Chapter 1: Node.js and Backend Foundations

1.1 Introduction to MERN (MongoDB, Express.js, React.js, Node.js)

The MERN stack is a popular and powerful collection of JavaScript-based technologies used for developing full-stack web applications. "Full-stack" means it covers both the front-end (what users see and interact with) and the back-end (the server, database, and application logic).

The acronym MERN stands for:

- **M - MongoDB:** This is a NoSQL (non-relational) database. Unlike traditional relational databases that use tables and rigid schemas, MongoDB stores data in flexible, JSON-like documents. This makes it highly scalable and well-suited for handling large amounts of diverse data, and it integrates seamlessly with JavaScript-based applications.
- **E - Express.js:** Express is a fast, minimalist web application framework for Node.js. It simplifies the process of building robust APIs and server-side logic, handling HTTP requests, routing, and middleware. It acts as the bridge between your front-end and your database.
- **R - React.js:** React is a JavaScript library for building user interfaces (UIs). It's known for its component-based architecture, which allows developers to create reusable UI elements. React uses a virtual DOM (Document Object Model) to efficiently update and render only the necessary parts of the UI, leading to highly interactive and dynamic single-page applications (SPAs).
- **N - Node.js:** Node.js is a JavaScript runtime environment that allows you to execute JavaScript code outside of a web browser. It serves as the back-end component of the MERN stack, enabling server-side scripting. Node.js's event-driven, non-blocking I/O model makes it highly efficient and scalable for building backend services that can handle many concurrent connections.

How the MERN Stack Works:

In a MERN application:

1. The **React.js** front-end handles the user interface and interactions. When a user performs an action (like clicking a button or submitting a form), React sends an HTTP request to the back-end.
2. The **Express.js** framework, running on a **Node.js** server, receives these requests. Express processes the request, performs any necessary business logic or validation, and then interacts with the database.
3. **MongoDB** stores and retrieves the application's data in JSON-like documents.
4. Once the database operation is complete, Express sends a response back to the React front-end.
5. React then updates the UI based on the new data or status received.

Benefits of the MERN Stack:

- **Full-stack JavaScript:** One of the biggest advantages is using a single language (JavaScript) across the entire application stack (front-end, back-end, and database interaction). This streamlines development, simplifies the learning curve for JavaScript developers, and promotes code reusability.
- **Efficiency and Speed:** The integrated nature of the MERN components allows for rapid application development.
- **Scalability:** Each component of the MERN stack is designed to be scalable, making it suitable for applications that need to handle a growing number of users and data.
- **Strong Community Support:** All MERN technologies are open-source with large, active communities, providing ample resources, tutorials, and support for developers.
- **JSON-native:** The consistent use of JSON (or BSON in MongoDB) for data exchange across the stack simplifies data flow and reduces the need for data transformation.

1.2 Understanding MVC and Component-Based Architecture

When building applications with the **MERN stack** (MongoDB, Express.js, React, Node.js), it's helpful to understand how **MVC architecture** and **Component-Based Architecture** work, and how they **interact** or **differ** in this context.

Model-View-Controller (MVC)

MVC is a software architectural pattern that separates an application into three main interconnected components:

- **Model:**
  Manages the application's data, business logic, and rules. It represents the "what" of the application. It's responsible for retrieving, storing, and manipulating data. It might interact with a database, external APIs, or other data sources. The Model is independent of the user interface.
  In an e-commerce application, the Model would handle product information (name, price, description), user data, order details, and the logic for calculating discounts or processing payments.
- **View:**
  Handles the presentation layer and displays data to the user. It represents the "how" the data is shown. It's responsible for the user interface (UI), including HTML, CSS, and client-side JavaScript that renders the data received from the Model. It also captures user input.
  For the e-commerce app, the View would be the actual web page displaying product listings, a shopping cart, or a user's profile.
- **Controller:**
  Acts as an intermediary between the Model and the View. It represents the "who" or "what controls" the interaction. It receives user input from the View (e.g., a button click, form submission), processes it, and then updates the Model or the View accordingly. It contains the application's flow logic.

When a user clicks "Add to Cart," the Controller receives this input. It then tells the Model to update the shopping cart data, and once the Model is updated, the Controller might tell the View to re-render the shopping cart to show the new item.

