CONTENTS

NODE JS DEVELOPMENT GUIDE

In modern web application development, Node.js and Express.js are fundamental technologies used to build fast, scalable applications. Real-time communication is another important aspect of many applications, and Node.js, with the help of specific modules, makes it possible to implement real-time features easily.

Below is an explanation of each of these technologies and the modules needed to implement real-time features in a Node.js project.

- **Node.js** is an open-source, cross-platform runtime environment that allows developers to execute JavaScript code on the server side.
- It uses **V8**, the JavaScript engine from Chrome, to execute code efficiently.
- **Non-blocking, event-driven I/O** makes Node.js highly suitable for I/O-heavy operations like web servers, APIs, and real-time applications.

Key Features of Node.js:

- **Single-threaded Event Loop**: Handles concurrent operations without multi-threading, allowing it to process many connections simultaneously.
- **Asynchronous and Non-blocking**: Non-blocking I/O allows Node.js to handle numerous requests concurrently, making it very fast.
- **Extensive Package Ecosystem**: With npm (Node Package Manager), Node.js has access to a large number of libraries and modules for various functionalities, including real-time communication, database integration, and more.

Use Case:

- Node.js is ideal for applications that require fast, scalable, and I/O-intensive operations like APIs, real-time apps, and single-page applications (SPAs).

What is Express.js?

- **Express.js** is a minimalist and flexible web application framework built on top of Node.js.
- It simplifies the process of building APIs and web servers, providing helpful abstractions over Node's core HTTP module.
- Express.js handles routing, middleware integration, and request/response handling.

Key Features of Express.js:

- **Routing**: Simplifies URL routing and HTTP method handling (GET, POST, PUT, DELETE).
- **Middleware Support**: Express allows you to use middleware functions that can execute code, modify the request or response objects, and pass control to the next middleware.
- **Template Engines**: It supports various templating engines like EJS, Pug, and Handlebars for rendering dynamic content.
- **Error Handling**: Provides an easy way to manage errors in your application.

Use Case:

- Express.js is used to build web servers, RESTful APIs, and backend logic for web applications. It is often used in conjunction with front-end frameworks to develop full-stack applications.

1. IDES SETUP AND CONFIGURATION

Concept:

- **IDE (Integrated Development Environment)** is essential for developers to write, test, and debug their code.
- Popular IDEs for Node.js include **Visual Studio Code (VSCode)**, **WebStorm**, and **Sublime Text**.

Setup:

1. **VSCode**: One of the most popular IDEs for Node.js development.
   - **Installation**: Download VSCode from here.
   - **Extensions**:
     - **Node.js Extension Pack** (includes ESLint, Prettier, Debugger for Node.js, etc.)
     - **npm Intellisense** for autocompletion of npm modules.
     - **Debugger for Chrome** to debug Node.js code.
2. **WebStorm**: A commercial IDE from JetBrains with built-in Node.js support.
   - **Installation**: Download WebStorm from here.
   - It offers built-in Node.js debugging, profiling, and testing features.

Configuration:

- **VSCode Settings**: Configure workspace settings, such as enabling auto-formatting, configuring linting rules, or integrating Git.
- **Debug Configuration**: Set up launch.json in .vscode to specify how to run and debug Node.js applications.

Use Case:

- Developing a simple "To-Do List" app in Node.js using Express.js and MongoDB.

Interview Questions:

1. What are some common IDEs for Node.js development?
2. How do you configure a debugger for Node.js in VSCode?

Coding Script:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            "skipFiles": ["<node_internals>/**"],
            "program": "${workspaceFolder}/app.js"
        }
    ]
}
```

2. NPM AND YARN COMMANDS

Concept:

- **npm** (Node Package Manager) is the default package manager for Node.js.
- **Yarn** is a faster alternative to npm with better dependency management.

Commands for NPM:

1. npm init: Initializes a new Node.js project.
2. npm install <package>: Installs a package.
3. npm install --save <package>: Installs and adds the package to package.json.
4. npm update: Updates all packages.
5. npm run <script>: Runs a script defined in package.json.

Commands for Yarn:

1. yarn init: Initializes a new Node.js project.
2. yarn add <package>: Installs a package and adds it to package.json.
3. yarn upgrade: Upgrades all dependencies.
4. yarn run <script>: Runs a script defined in package.json.

Use Case:

- Managing dependencies for a project like an API server built using Express.js.

Interview Questions:

1. What is the difference between npm and yarn?
2. How do you install a package using npm and yarn?

Coding Script:

```
# Installing dependencies
npm install express --save  # Using npm

yarn add express  # Using yarn
```

3. WHAT IS NODE.JS?

Concept:

- **Node.js** is a JavaScript runtime built on Chrome's V8 engine, enabling the execution of JavaScript code on the server-side.
- **Non-blocking, event-driven architecture** makes Node.js suitable for building scalable network applications.

Key Features:

- **Asynchronous I/O**: Node.js uses non-blocking, event-driven I/O operations.
- **Single-threaded Event Loop**: Handles concurrent requests with a single thread.

Use Case:

- Real-time chat applications, RESTful APIs, file servers.

Interview Questions:

1. What is Node.js, and why is it used in backend development?
2. What are the key features of Node.js that make it efficient for building scalable apps?

Coding Script:

```
// Simple Node.js server

const http = require('http');

const server = http.createServer((req, res) => {

  res.write('Hello, World!');

  res.end();

});

server.listen(3000, () => {

  console.log('Server running on port 3000');

});
```

4. NODE.JS ARCHITECTURE

Concept:

- Node.js follows a **single-threaded, event-driven model** for handling concurrent requests.
- It uses the **event loop** to handle multiple requests in a non-blocking manner.

Key Components:

1. **V8 Engine**: Compiles JavaScript into machine code.
2. **Libuv**: Handles I/O operations.
3. **Event Loop**: Manages asynchronous events and operations.

Use Case:

- Building a real-time chat application with multiple users without blocking the thread.

Interview Questions:

1. What is the event loop in Node.js, and how does it work?
2. How does Node.js handle concurrency?

Coding Script:

```
// Event-driven example in Node.js
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('data', () => {
  console.log('Data received!');
});

setTimeout(() => {
  emitter.emit('data');  // Emitting event
}, 1000);
```

5. NPM AND MODULES

Concept:

- **NPM Modules**: Reusable pieces of code or libraries that can be imported and used in Node.js projects.
- Node.js modules are divided into two categories:
    1. **Core Modules**: Built-in (e.g., fs, http).
    2. **External Modules**: Installed via npm.

Example of Module Usage:

```
// Using the fs module to read a file
const fs = require('fs');
fs.readFile('sample.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Use Case:

- Reading files from disk and serving them in a web application.

Interview Questions:

1. How do you create and export a custom module in Node.js?
2. Explain how you would use an external package like express in a Node.js project.

Coding Script:

```
// Exporting a custom module
module.exports.greet = () => {
  console.log('Hello from custom module!');
};

// Importing the custom module
const greet = require('./greet');
greet.greet();
```

6. EXPRESS.JS SETUP, FEATURES, AND PROJECT CONFIGURATION

Concept:

- **Express.js** is a lightweight, flexible, and minimalist web framework for Node.js that simplifies the creation of web servers and APIs.

Features:

1. **Routing**: Define routes for handling HTTP requests.
2. **Middleware**: Handle requests before they reach the route handler.
3. **Template Engines**: Use engines like EJS or Pug for rendering dynamic HTML.

Project Configuration:

1. **Installing Express**: npm install express --save
2. **Creating a Basic Server**:

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

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

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

Use Case:

- Developing a RESTful API that handles user authentication.

Interview Questions:

1. How do you set up a simple Express.js server?
2. What are middleware functions in Express.js?

Coding Script:

```
// Middleware example in Express.js
app.use((req, res, next) => {
  console.log('Request received');
  next();  // Pass the request to the next middleware
});
```

7. ASYNCHRONOUS PROGRAMMING

Concept:

- **Asynchronous programming** allows non-blocking code execution, making applications faster and more scalable.
- **Callbacks**, **Promises**, and **async/await** are used to handle asynchronous tasks in Node.js.

Key Features:

1. **Callbacks**: Functions that are passed as arguments to other functions and executed after a task is complete.
2. **Promises**: Represent the eventual completion or failure of an asynchronous operation.
3. **Async/Await**: Modern way to write asynchronous code that looks synchronous.

Use Case:

- Reading a file asynchronously using Node.js.

Interview Questions:

1. What is a callback function? Give an example.
2. How does async/await improve asynchronous programming?

Coding Script:

```
// Example of using Promises
const fs = require('fs').promises;

fs.readFile('example.txt', 'utf8')
  .then(data => console.log(data))
  .catch(err => console.log(err));

// Example of using async/await
async function readFile() {
 try {
    const data = await fs.readFile('example.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.log(err);
  }
}
readFile();
```

8. EVENT LOOP

Concept:

- **Event Loop** is a core part of Node.js's non-blocking, asynchronous architecture.
- It enables Node.js to handle many operations concurrently, without waiting for I/O operations to complete, by utilizing a **single thread** to manage multiple events.

Key Phases:

1. **Timers Phase**: Executes scheduled timers via setTimeout or setInterval.
2. **I/O Callbacks**: Executes callbacks for I/O tasks such as file reading.
3. **Poll Phase**: Checks for new events and executes their callbacks.
4. **Check Phase**: Executes callbacks scheduled by setImmediate.
5. **Close Callbacks**: Executes callbacks like socket.on('close', ...).

Use Case:

- Handling multiple HTTP requests concurrently without blocking the main thread.

Interview Questions:

1. Explain the concept of the event loop in Node.js.
2. How does the event loop handle asynchronous tasks and I/O operations?

