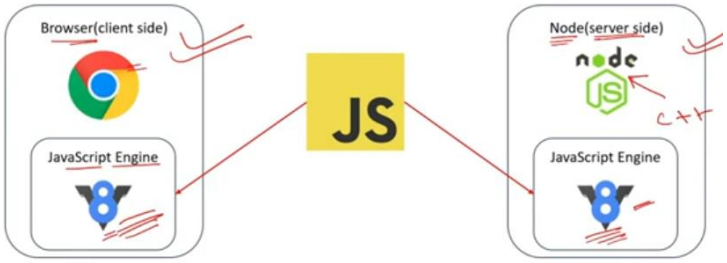
NodeJS

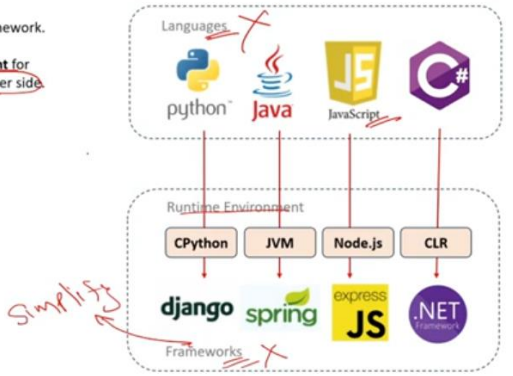
Q. How Node is a runtime environment on server side? What is V8?

- 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.



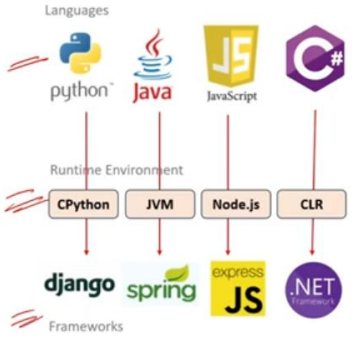
Q. What is Node.js? V. IMP.

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



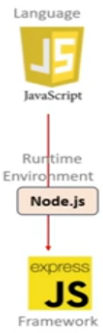
Q. What is the difference between Runtime environment & Framework?

- **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.

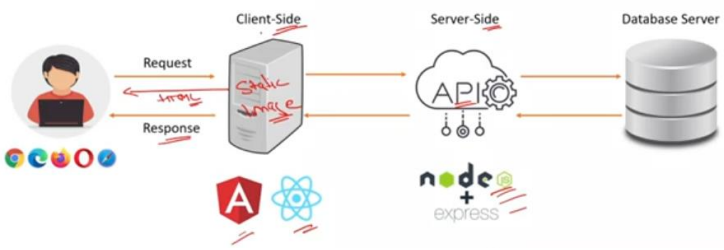


Q. What is the difference between Node.js & Express.js?

- 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.



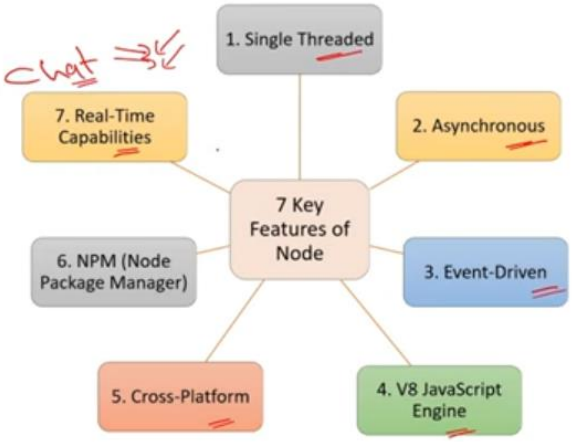
Q. What are the differences between Client-Side(Browser) & Server-Side(Node.js)?



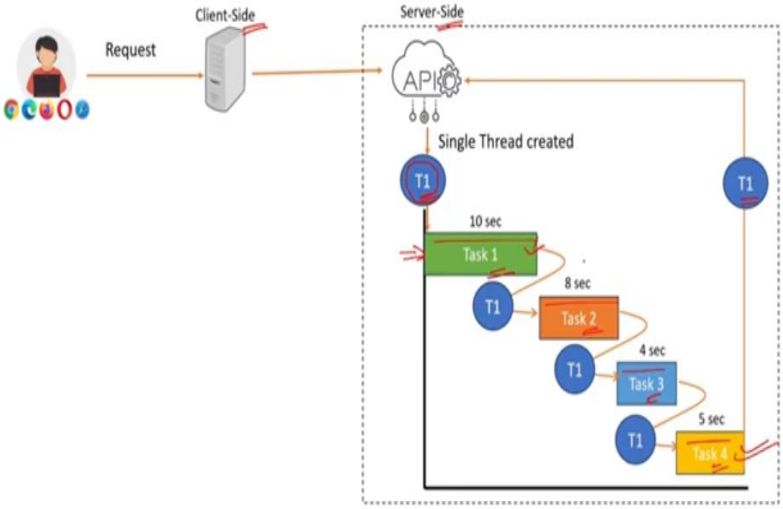
Q. What are the differences between Client Side(Browser) & Server-Side(Node.js)?