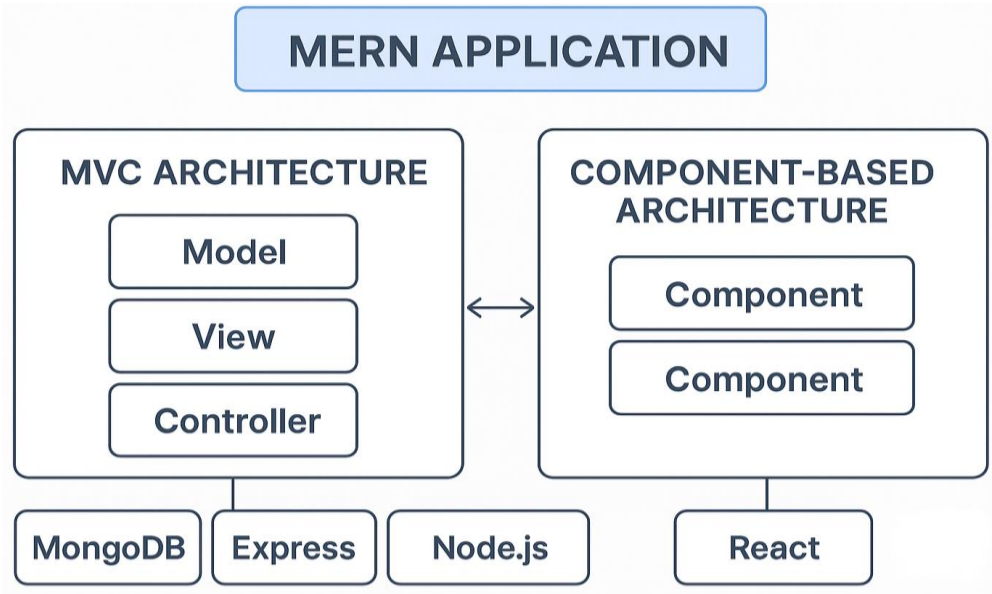
Component-Based Architecture

Component-Based Architecture (CBA) focuses on building applications by assembling self-contained, reusable, and independent building blocks called "components." This approach is particularly prominent in modern front-end frameworks like React, Angular, and Vue.js.

- **Component:**
  Responsibility: A component is a self-contained unit of functionality and UI. It encapsulates its own logic, data, and presentation.
  Functionality: Think of a component as a Lego block. It has a specific purpose (e.g., a button, a navigation bar, a user profile card, a product listing). It manages its own state and can communicate with other components through well-defined interfaces (props, events).
  Reusability: Once a component is built, it can be reused multiple times throughout the application or in different projects, reducing development time and ensuring consistency.
  Encapsulation: A component hides its internal implementation details and exposes only what's necessary through its public interface.
  Independence: Components have minimal dependencies on other components, making them easier to develop, test, and maintain in isolation.

How it works (e.g., in React):

In React, you break down your UI into a hierarchy of components. A large page might be composed of smaller components, which in turn are composed of even smaller ones. Data flows down the component tree (from parent to child components via "props"), and events or callbacks flow up.



1.3 Node.js

Node.js is a powerful open-source, cross-platform JavaScript runtime environment that allows developers to build scalable network applications. It's built on Chrome's V8 JavaScript engine and is widely used for back-end services, APIs, and real-time applications.

Core Modules and Custom Modules

Core Modules: Node.js comes with a set of built-in modules that provide essential functionalities without needing to be installed separately. These modules are readily available for use in any Node.js application. You access them using the `require()` function.

Some commonly used core modules include:

- **`http`**: For creating HTTP servers and making HTTP requests.
- **`fs`**: For interacting with the file system (reading, writing, deleting files, etc.).
- **`path`**: For working with file and directory paths.
- **`events`**: For implementing event-driven programming (Event Emitter).
- **`os`**: For retrieving operating system-related information.
- **`util`**: For utility functions (e.g., `promisify`).
- **`stream`**: For working with data streams.

Example of using a core module (`fs`):

```
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) =>
{
   if (err)
   {
      console.error('Error reading file:', err);
      return;
   }
   console.log('File content:', data);
});
```

Custom Modules: These are modules that you create yourself to organize your code into smaller, reusable units. Custom modules help in maintaining a clean and modular codebase, promoting reusability and separation of concerns.

To create a custom module, you define functions, variables, or classes and then export them using `module.exports` or `exports`. To use a custom module, you `require()` it by its relative or absolute path.

```
// my_module.js
const PI = 3.14159;
function calculateCircleArea(radius)
{
  return PI * radius * radius;
}
function calculateCircumference(radius)
 {
  return 2 * PI * radius;
}
module.exports = {
  area: calculateCircleArea,
  circumference: calculateCircumference,
  PI: PI
};

// main.js
const circleUtils = require('./my_module');
console.log('Area of circle with radius 5:', circleUtils.area(5));
console.log('PI:', circleUtils.PI);
```

Working with the File System and Streams

File System (`fs` module): The `fs` module provides a wide range of functions for interacting with the file system. These functions can be synchronous (blocking) or asynchronous (non-blocking). Asynchronous methods are generally preferred in Node.js to avoid blocking the event loop.

Common `fs` operations:

- **Reading files:** `fs.readFile()`, `fs.readFileSync()`, `fs.createReadStream()`
- **Writing files:** `fs.writeFile()`, `fs.writeFileSync()`, `fs.createWriteStream()`
- **Appending to files:** `fs.appendFile()`, `fs.appendFileSync()`
- **Deleting files:** `fs.unlink()`, `fs.unlinkSync()`
- **Checking file existence:** `fs.existsSync()`, `fs.access()`
- **Creating/removing directories:** `fs.mkdir()`, `fs.rmdir()`

```
const fs = require('fs');
const content = 'Hello, Node.js File System!';
// Write to a file
fs.writeFile('notes.txt', content, (err) => {
  if (err) {
    console.error('Error writing file:', err);
    return;
  }
  console.log('File "notes.txt" written successfully.');
  // Read from the file
  fs.readFile('notes.txt', 'utf8', (err, data) => {
    if (err) {
      console.error('Error reading file:', err);
      return;
    }
    console.log('Content of "notes.txt":', data);
  });
});
```

Streams: Streams are a fundamental concept in Node.js for handling data efficiently, especially when dealing with large amounts of data. They allow you to process data in chunks rather than loading the entire data into memory, which can lead to better performance and reduced memory consumption.

