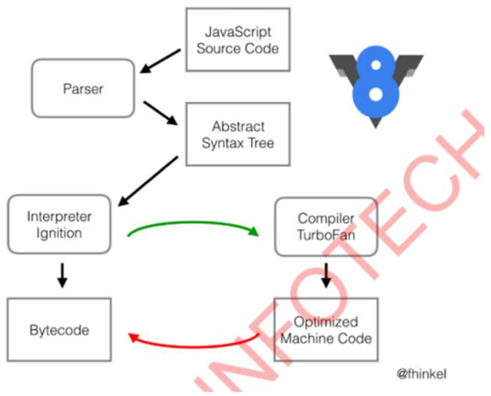
Workflow:

1. Parser: Converts JS source code to an Abstract Syntax Tree (AST).

2. Interpreter (Ignition): Generates bytecode.

3. Compiler (TurboFan): Optimizes bytecode to machine code.

Key Role: Powers Node.js's JavaScript runtime



Node.js Overview

Definition: JavaScript runtime built on V8 for server-side and networking apps.

Developed By: Ryan Dahl (2009).

Cross-Platform: Runs on macOS, Linux, Windows.

Core Features:

- Event-driven, non-blocking I/O model.
- Single-threaded but scalable.
- Lightweight and efficient.

Node.js Architecture

Key Components:

V8 Engine: Executes JS code.

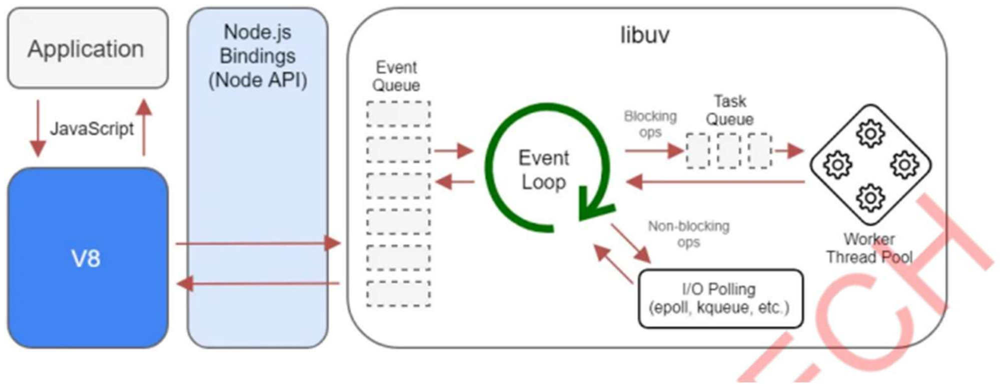
libuv: Handles event loop and async I/O (uses OS async APIs like epoll, kqueue).

Thread Pool: Used for:

- Crypto operations.
- DNS lookups.
- File I/O (when no OS async API exists).

Standard Library: Provides modules for:

Filesystem (fs), HTTP, TCP/UDP, Buffers, Crypto



Node JS Misconceptions

Single-Threaded?:

Yes: Event loop and V8 run on a single thread (main call stack).

But: Thread pool (via libuv) handles blocking operations.

Thread Pool Usage:

Not for all I/O! Only when no OS async API is available.

Used for CPU-bound tasks (e.g., crypto.pbkdf2).

Event Loop vs. V8 Threads:

Event loop runs in one thread; V8 may use additional threads for optimization

Event Loop & Non-Blocking I/O

- Event-Driven Model:

Event Queue: Holds pending callbacks.

Event Loop: Continuously checks and executes callbacks from phases:

1. Timers: setTimeout, setInterval.

2. I/O Callbacks: Network/File operations.

3. Idle/Prepare: This phase is internal to Node.js and prepares for the Poll phase.

4. Poll Phase: Waits for new I/O events.

5. Check Phase: setImmediate callbacks.

6. Close Phase: Cleanup (e.g., socket.on('close')).

Non-Blocking I/O: Achieved via libuv's async OS APIs (e.g., epoll on Linux).