	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.

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

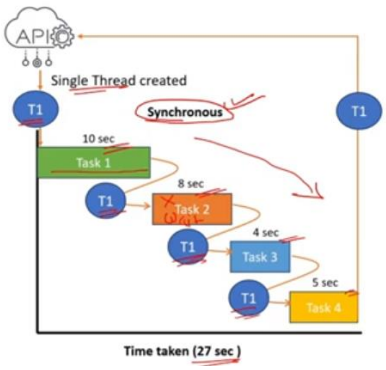


Q. What is Single Threaded Programming?



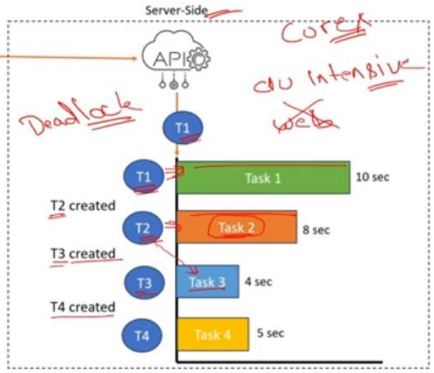
Q. What is Synchronous Programming?

- 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.



Q. What is Multi Threaded Programming?

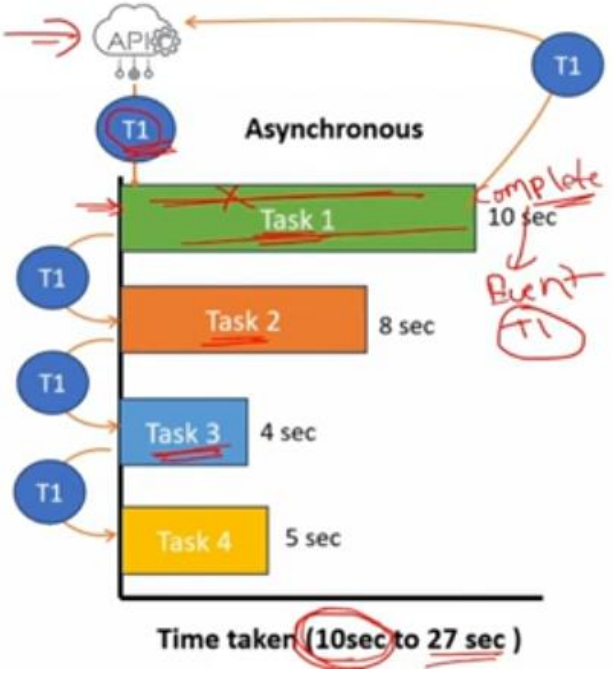
- In a multi-threaded application, multiple threads can execute multiple tasks simultaneously and parallelly.
- As soon as one thread starts a task, the next task is immediately initiated by creating a second thread.



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.

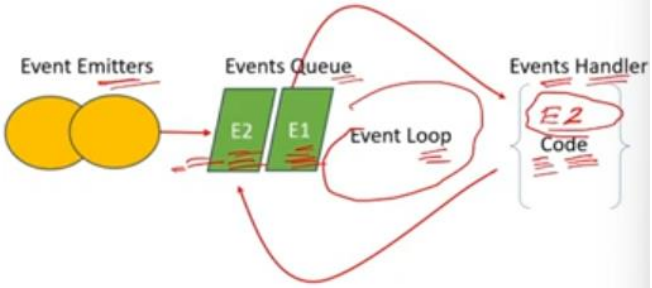


Q. What is the difference between Synchronous & Asynchronous programming?

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.

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.



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.

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

When to Use Node.js?

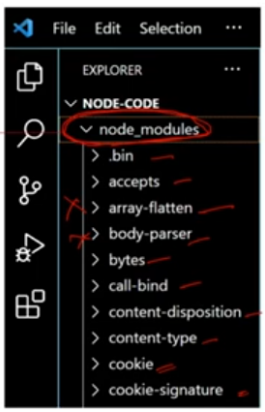
- Ideal for **real-time applications** like chat applications, online gaming, and collaborative tools due to its event-driven architecture.
- Excellent for building **lightweight and scalable RESTful APIs** that handle a large number of concurrent connections.
- Well-suited for building **microservices-based architectures**, enabling modular and scalable systems.