Node.js has four types of streams:

- **Readable Streams:** For reading data (e.g., `fs.createReadStream()`, `http.IncomingMessage`).
- **Writable Streams:** For writing data (e.g., `fs.createWriteStream()`, `http.ServerResponse`).
- **Duplex Streams:** Both readable and writable (e.g., `net.Socket`).
- **Transform Streams:** Duplex streams that can modify data as it passes through (e.g., `zlib.createGzip()`).

Example (Using streams to copy a large file):

```
const fs = require('fs');
const readableStream = fs.createReadStream('large_input.txt');
const writableStream = fs.createWriteStream('large_output.txt');
readableStream.on('data', (chunk) => {
  // Process or write each chunk of data
  writableStream.write(chunk);
});
```

```
readableStream.on('end', () => {
   console.log('Finished reading file.');
   writableStream.end(); // Important to signal the end of writing
});

readableStream.on('error', (err) => {
   console.error('Error reading stream:', err);
});

writableStream.on('finish', () => {
   console.log('Finished writing file.');
});

writableStream.on('error', (err) => {
   console.error('Error writing stream:', err);
});
```

Asynchronous Programming in Node.js

Node.js is inherently asynchronous and non-blocking, which is crucial for its performance. This means that when an operation like reading a file or making a network request is initiated, Node.js doesn't wait for it to complete. Instead, it moves on to execute other code, and when the operation finishes, a callback function is invoked.

Key concepts in asynchronous programming in Node.js:

- **Event Loop:** The heart of Node.js's asynchronous nature. It continuously monitors the call stack and the message queue. When the call stack is empty, it picks up tasks from the message queue (which are often results of I/O operations or timers) and pushes them onto the call stack for execution.
- **Callbacks:** Functions that are passed as arguments to other functions and are executed once the asynchronous operation is complete.
- **Promises:** A more structured way to handle asynchronous operations, providing a cleaner alternative to nested callbacks.
- **Async/Await:** Syntactic sugar built on top of Promises, making asynchronous code look and behave more like synchronous code, improving readability.

Callbacks and Callback Hell

Callbacks: As mentioned, callbacks are fundamental to asynchronous operations in Node.js.

Example (simple callback):

```
function fetchData(callback) {
   setTimeout(() => {
      const data = 'Some data from the server';
      callback(null, data); // null for error, data for success
   }, 2000); // Simulate 2-second delay
}
console.log('Fetching data...');
fetchData((error, result) => {
   if (error) {
      console.error('Error:', error);
      return;
   }
   console.log('Data received:', result);
});
console.log('Continuing with other tasks...');
```

Callback Hell (or Pyramid of Doom): This occurs when you have multiple nested asynchronous operations, leading to deeply indented and difficult-to-read and maintain code. Error handling also becomes more complex.

Example of Callback Hell:

```
fs.readFile('file1.txt', 'utf8', (err, data1) => {
   if (err) throw err;
   console.log('File 1:', data1);
   fs.readFile('file2.txt', 'utf8', (err, data2) => {
      if (err) throw err;
      console.log('File 2:', data2);
      fs.readFile('file3.txt', 'utf8', (err, data3) => {
         if (err) throw err;
         console.log('File 3:', data3);       // And so on...
         fs.writeFile('combined.txt', `${data1}\n${data2}\n${data3}`, (err) => {
            if (err) throw err;
            console.log('Combined file written.');
         });
      });
   });
});
```

1.4 Promises and async/await,

Solutions to Callback Hell:

1. **Named Functions:** Break down the nested callbacks into named functions. This improves readability but doesn't eliminate nesting.
2. **Promises (Preferred for cleaner code):** Promises provide a more elegant way to handle asynchronous operations by chaining .then() calls and using .catch() for error handling.
3. **Async/Await (Most modern and readable):** Built on top of Promises, `async/await` allows you to write asynchronous code that looks and feels synchronous, making it much easier to read and debug.

Promises

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. It's a way to handle asynchronous operations more synchronously and to avoid "callback hell."

A Promise can be in one of three states:

- **Pending:** The initial state; neither fulfilled nor rejected.
- **Fulfilled (Resolved):** The operation completed successfully, and the Promise has a resulting value.
- **Rejected:** The operation failed, and the Promise has a reason for the failure (an error).

Creating a Promise:

You create a new Promise using the new Promise() constructor, which takes an executor function as an argument. The executor function itself takes two arguments: resolve and reject.

- resolve(value): Call this when the asynchronous operation is successful, passing the resulting value.
- reject(error): Call this when the asynchronous operation fails, passing an error object.

JavaScript

```
function simulateAsyncOperation(success) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (success) {
        resolve("Data successfully fetched!");
      } else {
        reject(new Error("Failed to fetch data."));
      }
    }, 1500); // Simulate an asynchronous task taking 1.5 seconds
  });
}
```

```
// Example usage:
console.log("Starting async operation (success case)...");
simulateAsyncOperation(true)
  .then((message) => {
    // This block runs if the Promise is resolved (successful)
    console.log("Success:", message);
  })
  .catch((error) => {
    // This block runs if the Promise is rejected (failed)
    console.error("Error:", error.message);
  });

console.log("\nStarting async operation (failure case)...");
simulateAsyncOperation(false)
  .then((message) => {
    console.log("Success:", message);
  })
  .catch((error) => {
    console.error("Error:", error.message);
  });

console.log("\nCode continues to run while promises are pending...");
```

Consuming a Promise:

- **.then(onFulfilled, onRejected):** The primary way to consume a Promise.
  - onFulfilled: A function that is called when the Promise is fulfilled. It receives the resolved value.
  - onRejected: (Optional) A function that is called when the Promise is rejected. It receives the rejection reason (error).
- **.catch(onRejected):** A shorthand for .then(null, onRejected). It's commonly used to handle errors at the end of a Promise chain.
- **.finally(onFinally):** (ES2018+) A function that is called when the Promise is settled (either fulfilled or rejected). It receives no arguments and is useful for cleanup operations regardless of the outcome.