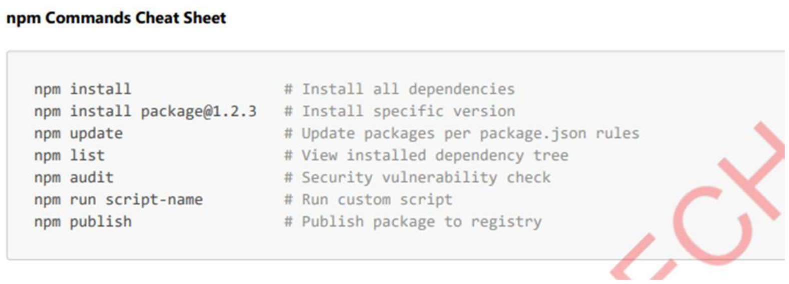
npm package manager

What is npm?

Node Package Manager: World's largest software registry for JavaScript packages.

Key Functions:

- Install/uninstall packages
- Manage dependencies
- Run project scripts


npm Commands Cheat Sheet

```
npm install                # Install all dependencies
npm install package@1.2.3  # Install specific version
npm update                 # Update packages per package.json rules
npm list                   # View installed dependency tree
npm audit                  # Security vulnerability check
npm run script-name        # Run custom script
npm publish                # Publish package to registry
```

Install Types

Command	Effect	Saves to
npm install package	Local install	node_modules
npm install -g package	Global install	System-wide

package-lock.json

- Records exact versions of all installed packages
- Creates reproducible builds
- Generated automatically on npm install

Key Features

- Locks dependency tree structure
- Uses integrity hashes (SHA-512) for security
- Should be committed to version control

npm alternatives

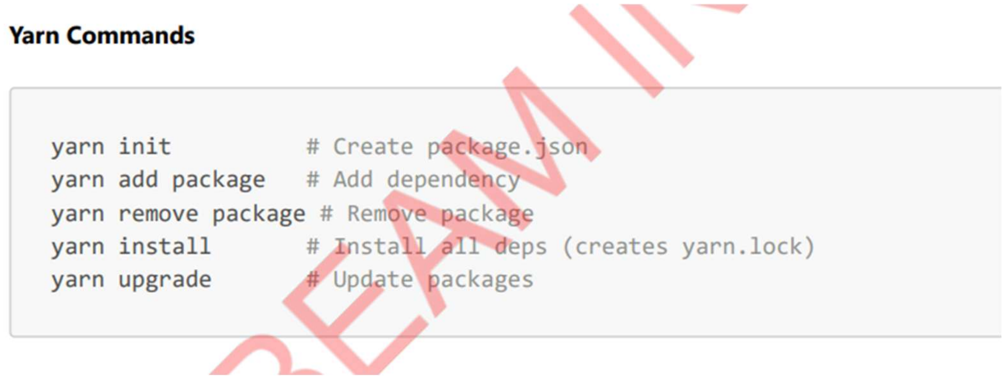
- Yarn: Faster, deterministic installs
- pnpm: Disk-efficient (hard links)
- Bun: All-in-one runtime with built-in package manager

What is Yarn?

Fast, reliable dependency manager created by Facebook (now community-maintained)
Alternative to npm with key improvements:

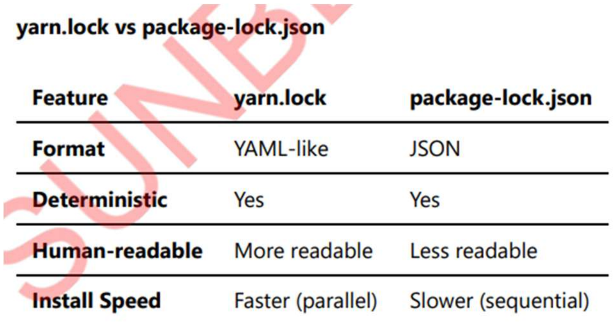
- Deterministic installs via yarn.lock (instead of package-lock.json)
- Parallel downloads for faster installation
- Works with npm registry (same packages)

- Caches packages for repeat installs without internet



Yarn Commands

```
yarn init           # Create package.json
yarn add package    # Add dependency
yarn remove package # Remove package
yarn install        # Install all deps (creates yarn.lock)
yarn upgrade        # Update packages
```



yarn.lock vs package-lock.json

Feature	yarn.lock	package-lock.json
Format	YAML-like	JSON
Deterministic	Yes	Yes
Human-readable	More readable	Less readable
Install Speed	Faster (parallel)	Slower (sequential)

Express.js

What is Express?