When not to use Node.js(disadvantages):

- **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.

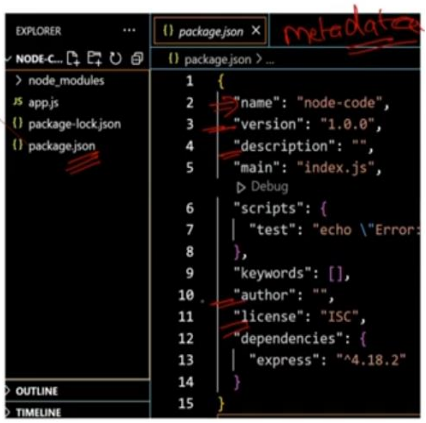
Q. What is NPM? What is the role of node_modules folder?

- 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.



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.

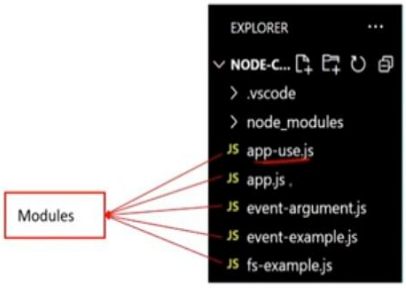


Q. What are Modules in Node? What is the difference between a function & module?

- 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.



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",
};
```

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

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

```
// 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
```

```
// 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');
```

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
});
```

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

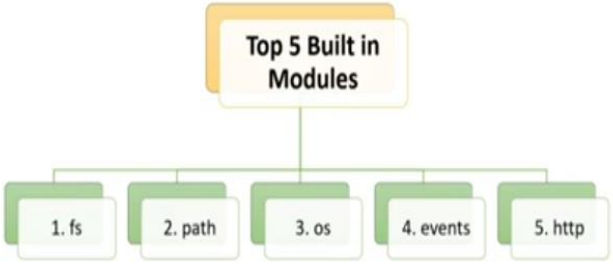
These **user-defined modules(JS files)** created by developers for specific functionalities.

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.

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?

- 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.");
});
```

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.

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'
}
*/
```

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');
```

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
```

Cross platform

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
```

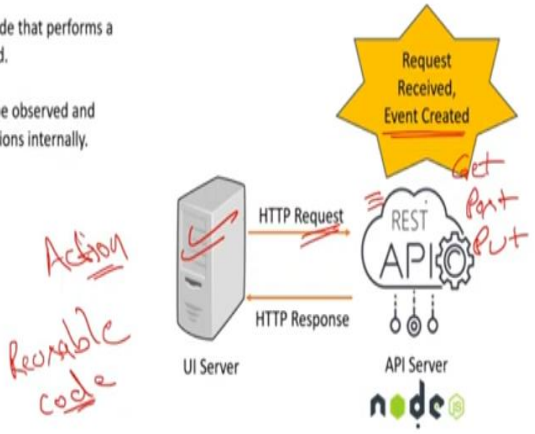
Q. What are Event Arguments?

- 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.
```

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.

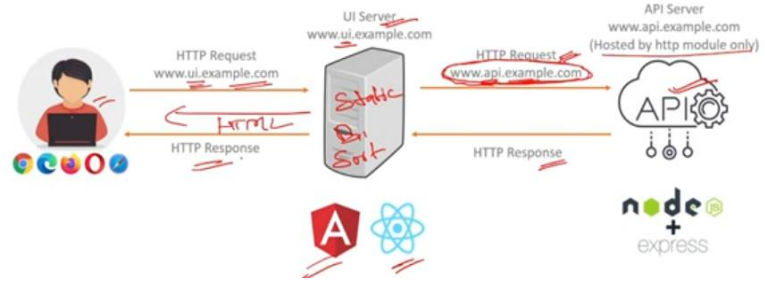


Action

Reusable code

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.



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.**


```
// 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
```

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.