Promise Chaining:

One of the biggest advantages of Promises is the ability to chain them. When a .then() callback returns a Promise, the next .then() in the chain will wait for that returned Promise to resolve before executing. This allows you to sequence asynchronous operations.

JavaScript

```
function step1() {
   return new Promise(resolve => {
      setTimeout(() => {
         console.log("Step 1 complete.");
         resolve(10);
      }, 1000);
   });
}

function step2(value) {
   return new Promise(resolve => {
      setTimeout(() => {
         console.log(`Step 2 complete with value: ${value + 5}`);
         resolve(value + 5);
      }, 800);
   });
}

function step3(value) {
   return new Promise(resolve => {
      setTimeout(() => {
         console.log(`Step 3 complete with value: ${value * 2}`);
         resolve(value * 2);
      }, 600);
   });
}

console.log("Starting promise chain...");
step1()
   .then(result1 => step2(result1)) // result1 from step1
   .then(result2 => step3(result2)) // result2 from step2
   .then(finalResult => {
      console.log("All steps complete. Final result:", finalResult);
   })
   .catch(error => {
      console.error("An error occurred in the chain:", error);
   });
```

Promise.all(), Promise.race(), Promise.allSettled(), Promise.any():

- **Promise.all(iterable):** Takes an iterable (e.g., an array) of Promises and returns a single Promise. This returned Promise:
  - Resolves when *all* of the input Promises have resolved. The resolved value is an array of the resolved values from the input Promises, in the same order.
  - Rejects as soon as *any* of the input Promises reject. The rejection reason is the reason of the first Promise that rejected.
- **Promise.race(iterable):** Takes an iterable of Promises and returns a single Promise. This returned Promise:
  - Resolves or rejects as soon as *any* of the input Promises resolve or reject. The resolved/rejected value is the value/reason of the first Promise that settled.
- **Promise.allSettled(iterable):** (ES2020+) Takes an iterable of Promises and returns a single Promise. This returned Promise resolves when *all* of the input Promises have settled (either resolved or rejected). The resolved value is an array of objects, each describing the outcome of a Promise (e.g., { status: 'fulfilled', value: ... } or { status: 'rejected', reason: ... }). Useful when you need to know the outcome of all Promises, even if some fail.
- **Promise.any(iterable):** (ES2021+) Takes an iterable of Promises and returns a single Promise. This returned Promise:
  - Resolves as soon as *any* of the input Promises fulfill. The resolved value is the value of the first Promise that fulfilled.
  - Rejects if *all* of the input Promises reject. The rejection reason is an AggregateError containing all the rejection reasons.

Async/Await

async and await are modern JavaScript syntax (ES2017+) built on top of Promises to make asynchronous code look and behave more like synchronous code, significantly improving readability and maintainability.

- **async keyword:**
  - You use async before a function declaration to denote that the function is asynchronous.
  - An async function always returns a Promise. If the function returns a non-Promise value, it's implicitly wrapped in a resolved Promise. If it throws an error, it implicitly returns a rejected Promise.
- **await keyword:**
  - You can only use await inside an async function.
  - await pauses the execution of the async function until the Promise it's waiting for settles (resolves or rejects).
  - If the Promise resolves, await returns the resolved value.
  - If the Promise rejects, await throws the rejected error, which you can catch using a try...catch block.

Example of async/await:

JavaScript

```
function fetchUserData(userId) {
   return new Promise((resolve, reject) => {
      setTimeout(() => {
         if (userId === 1) {
            resolve({ id: 1, name: 'Alice', email: 'alice@example.com' });
         } else if (userId === 2) {
            resolve({ id: 2, name: 'Bob', email: 'bob@example.com' });
         } else {
            reject(new Error(`User with ID ${userId} not found.`));
         }
      }, 1000);
   });
}

async function getUserAndDisplay(userId) {
   try {
      console.log(`Fetching user data for ID: ${userId}...`);
      const user = await fetchUserData(userId); // Pause here until fetchUserData resolves
      console.log(`User found: ${user.name} (${user.email})`);
   } catch (error) {
      console.error(`Error fetching user for ID ${userId}:`, error.message);
   }
}

console.log("Application started.");
getUserAndDisplay(1);
getUserAndDisplay(3);
console.log("Other operations continue executing while async functions are pending.");
```

async/await with try...catch for error handling:

The try...catch block is the standard way to handle errors with async/await. If an awaited Promise rejects, it acts like a synchronous throw, and the catch block will be executed.

Benefits of async/await:

- **Readability:** Asynchronous code looks almost identical to synchronous code, making it much easier to understand the flow.
- **Maintainability:** Easier to debug and reason about the sequence of operations.
- **Error Handling:** try...catch blocks work naturally, just like with synchronous code, simplifying error management compared to nested .catch() calls in Promise chains.
- **Debugging:** Stepping through async/await code in a debugger is often more straightforward than with raw Promises or callbacks.