- **Minimalist web framework for Node.js**
- **Simplifies server creation with routing, middleware, and HTTP helpers**
- **Used by companies like Uber, Accenture, and Twitter**

Developed by: TJ Holowaychuk (initial release) and now maintained by Node.js Foundation and opensource contributors

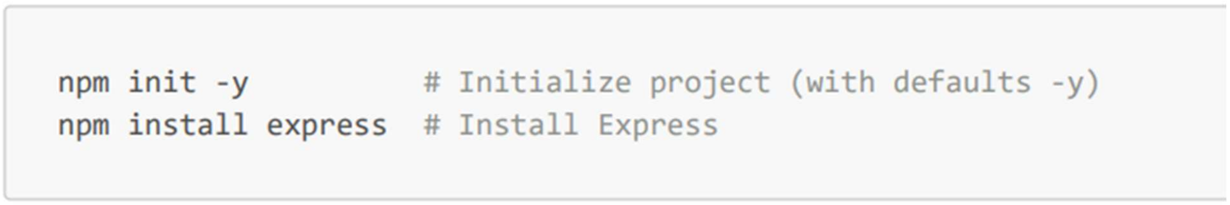
What it is:

- **A minimal, unopinionated web framework for Node.js**
- **Provides robust routing and HTTP utility methods**
- **Designed for building web applications and APIs**

Key Characteristics:

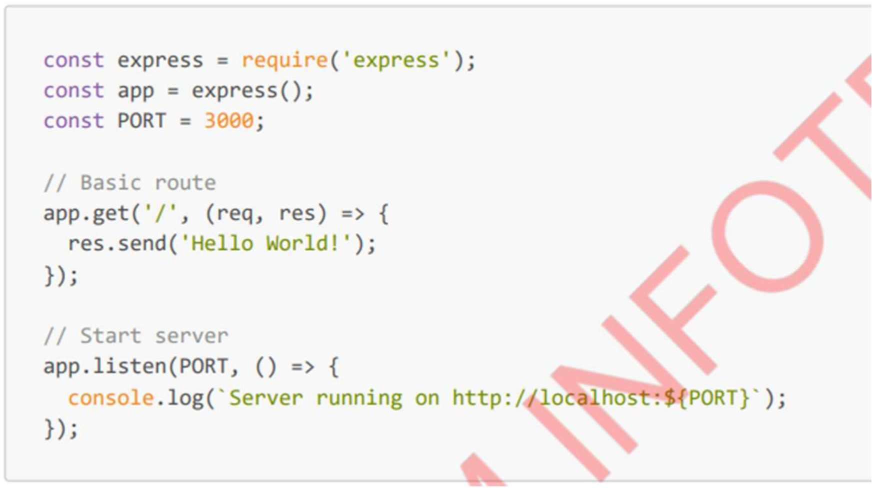
- **Fast server-side development**
- **Middleware-centric architecture**
- **Minimal core that can be extended with middleware**

Installation



```
npm init -y           # Initialize project (with defaults -y)
npm install express   # Install Express
```