HTML + Data

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}`);
});
```

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.

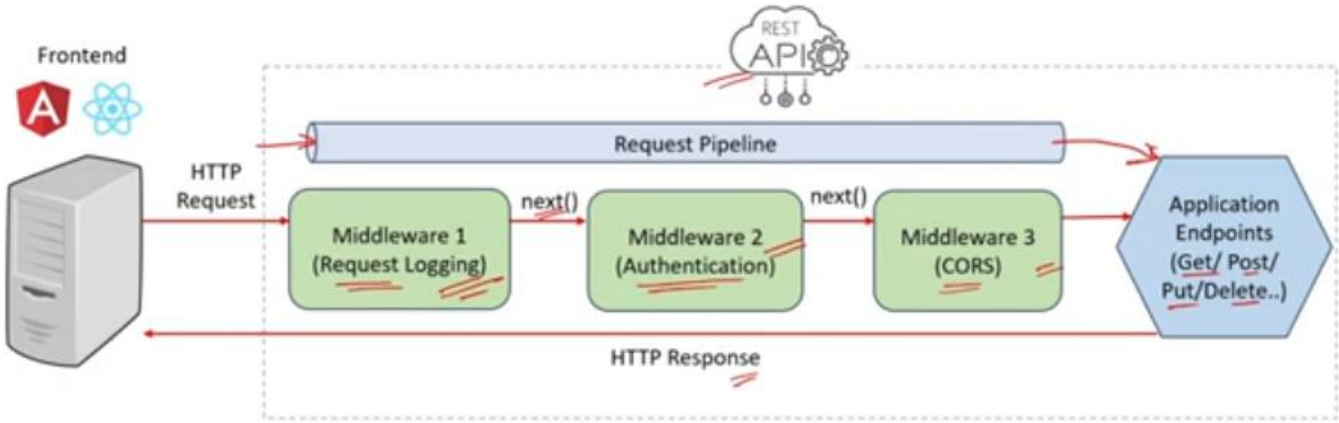
```
// Create an Express.js application
const express = require("express");
const app = express();
```

- 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}`);
});
```

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

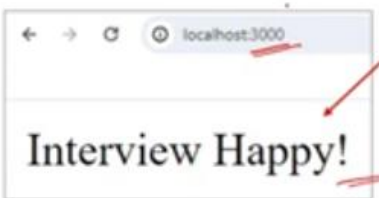
- A middleware in Express.js is a function that handles HTTP requests, performs operations, and passes control to the next middleware.



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}`);
});
```

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}`);
});
```

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}`);
});
```

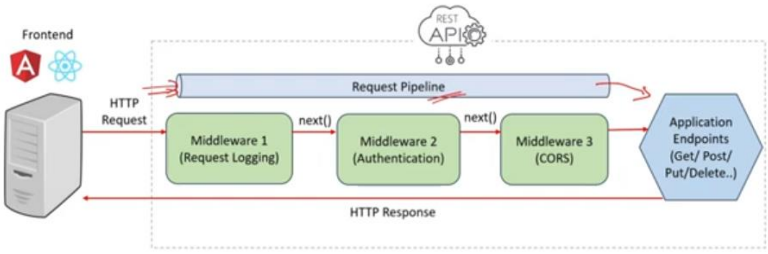
Q. How to use middleware globally for a specific route?

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

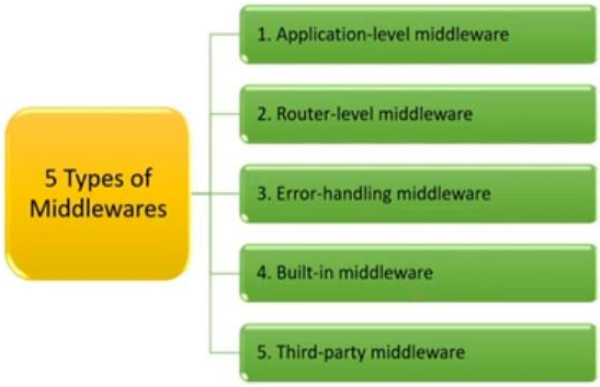

```
const express = require('express');
const app = express();

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}`);
});
```

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.



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



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.

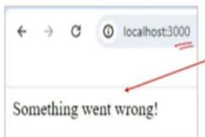
```
const express = require("express");
const app = express();

// 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); globally

// 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}`);
});
```

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.




```
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\
```

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!");
});
```

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}`);
});
```

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());
```

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

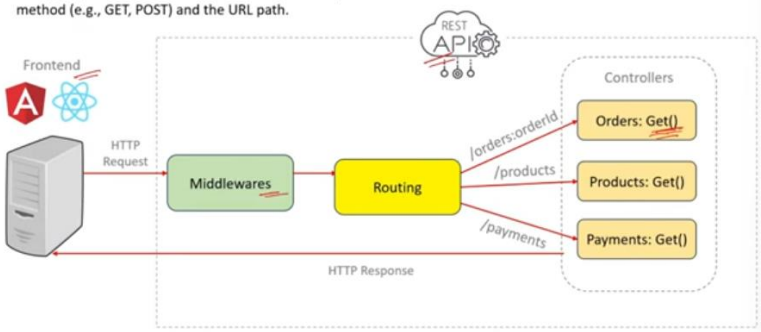
1. Application-level middleware	Middleware applied to all routes; commonly used for logging, authentication, etc.
2. Router-level middleware	Middleware specific to certain routes; applied using or .
3. Built-in middleware	Pre-packaged middleware included with Express.js, like for serving static files.
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.