1.5 Event Loop and EventEmitter

Now we are moving into the heart of Node.js's asynchronous nature! The Event Loop and EventEmitter are fundamental concepts for understanding how Node.js handles concurrent operations without blocking.

The Event Loop

The Event Loop is the most crucial concept in Node.js. It's the mechanism that allows Node.js to perform non-blocking I/O operations despite JavaScript being single-threaded. It delegates operations to the system kernel whenever possible.

How it Works (Simplified):

1. **Call Stack:** This is where your JavaScript code executes. When a function is called, it's pushed onto the stack. When it returns, it's popped off.
2. **Node.js APIs (Web APIs in browser context):** These are capabilities provided by the Node.js runtime (or browser environment) for performing operations that might take time (e.g., `setTimeout`, `fs.readFile`, network requests). When you call one of these, Node.js delegates the operation to the underlying system (C++ threads, OS kernel, etc.).
3. **Callback Queue (or Message Queue / Task Queue):** When an asynchronous operation delegated to a Node.js API completes (e.g., a file is read, a network request finishes), its associated callback function is placed into the Callback Queue.
4. **Event Loop:** This is the continuous loop that constantly checks two things:
    - Is the **Call Stack empty**? (Is there any synchronous code currently running?)
    - Is there any **callback** waiting in the **Callback Queue**?

    If the Call Stack is empty, the Event Loop takes the *first* callback from the Callback Queue and pushes it onto the Call Stack for execution. This process repeats indefinitely.

Phases of the Event Loop:

The Node.js Event Loop is more complex than a single queue. It operates in phases (or "ticks"), and each phase has its own queue of callbacks. This ordering is crucial for how different types of asynchronous operations are prioritized.

- **Timers (`setTimeout()`, `setInterval()`):** Execute callbacks scheduled by `setTimeout()` and `setInterval()`.
- **Pending Callbacks:** Executes I/O callbacks deferred until the next loop iteration (e.g., some system errors, TCP errors).
- **Idle, Prepare:** Used internally by Node.js.
- **Poll:**
    - Retrieves new I/O events (e.g., completed file reads, network connections).
    - Executes I/O callbacks (most `fs` and `net` callbacks are executed here).

    - If there are no pending I/O events, it might block here briefly, waiting for new events.
    - **Crucially, if `setImmediate()` callbacks are queued, the event loop will exit the poll phase and proceed to the `check` phase.**
- **Check (`setImmediate()`):** Executes callbacks scheduled by `setImmediate()`.
- **Close Callbacks:** Executes `close` event callbacks (e.g., `socket.on('close', ...)`, `server.on('close', ...)`)

`process.nextTick()` and `Promise.resolve().then()` (Microtask Queue):
These are *not* part of the Event Loop phases directly. They belong to a separate, higher-priority queue called the **Microtask Queue** (or Job Queue).

- `process.nextTick()` callbacks are executed *immediately* after the current operation on the Call Stack completes, but *before* the Event Loop moves to the next phase.
- Resolved Promises (`.then()`, `await`) are also placed in the Microtask Queue.

Order of Execution:

1. **Current Synchronous Code:** Everything on the Call Stack.
2. **`process.nextTick()` callbacks:** All pending `nextTick` callbacks are executed.
3. **Microtask Queue (Promises):** All pending Promise callbacks are executed.
4. **Event Loop Phases:** The loop proceeds through `timers`, `pending callbacks`, `poll`, `check`, `close callbacks`. After each phase, `process.nextTick()` and Microtask queues are checked again before moving to the *next* Event Loop phase.

Why is this important?

Understanding the Event Loop is vital for:

- **Predicting execution order:** Knowing when your asynchronous code will run.
- **Avoiding blocking operations:** Recognizing that long-running synchronous code will "block" the Event Loop, preventing other callbacks from executing.
- **Optimizing performance:** Designing applications that leverage non-blocking I/O effectively.

Example Illustrating Event Loop Order:

```
console.log('1. Start');
setTimeout(() => {
   console.log('4. setTimeout callback (Timer phase)');
}, 0); // Minimum delay, but still enters timer phase

setImmediate(() => {
   console.log('5. setImmediate callback (Check phase)');
});

Promise.resolve().then(() => {
   console.log('3. Promise.then (Microtask Queue)');
});
```

```
process.nextTick(() => {
   console.log('2. process.nextTick (NextTick Queue)');
});

const fs = require('fs');
fs.readFile(__filename, () => { // __filename is the current file path
   console.log('6. fs.readFile callback (Poll phase)');
});

console.log('1. End (Synchronous code)');
```

Expected Output (may vary slightly based on environment/timing, but general order is consistent):

```
1. Start
1. End (Synchronous code)
2. process.nextTick (NextTick Queue)
3. Promise.then (Microtask Queue)
4. setTimeout callback (Timer phase)
5. setImmediate callback (Check phase)
6. fs.readFile callback (Poll phase)
```

EventEmitter

The `EventEmitter` is a class in Node.js that provides an implementation of the **Observer pattern** (also known as the Publisher/Subscriber pattern). It allows you to create custom events and then emit (trigger) these events, which will then execute all registered listener functions.

Many of Node.js's core modules (like `fs.ReadStream`, `http.Server`, `net.Socket`) extend `EventEmitter` to signal changes in state or completion of operations.