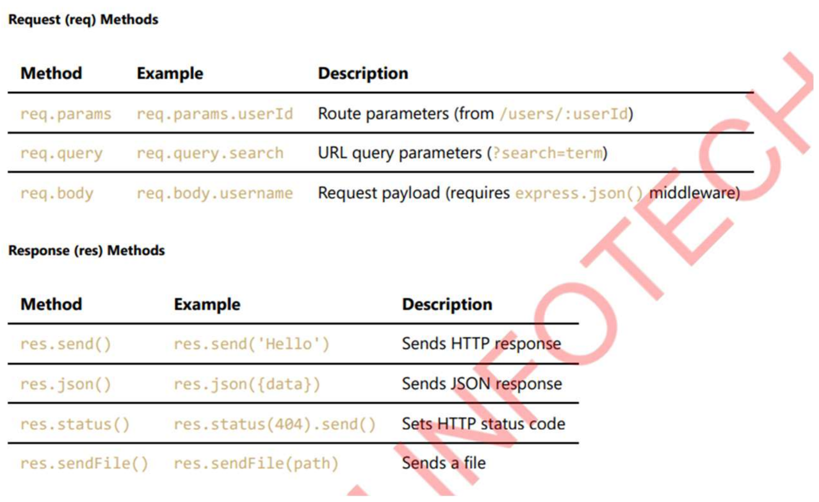
Basic Server Setup



```
const express = require('express');
const app = express();
const PORT = 3000;

// Basic route
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// Start server
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

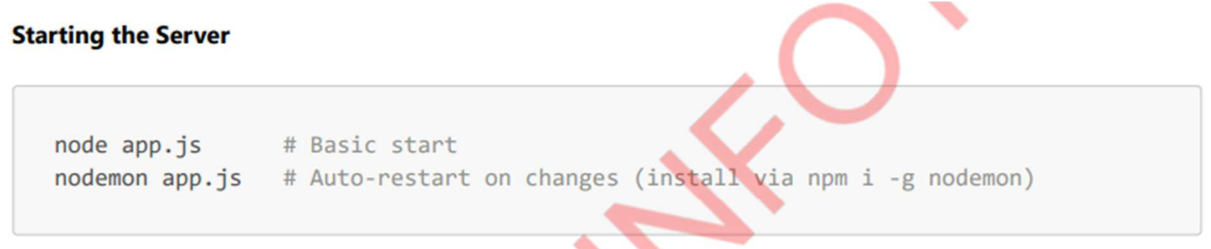


Request (req) Methods

Method	Example	Description
`req.params`	`req.params.userId`	Route parameters (from `/users/:userId`)
`req.query`	`req.query.search`	URL query parameters (`?search=term`)
`req.body`	`req.body.username`	Request payload (requires `express.json()` middleware)

Response (res) Methods

Method	Example	Description
`res.send()`	`res.send('Hello')`	Sends HTTP response
`res.json()`	`res.json({data})`	Sends JSON response
`res.status()`	`res.status(404).send()`	Sets HTTP status code
`res.sendFile()`	`res.sendFile(path)`	Sends a file



Starting the Server

```
node app.js       # Basic start
nodemon app.js    # Auto-restart on changes (install via npm i -g nodemon)
```

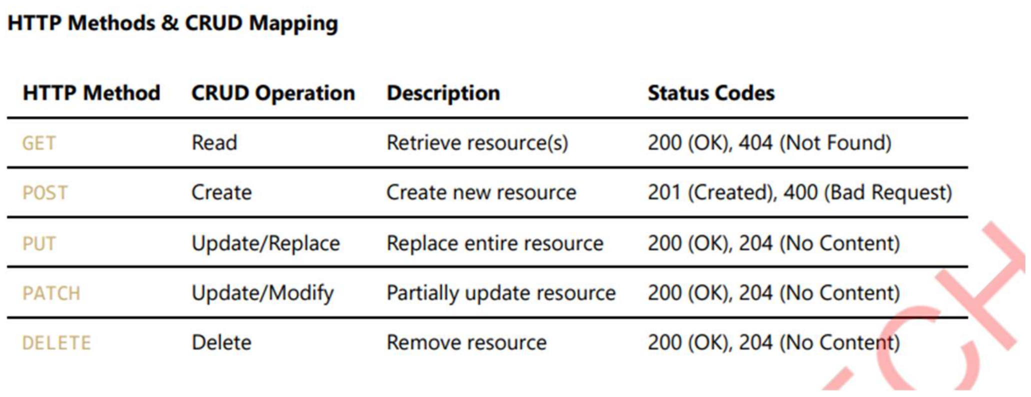
REST Protocol

What is REST?

REpresentational State Transfer (REST) is an architectural style for distributed systems

Key Principles:

- **Stateless: Each request contains all necessary information**
- **Client-Server: Separation of concerns**
- **Cacheable: Responses can be cached**
- **Uniform Interface: Consistent resource identification and manipulation**
- **Layered System: Intermediary servers can be inserted without client awareness**


HTTP Methods & CRUD Mapping

HTTP Method	CRUD Operation	Description	Status Codes
GET	Read	Retrieve resource(s)	200 (OK), 404 (Not Found)
POST	Create	Create new resource	201 (Created), 400 (Bad Request)
PUT	Update/Replace	Replace entire resource	200 (OK), 204 (No Content)
PATCH	Update/Modify	Partially update resource	200 (OK), 204 (No Content)
DELETE	Delete	Remove resource	200 (OK), 204 (No Content)

REST Best Practices

1. Resource Naming:

- **Use nouns (not verbs): /books not /getBooks**
- **Plural form for collections: /users not /user**

2. HTTP Status Codes:

- **200 - Successful GET/PUT/PATCH**
- **201 - Resource created**
- **204 - No content (successful DELETE)**
- **400 - Bad request**
- **401 - Unauthorized**
- **404 - Not found**
- **500 - Server error**

3. Versioning:

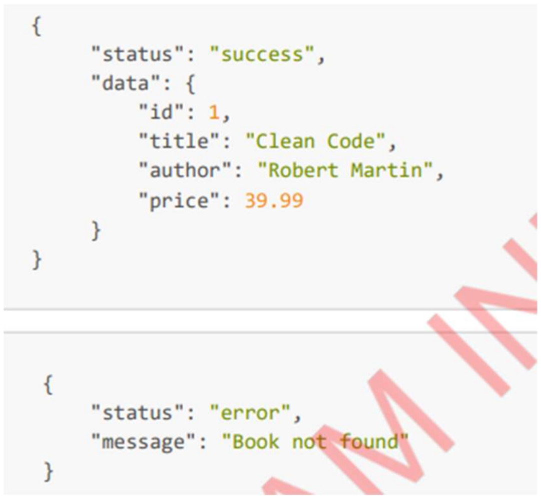
GET /v1/books

Accept: application/vnd.example.v1+json

4. Filtering/Sorting:

GET /books?author=Martin&sort=-price

5. Consistent response (like JSend format):