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.

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.

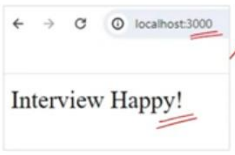


Q. What is the difference between middleware & routing in Express?

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).

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).




```
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}`);
});
```

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}`);
});
```

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}`);
});
```

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}`);
});
```

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.).

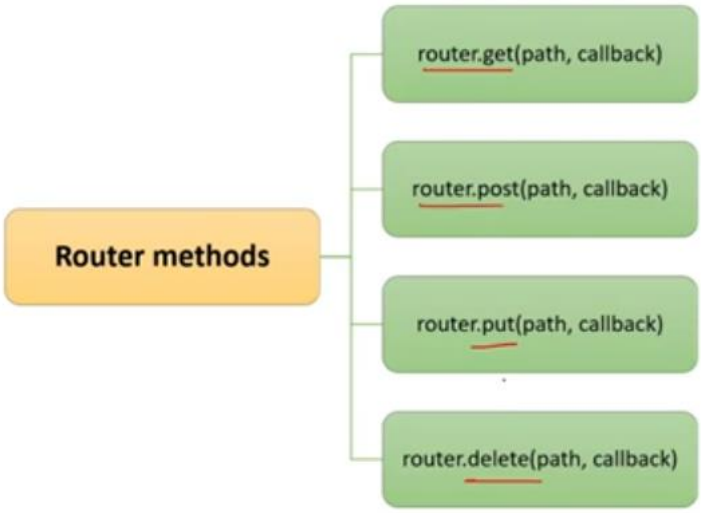
- Implement router method

```
// 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");
});
```

Q. What are the types of Router Methods?



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.

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.

```
// 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");
});
```

```
// 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;
```

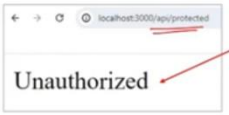
Q. What is express.Router() in Express.js?

- 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;
```

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);
```

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.

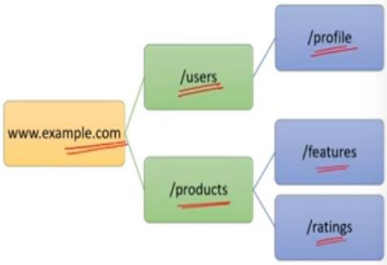

```
const express = require("express");
const app = express();

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
```

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



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

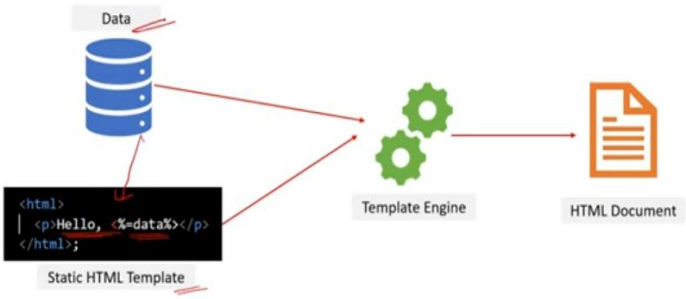
```
// usersRouter.js
const express = require("express");
const usersRouter = express.Router();

// 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;
```

```
// 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;
```

```
// 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");
});
```

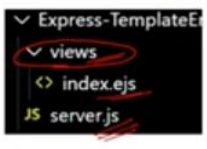
Q. What are Template Engines in Express.js? V. IMP.

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

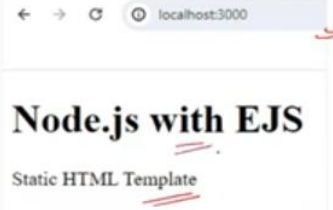


Q. How to implement EJS templating engine in a Express.js application?