Key Methods:

- **`emitter.on(eventName, listener)`:** Registers a `listener` function to be called whenever `eventName` is emitted. (Alias: `addListener`)
- **`emitter.emit(eventName, [arg1], [arg2], ...)`:** Synchronously calls all registered listeners for the specified `eventName`, passing any additional arguments to them.
- **`emitter.once(eventName, listener)`:** Registers a `listener` that will be invoked only once for `eventName`, and then automatically removed.
- **`emitter.removeListener(eventName, listener)`:** Removes a specific `listener` function from the event `eventName`.
- **`emitter.removeAllListeners([eventName])`:** Removes all listeners for a specific `eventName`, or all listeners for all events if no `eventName` is provided.

Example of Custom EventEmitter:

```
const EventEmitter = require('events');
class MyCustomEmitter extends EventEmitter {}
const myEmitter = new MyCustomEmitter();

// Register listeners for 'greet' event
myEmitter.on('greet', (name) => {
   console.log(`Hello, ${name}!`);
});

myEmitter.on('greet', (name) => {
   console.log(`Nice to see you, ${name}.`);
});

// Register a listener for 'alarm' event that runs once
myEmitter.once('alarm', (message) => {
   console.log(`Alarm: ${message}`);
});

// Register an error listener (important for error events!)
myEmitter.on('error', (err) => {
   console.error('Whoops! There was an error:', err.message);
});

// Emit events
console.log('Emitting "greet" event...');
myEmitter.emit('greet', 'Alice'); // Both 'greet' listeners will be called
console.log('---');

console.log('Emitting "alarm" event...');
myEmitter.emit('alarm', 'Time to wake up!'); // This listener runs
myEmitter.emit('alarm', 'Another alarm!');   // This listener will NOT run (once)
console.log('---');

// Emit an error event
myEmitter.emit('error', new Error('Something went wrong during operation X.'));
console.log('---');

// Removing a specific listener
const specificListener = (data) => console.log('Specific data received:', data);
myEmitter.on('dataReceived', specificListener);
myEmitter.emit('dataReceived', { value: 100 });
myEmitter.removeListener('dataReceived', specificListener);
myEmitter.emit('dataReceived', { value: 200 }); // This will not log, listener removed
```

```
// Handling a common error: emitting an 'error' event without a listener
// If you emit an 'error' event and there are no listeners for it, Node.js will
// throw an uncaught error and likely crash the process.
// To demonstrate (uncomment to see crash):
// myEmitter.removeAllListeners('error'); // Remove the existing error handler
// myEmitter.emit('error', new Error('This will crash if no error listener!'));
```

Why use EventEmitter?

- **Decoupling:** It decouples the publisher (who emits the event) from the subscribers (who listen to the event), making your code more modular and easier to manage.
- **Event-driven Architecture:** It's fundamental to building event-driven applications, where different parts of your system react to events happening elsewhere.
- **Asynchronous Nature:** While `emit` itself is synchronous, the actions taken by listeners might be asynchronous, fitting well with Node.js's paradigm.
- **Core Node.js Usage:** Understanding EventEmitter helps you work effectively with many built-in Node.js modules that use this pattern.

1.6 Creating a Basic HTTP Server, Using http module to create a server, Handling requests and responses, Serving static files

Let's delve into creating a basic HTTP server in Node.js, which is a fundamental skill for building web applications and APIs.

Creating a Basic HTTP Server

Node.js provides the built-in `http` module, which allows you to create HTTP servers and clients without needing any external libraries. This module is low-level, giving you fine-grained control over the request and response cycle.

Using the http module to create a server
The core of creating an HTTP server involves:

1. **Requiring the `http` module:** `const http = require('http');`
2. **Creating a server instance:** `http.createServer(requestListener);`
    - The `requestListener` is a callback function that executes every time the server receives an HTTP request. It takes two arguments: `request` (often `req`) and `response` (often `res`).
3. **Starting the server to listen for requests:** `server.listen(port, [hostname], [callback]);`
    - `port`: The port number the server will listen on (e.g., 3000, 8080).
    - `hostname` (optional): The IP address or hostname to bind to (e.g., `'127.0.0.1'` for localhost). If omitted, it defaults to `0.0.0.0` (listens on all available network interfaces).
    - `callback` (optional): A function that is called once the server starts listening.

Basic Server Example:

```
const http = require('http');
const hostname = '127.0.0.1'; // Localhost
const port = 3000;
// The requestListener function
const server = http.createServer((req, res) => {
   // Set the HTTP status code and Content-Type header
   res.statusCode = 200; // OK
   res.setHeader('Content-Type', 'text/plain'); // Tell the client it's plain text
   // Send the response body and end the response
   res.end('Hello, Node.js HTTP Server!\n');
});
// Start the server
server.listen(port, hostname, () => {
   console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this:

1. Save it as `server.js`.
2. Open your terminal/command prompt.
3. Navigate to the directory where you saved the file.
4. Run: `node server.js`
5. Open your web browser and go to `http://localhost:3000/`.

   You should see "Hello, Node.js HTTP Server!".

Handling requests and responses

The `req` (request) and `res` (response) objects are central to handling HTTP communication.

`req` (IncomingMessage object): The `req` object provides information about the incoming HTTP request.

- `req.url`: The URL path of the request (e.g., `/`, `/about`, `/api/users`).
- `req.method`: The HTTP method (e.g., `GET`, `POST`, `PUT`, `DELETE`).
- `req.headers`: An object containing the request headers.
- `req.body`: For `POST` or `PUT` requests, the body data needs to be collected from the request stream (as `req` is a Readable Stream).

`res` (ServerResponse object): The `res` object is used to send data back to the client.

- `res.statusCode`: Sets the HTTP status code (e.g., 200 for OK, 404 for Not Found, 500 for Internal Server Error).
- `res.setHeader(name, value)`: Sets a single HTTP response header.
- `res.writeHead(statusCode, [statusMessage], [headers])`: A more convenient way to set status code and multiple headers at once.
- `res.write(chunk)`: Writes a chunk of the response body. Can be called multiple times.
- `res.end([data], [encoding], [callback])`: Signals that all of the response headers and body have been sent; this method *must* be called on each response. It can optionally send the last chunk of data.

Example with Request/Response Handling and Basic Routing:

```
const http = require('http');
const server = http.createServer((req, res) => {
    // Log request details
    console.log(`Request received: ${req.method} ${req.url}`);
    // Basic routing based on URL
    if (req.url === '/') {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end('<h1>Welcome to the Home Page!</h1><p>Try navigating to /about or
/api/data</p>');
    } else if (req.url === '/about') {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end('<h1>About Us</h1><p>This is a simple Node.js HTTP server example.</p>');
    } else if (req.url === '/api/data' && req.method === 'GET') {
        res.writeHead(200, { 'Content-Type': 'application/json' });
        const data = {
            message: 'This is some JSON data',
            timestamp: new Date().toISOString(),
            items: ['item1', 'item2', 'item3']
        };
        res.end(JSON.stringify(data)); // Send JSON data
    } else if (req.url === '/submit' && req.method === 'POST') {
        let body = '';
        req.on('data', chunk => {
            body += chunk.toString(); // Collect data chunks
        });
        req.on('end', () => {
            console.log('Received POST data:', body);
            res.writeHead(200, { 'Content-Type': 'text/plain' });
            res.end(`Data received: ${body}`);
        });
    } else {
        // Handle 404 Not Found
        res.writeHead(404, { 'Content-Type': 'text/html' });
        res.end('<h1>404 Not Found</h1><p>The page you requested does not exist.</p>');
```

```
    }
});

server.listen(3000, '127.0.0.1', () => {
    console.log('Server running at http://127.0.0.1:3000/');
});
```

Serving Static Files

Serving static files (HTML, CSS, JavaScript, images) is a common task for web servers. You'll typically use the `fs` (File System) module to read the files and the `path` module to construct safe file paths.

Key considerations:

- **File Path Resolution:** Ensure you construct file paths correctly and securely (prevent directory traversal attacks).
- **Content-Type Header:** Set the appropriate `Content-Type` header so the browser knows how to interpret the file (e.g., `text/html`, `text/css`, `image/jpeg`).
- **Error Handling:** Gracefully handle cases where the file doesn't exist (404 Not Found).
- **Streams for Large Files:** For very large files, it's more efficient to use `fs.createReadStream()` and `pipe()` to the response, rather than `fs.readFile()` which loads the entire file into memory.

Example: Serving Static Files

First, create a `public` directory in the same location as your `server.js` and add some files:

`public/index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Static Page</title>
    <link rel="stylesheet" href="/style.css">
</head>
<body>
    <h1>Hello from Static HTML!</h1>
    <p>This is served directly from the public directory.</p>
    <img src="/node_logo.png" alt="Node.js Logo" width="100">
    <script src="/script.js"></script>
</body>
</html>
```

```public/style.css```:

```css
```
```

```
body {
    font-family: sans-serif;
    background-color: #f0f0f0;
    color: #333;
    text-align: center;
    padding: 20px;
}
h1 {
    color: #4CAF50;
}
```
```public/script.js`:
```javascript
console.log('Script loaded from static file!');
document.addEventListener('DOMContentLoaded', () => {
    const p = document.querySelector('p');
    if (p) {
        p.textContent += ' (JavaScript added this!)';
    }
});
```
You can download a small `node_logo.png` and place it in the `public` directory, or use a placeholder like `https://placehold.co/100x100/A0A0A0/FFFFFF?text=Logo` if you prefer.

Now, modify your `server.js`

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

const hostname = '127.0.0.1';
const port = 3000;
const publicDirectory = path.join(__dirname, 'public'); // Path to your static files

// Helper function to get Content-Type based on file extension
const getContentType = (filePath) => {
    const extname = String(path.extname(filePath)).toLowerCase();
    const mimeTypes = {
        '.html': 'text/html',
        '.js': 'text/javascript',
        '.css': 'text/css',
        '.json': 'application/json',
        '.png': 'image/png',
        '.jpg': 'image/jpg',
        '.gif': 'image/gif',
        '.svg': 'image/svg+xml',
        '.ico': 'image/x-icon'
    };
    return mimeTypes[extname] || 'application/octet-stream';
};
```
```

```
const server = http.createServer((req, res) => {
    console.log(`Request received: ${req.method} ${req.url}`);

    let filePath = req.url === '/' ? '/index.html' : req.url; // Default to index.html for root
    let fullPath = path.join(publicDirectory, filePath);

    // Security check: Prevent directory traversal
    if (!fullPath.startsWith(publicDirectory)) {
        res.writeHead(403, { 'Content-Type': 'text/plain' });
        res.end('403 Forbidden: Access denied.\n');
        return;
    }