```
{
    "status": "success",
    "data": {
        "id": 1,
        "title": "Clean Code",
        "author": "Robert Martin",
        "price": 39.99
    }
}
```

```
{
    "status": "error",
    "message": "Book not found"
}
```

Middleware in Express.js

Core Concepts

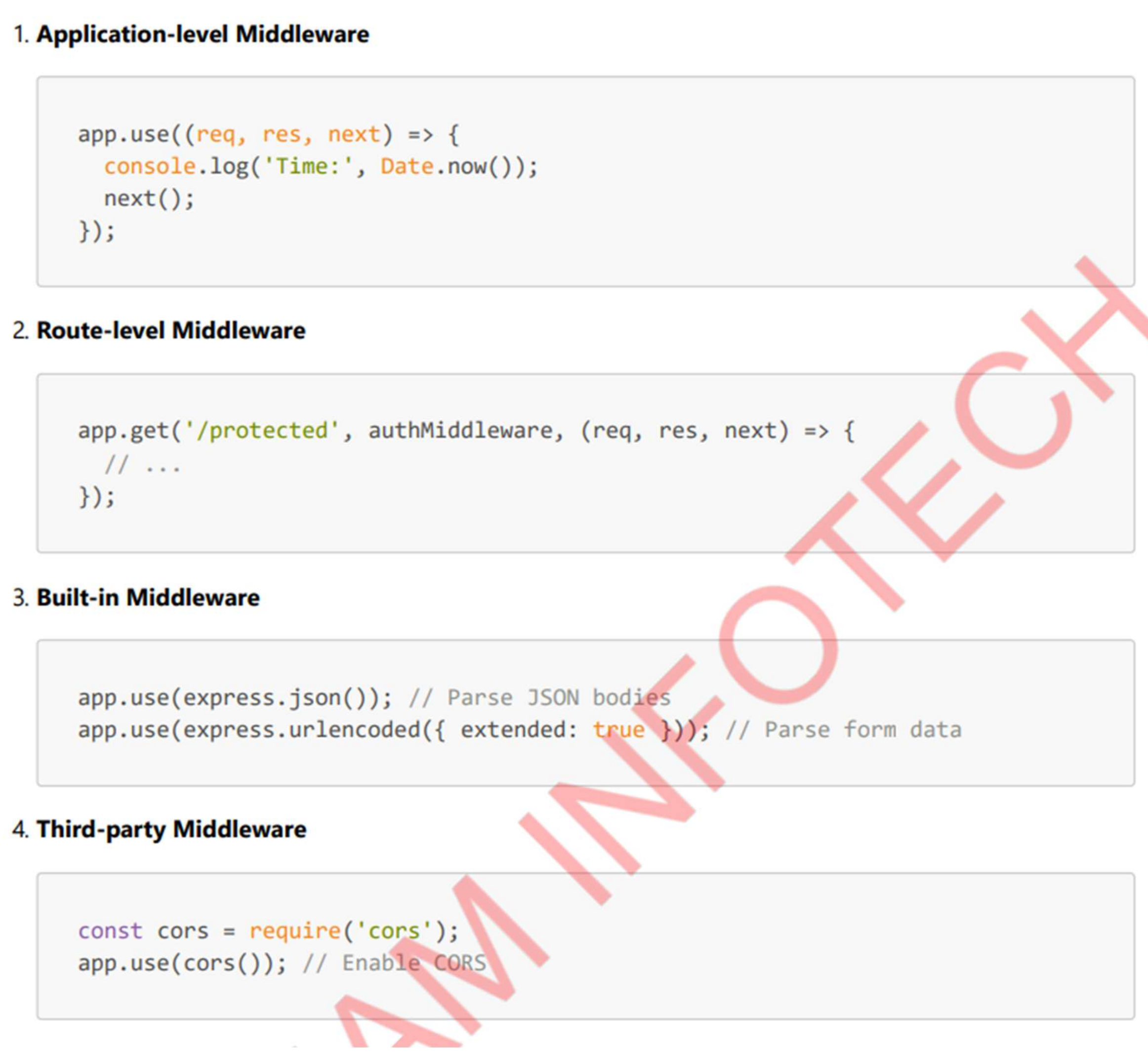
Definition: Functions that execute during the request-response cycle

$$\begin{tikzcd}\boxed{\text{Request}} \arrow[r] & \boxed{\text{Middleware 1}} \arrow[r] & \boxed{\text{Middleware 2}} \arrow[r] & \boxed{\text{Route Handler}} \arrow[r] & \boxed{\text{Response}}\end{tikzcd}$$

Key Characteristics:

- **Access to req, res objects**
- **Can modify requests/responses**
- **Must call next() to pass control (or end the cycle)**

Types of Middleware



1. **Application-level Middleware**

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

2. **Route-level Middleware**