```
npm install ejs
```



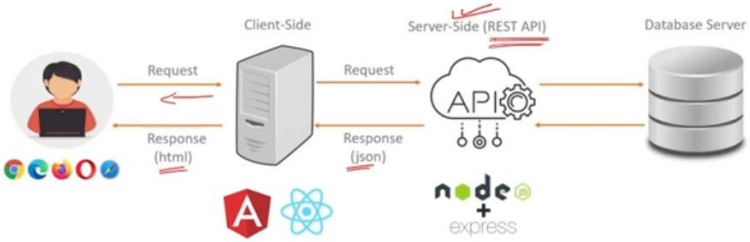

```
<!-- index.ejs -->
<html lang="en">
  <head>
    <title>EJS Example</title>
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Static HTML Template</p>
  </body>
</html>
```



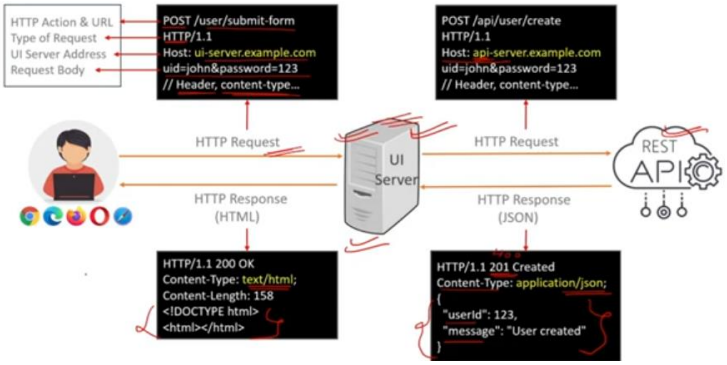

```
//server.js
const express = require('express');
const app = express();
const path = require('path');

// 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}`);
});
```

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.



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



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

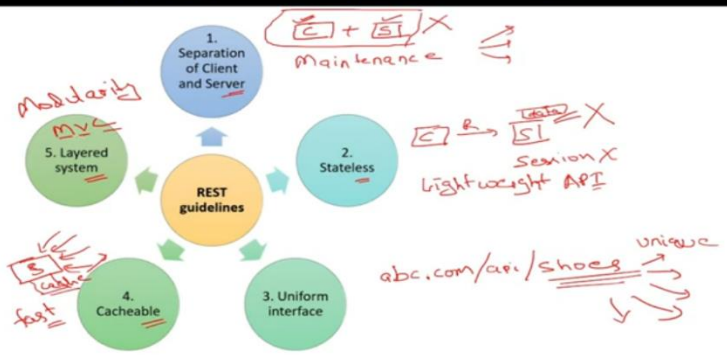
HTTP Request

- 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.

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



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.

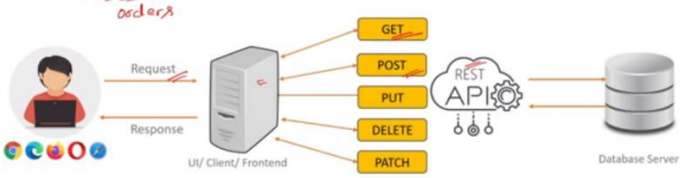
5. Layered system — The system should follow layered pattern.
- Advantage: A layered system, such as the Model-View-Controller (MVC) pattern, promotes **modular design** and separation of concerns.

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.

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.



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)

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

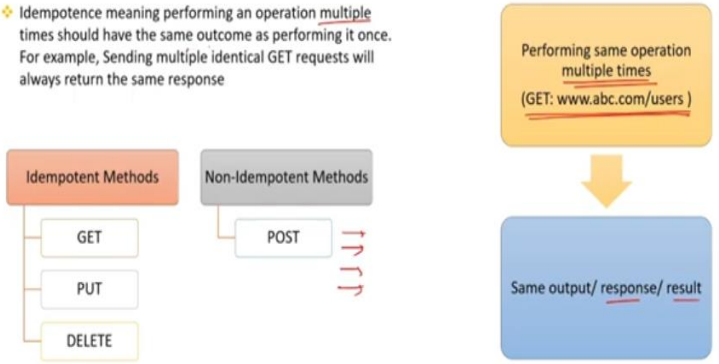
PUT	PATCH
Both PUT and PATCH method are used to **update a resource** by replacing the resource with the new data provided in the request.	
Full Resource Replacement: In a PUT request, the client sends the full updated resource in the request body, replacing the existing resource on the server.	**Partial 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
}
```

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



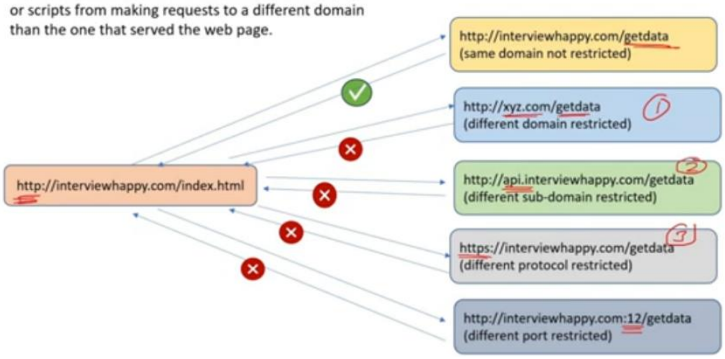
Q. What are the role of status codes in RESTful APIs?