    // Check if the file exists
    fs.stat(fullPath, (err, stats) => {
        if (err) {
            if (err.code === 'ENOENT') {
                // File not found
                res.writeHead(404, { 'Content-Type': 'text/html' });
                res.end('<h1>404 Not Found</h1><p>The requested file could not be found.</p>');
            } else {
                // Other server errors
                res.writeHead(500, { 'Content-Type': 'text/html' });
                res.end(`<h1>500 Internal Server Error</h1><p>${err.message}</p>`);
            }
            return;
        }

        // If it's a directory, try to serve index.html inside it
        if (stats.isDirectory()) {
            fullPath = path.join(fullPath, 'index.html');
            // Re-check if index.html exists in the directory
            fs.stat(fullPath, (err, indexStats) => {
                if (err) {
                    res.writeHead(404, { 'Content-Type': 'text/html' });
                    res.end('<h1>404 Not Found</h1><p>Directory listing is not allowed, and
index.html not found.</p>');
                    return;
                }
                serveFile(fullPath, res);
            });
        } else {
            // Serve the file
            serveFile(fullPath, res);
        }
    });
});
```

```
// Helper function to serve the file
function serveFile(fullPath, res) {
    const contentType = getContentType(fullPath);

    // Use createReadStream for efficiency, especially for larger files
    const readStream = fs.createReadStream(fullPath);

    readStream.on('open', () => {
        res.writeHead(200, { 'Content-Type': contentType });
        readStream.pipe(res); // Pipe the file content directly to the response
    });

    readStream.on('error', (err) => {
        console.error('Error reading file stream:', err);
        res.writeHead(500, { 'Content-Type': 'text/html' });
        res.end(`<h1>500 Internal Server Error</h1><p>Could not read file.</p>`);
    });
}
server.listen(port, hostname, () => {
    console.log(`Static server running at http://${hostname}:${port}/`);
});
```

1.7 Introduction to Package Management with npm, Installing, Updating, and Removing Packages, Using package.json and package-lock.json, npm Packages mongoose, express, react, cors.

1.7.1 Introduction to Package Management with npm

npm, or **Node Package Manager**, is the default package manager for Node.js. It's a command-line tool that lets you manage the external libraries and packages your project depends on. Think of it as a central hub where developers can share and reuse code, making it incredibly easy to add powerful functionalities to your applications without having to write them from scratch.

1.7.2 Installing, Updating, and Removing Packages

Using npm is straightforward. Here are the basic commands:

- **Installing**: To install a package, you use npm install <package-name>. For example, npm install express. This command downloads the package and its dependencies and places them in a folder called node_modules.
- **Updating**: To update an existing package to its latest version, you use npm update <package-name>. This is useful for getting the latest features or security patches.
- **Removing**: To uninstall a package you no longer need, you use npm uninstall <package-name>.

1.7.3 Using package.json and package-lock.json

- **package.json**: This file is at the heart of every npm project. It's a manifest that contains metadata about your project, such as its name, version, and scripts. Most importantly, it lists all the project's dependencies under the dependencies and devDependencies sections. When you share your project, you only share this file, and other developers can run npm install to get all the necessary packages.
- **package-lock.json**: This file is automatically generated and updated whenever npm modifies your node_modules folder. It locks the exact versions of every installed package and its dependencies. This ensures that every developer working on the project, and even the production environment, uses the **exact same package versions**, preventing unexpected bugs caused by version mismatches.

1.7.4 Common npm Packages

Here are some popular and essential npm packages:

- **mongoose**: This package is an **Object Data Modeling (ODM)** library for MongoDB. It provides a straightforward, schema-based solution to model your application data, allowing you to easily interact with your database.
- **express**: A minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It simplifies tasks like routing and handling HTTP requests.
- **react**: While technically a JavaScript library for building user interfaces, it's often managed with npm. It allows developers to create reusable UI components and manage the state of their application efficiently.
- **cors**: This package provides a Node.js middleware for handling **Cross-Origin Resource Sharing (CORS)**. It enables a server to specify who can access its resources from a different domain, which is crucial for building APIs that are consumed by web applications on different origins.

Question Bank

6 Marks Questions

1. Elaborate on the **Event Loop** in Node.js. Explain how it works with the call stack, the callback queue, and the Node.js API to enable non-blocking I/O.
2. Describe how a server handle multiple requests concurrently without using multiple threads.
3. You are tasked with processing a large CSV file. Explain why using a **Readable stream** is a more efficient approach than using fs.readFile(). Provide a Node.js code example that reads a file named large_data.csv using a stream and pipes its contents to a Writable stream that saves it to a new file named processed_data.csv.

5 Marks Questions

1. Differentiate between **Core Modules** and **Custom Modules** in Node.js.
2. Explain the problem of **"Callback Hell"** that can arise with asynchronous programming and demonstrate how **Promises** or **async/await** provide a cleaner, more readable solution.
3. Create a Node.js application that uses the http module to serve a simple web page. The server should listen on port 8080. When a user navigates to the root URL (/), it should respond with an index.html file.

4 Marks Questions

1. Explain the purpose of the **package.json** and **package-lock.json** files.
2. Write a Node.js script that uses the fs module's asynchronous methods (fs.readFile and fs.writeFile) to read the content of input.txt and then write that content to a new file called output.txt. Include error handling in your code.