```
app.get('/protected', authMiddleware, (req, res, next) => {
  // ...
});
```

3. **Built-in Middleware**

```
app.use(express.json()); // Parse JSON bodies
app.use(express.urlencoded({ extended: true })); // Parse form data
```

4. **Third-party Middleware**

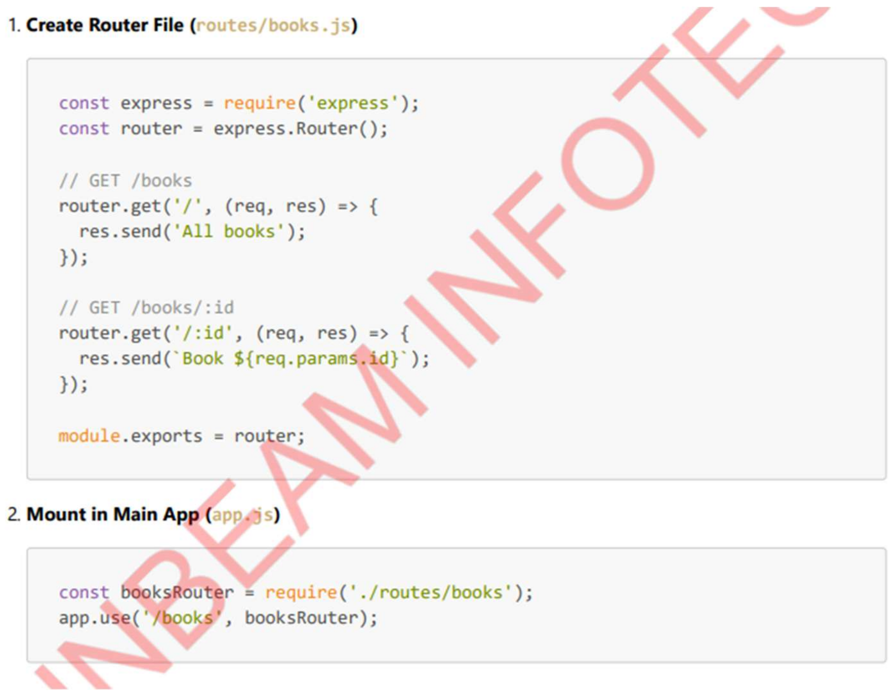
```
const cors = require('cors');
app.use(cors()); // Enable CORS
```

Express Router

Why Use Router?

- Organize routes into separate files
- Create modular, maintainable code
- Apply middleware to specific route groups
- Mount multiple routers with different paths

Router Implementation



1. **Create Router File** (`routes/books.js`)