- 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

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.



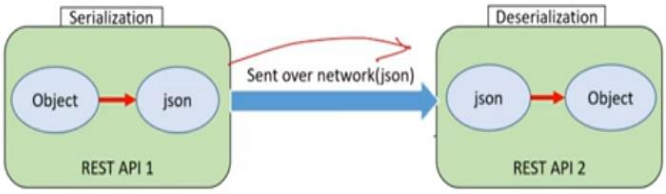
Q. How to remove CORS restrictions RESTful APIs?

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

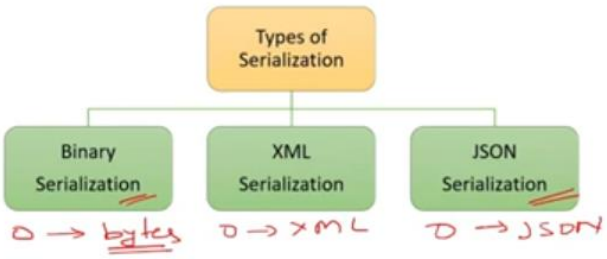
```
const express = require('express');
const cors = require('cors'); // Import cors module

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`);
});
```

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.



Q. What are the types of serialization?



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

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

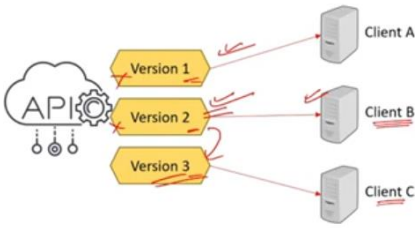
```
// Serialization (to JSON)
const obj = { name: "Happy", age: 39 };

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 }
```

Q. Explain the concept of versioning in RESTful APIs.

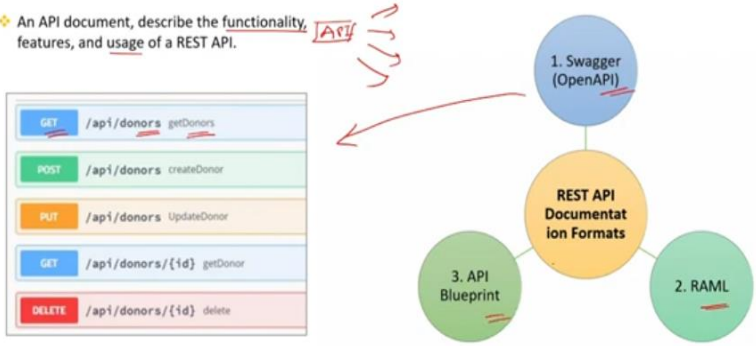
- Versioning in RESTful APIs refers to the practice of **maintaining multiple versions** of an API to support backward compatibility.




```
https://api.example.com/v1/resource
https://api.example.com/v2/resource
https://api.example.com/v3/resource
```

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.