```
const express = require('express');
const router = express.Router();

// GET /books
router.get('/', (req, res) => {
  res.send('All books');
});

// GET /books/:id
router.get('/:id', (req, res) => {
  res.send(`Book ${req.params.id}`);
});

module.exports = router;
```

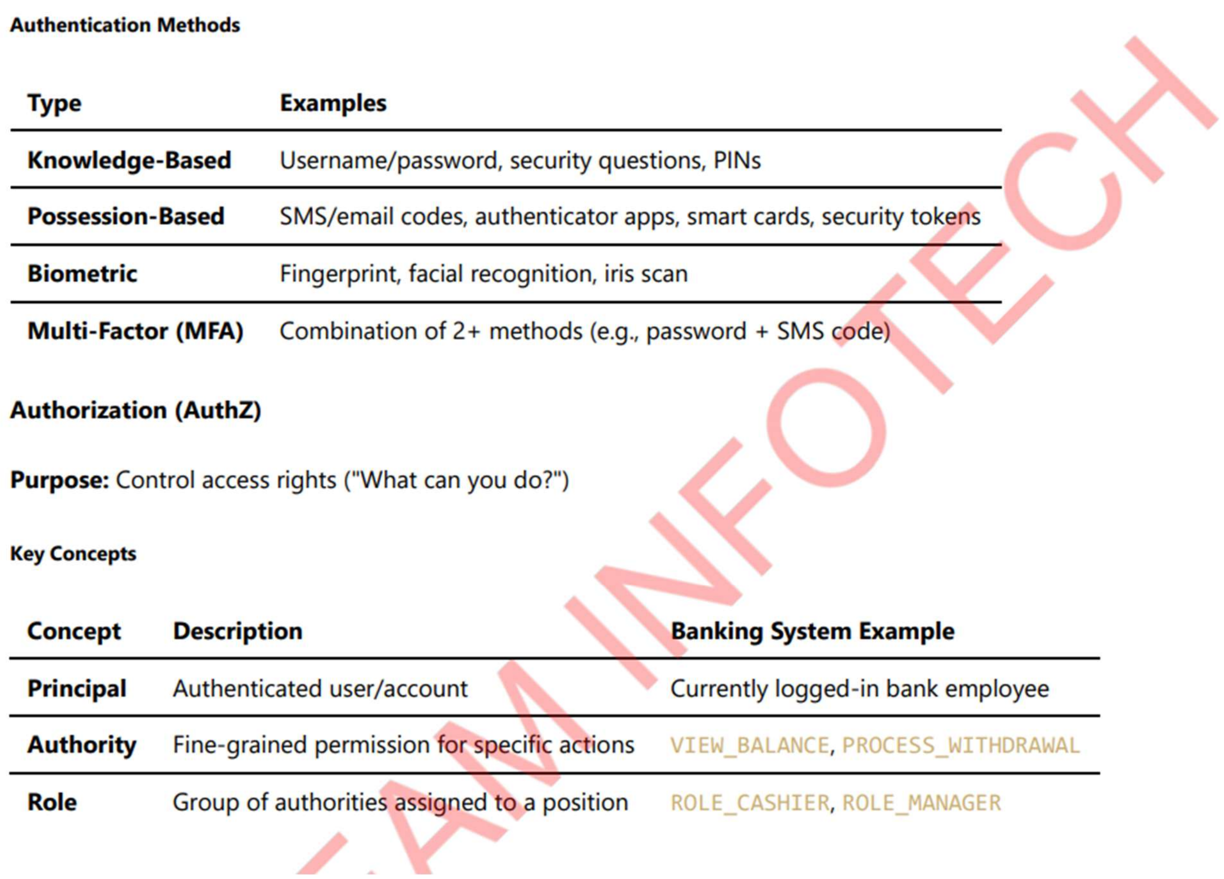
2. **Mount in Main App** (`app.js`)