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
```

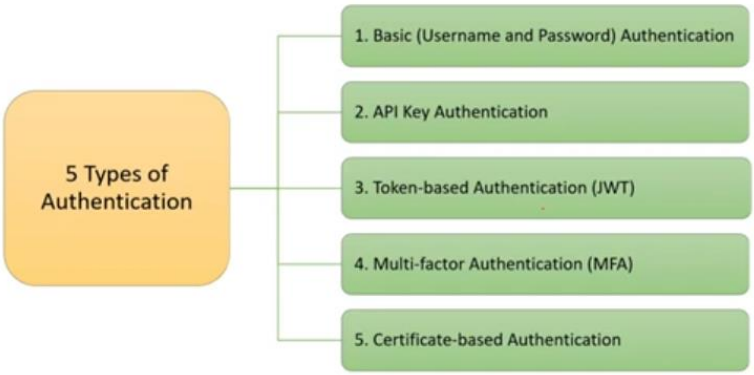
Q. What are Authentication and Authorization? V. IMP.

- 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?)

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



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?)

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)

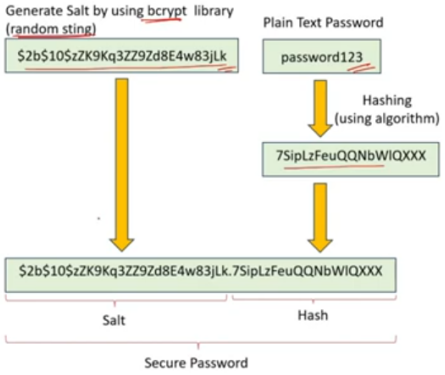
Encode

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

Stored in Encrypted format (More secure)

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.



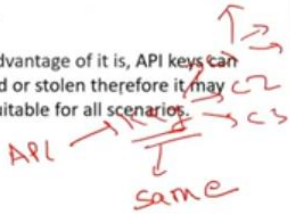
Q. How can we create Hash Passwords in Node.js?

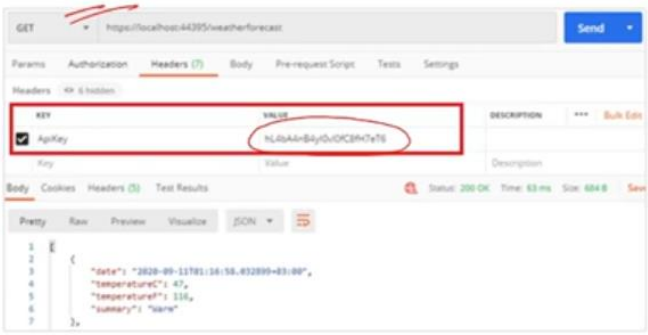
1. Generate a random salt
2. Create a hash object using SHA-256
3. Update the hash object with the salt and password
4. Get the hashed data in a hexadecimal string
5. Return the salt and hashed password as a string

```
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....
}
```

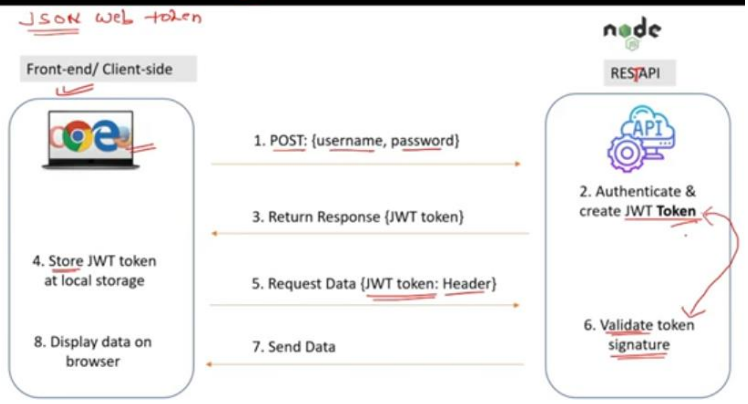
Q. What is API Key Authentication?

- **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.



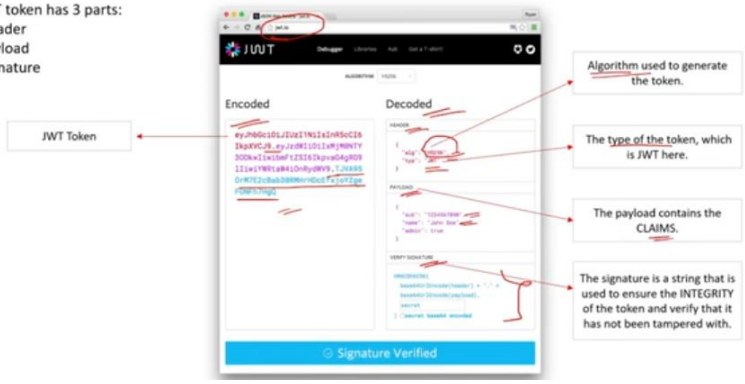


Q. What is Token based and JWT authentication? V. IMP.



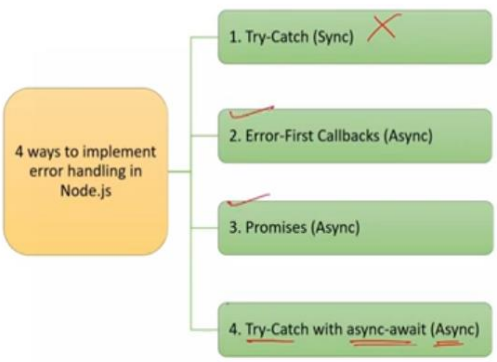
Q. What are the parts of JWT token?

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



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.



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
```

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
```

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));
```

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();
```

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