```
const booksRouter = require('./routes/books');
app.use('/books', booksRouter);
```

Express Security

Authentication (AuthN)

Purpose: Verify user identity ("Who are you?")



Authentication Methods

Type	Examples
Knowledge-Based	Username/password, security questions, PINs
Possession-Based	SMS/email codes, authenticator apps, smart cards, security tokens
Biometric	Fingerprint, facial recognition, iris scan
Multi-Factor (MFA)	Combination of 2+ methods (e.g., password + SMS code)

Authorization (AuthZ)

Purpose: Control access rights ("What can you do?")

Key Concepts

Concept	Description	Banking System Example
Principal	Authenticated user/account	Currently logged-in bank employee
Authority	Fine-grained permission for specific actions	`VIEW_BALANCE`, `PROCESS_WITHDRAWAL`
Role	Group of authorities assigned to a position	`ROLE_CASHIER`, `ROLE_MANAGER`

JWT (JSON Web Token)

JWT Overview

Definition: Open standard (RFC 7519) for secure information exchange

- **Key Characteristics:**
- **Compact (URL-safe string)**
- **Self-contained (includes all necessary information)**
- **Digitally signed (verifiable integrity)**

Common Uses:

- **Authorization (primary use case)**
- **Secure information exchange**
- **Stateless authentication**

-

Part	Content Type	Description	Example
Header	JSON	Token type + signing algorithm	`{"alg":"HS256","typ":"JWT"}`
Payload	JSON	Claims (user data) + standard fields (`iss`, `exp`, `sub`, etc.)	`{"userId":123,"role":"admin"}`
Signature	Binary	Cryptographic signature	HMAC-SHA256 output

Configuration

```
const jwt = require('jsonwebtoken');
const JWT_SECRET = process.env.JWT_SECRET || 'your-256-bit-secret'; // Always use
env vars in production
const TOKEN_EXPIRY = '7d'; // 7 days
```

Token creation

- To be created after successful login.

```
function createToken(user) {
  const payload = {
    userId: user.id,
    role: user.role,
    // Add other necessary claims (but avoid sensitive data)
  };
  return jwt.sign(payload, JWT_SECRET, { expiresIn: TOKEN_EXPIRY });
}
```

Token verification

- To be verified on each request

```
// 2. Token Verification
function verifyToken(token) {
  try {
    return jwt.verify(token, JWT_SECRET);
```

```
  } catch (err) {
    console.error('JWT verification failed:', err.message);
    return null;
  }
}
```

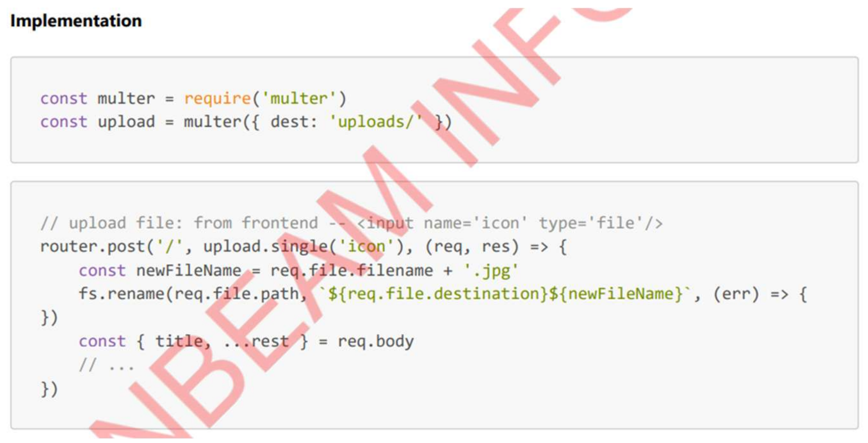
File Upload with Multer

Core Concepts

Multer: Node.js middleware for handling multipart/form-data

Key Features:

- **Disk storage engine (saves to filesystem)**
- **Memory storage engine (buffers in RAM)**
- **File filtering and size limits**
- **Multiple file upload support**



Implementation

```
const multer = require('multer')
const upload = multer({ dest: 'uploads/' })
```

```
// upload file: from frontend -- <input name='icon' type='file'/>
router.post('/', upload.single('icon'), (req, res) => {
    const newFileName = req.file.filename + '.jpg'
    fs.rename(req.file.path, `${req.file.destination}${newFileName}`, (err) => {
})
    const { title, ...rest } = req.body
    // ...
})
```