Coding Script:

```
// Example of setImmediate and setTimeout
console.log('Start');
setTimeout(() => {
  console.log('setTimeout executed');
}, 0);

setImmediate(() => {
  console.log('setImmediate executed');
});
console.log('End');
Output:
Start
End
setImmediate executed
setTimeout executed
```

9. ROUTING ARCHITECTURE WITH EMBEDDED AND EXTERNAL FUNCTIONS

Concept:

- **Routing** refers to the process of defining the paths and handlers for HTTP requests.
- **Embedded Functions** are handlers directly within the route definition.
- **External Functions** are modularized handlers stored in separate files.

Key Concepts:

1. **Embedded Functions**: Routes and handlers defined in the same file.
2. **External Functions**: Routes and handlers in separate modules for better code organization and reusability.

Use Case:

- Building a modular Express.js app where routes are defined in separate files.

Interview Questions:

1. What are embedded and external functions in routing? How are they different?
2. How would you define a route handler in an external file?

Coding Script:

```
// Embedded Route Function
app.get('/user', (req, res) => {
  res.send('User Info');
});

// External Route Function (userRoutes.js)
const express = require('express');
const router = express.Router();

router.get('/user', (req, res) => {
 res.send('User Info');
});

module.exports = router;

// In main app file (app.js)
const userRoutes = require('./userRoutes');
app.use(userRoutes);
```

10. ROUTER VALIDATION

Concept:

- **Router Validation** ensures that incoming requests meet certain criteria before passing them to the route handler.
- Validations can include checking query parameters, body content, and headers.

Techniques:

1. **Middleware for Validation**: Use middleware to validate request data.
2. **Libraries**: Libraries like **Joi**, **express-validator** help in validating incoming data.

Use Case:

- Validate user input before processing in a registration API.

Interview Questions:

1. How do you validate request data in Express.js?
2. What are some commonly used libraries for validation in Node.js?

Coding Script:

```
// Using express-validator for request body validation
const { body, validationResult } = require('express-validator');

app.post('/user', [
  body('email').isEmail(),
  body('password').isLength({ min: 5 })
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.send('User created');
});
```

11. HTTP STATUS CODES

Concept:

- **HTTP Status Codes** are returned by the server to indicate the result of a client's request. They are divided into categories:
  - **1xx**: Informational
  - **2xx**: Success (e.g., 200 OK)
  - **3xx**: Redirection (e.g., 301 Moved Permanently)
  - **4xx**: Client errors (e.g., 400 Bad Request)
  - **5xx**: Server errors (e.g., 500 Internal Server Error)

Key Status Codes:

- 200 OK: Request has succeeded.
- 201 Created: Resource was created successfully.
- 400 Bad Request: The request is malformed.
- 404 Not Found: The resource could not be found.
- 500 Internal Server Error: Server-side error.

Use Case:

- Returning appropriate HTTP status codes based on whether an API request is successful or not.

Interview Questions:

1. What does the HTTP status code 404 indicate?
2. When would you return a 500 status code in a Node.js API?

Coding Script:

```
app.get('/user/:id', (req, res) => {
  const user = getUserById(req.params.id);
  if (!user) {
    return res.status(404).json({ message: 'User not found' });
  }
  res.status(200).json(user);
});
```

12. EXPRESS.JS ARCHITECTURE

Concept:

- **Express.js Architecture** is based on middleware and routing.
  - **Middleware** functions process requests and responses.
  - **Routing** directs HTTP requests to appropriate handlers.

Key Components:

1. **Middleware**: Functions executed during the request-response cycle (e.g., for logging, authentication).
2. **Routing**: Defines routes for different HTTP methods (GET, POST, PUT, DELETE).
3. **Request/Response Objects**: Represents the HTTP request and response.

Use Case:

- Organizing a web app with authentication middleware and API route handlers.

Interview Questions:

1. Explain the architecture of an Express.js application.
2. What is the role of middleware in Express.js?

Coding Script:

```
// Example of middleware
app.use((req, res, next) => {
  console.log(`Request made to: ${req.url}`);
  next();  // Proceed to the next middleware or route handler
});
```

13. ROUTING AND MIDDLEWARE

Concept:

- **Routing**: Handles incoming HTTP requests and maps them to specific logic.
- **Middleware**: Preprocesses requests before they reach the route handler (e.g., for logging, security, validation).

Types of Middleware:

1. **Application-level middleware**: Bound to an instance of the app (app.use()).
2. **Router-level middleware**: Bound to an instance of express.Router().
3. **Built-in middleware**: Provided by Express (e.g., express.json() for JSON body parsing).
4. **Third-party middleware**: Installed libraries like morgan (for logging).

Use Case:

- Using middleware to check if a user is authenticated before accessing sensitive routes.

Interview Questions:

1. What are middleware functions in Express.js, and how do they work?
2. How do you define a route in Express.js?

Coding Script:

```
// Using built-in middleware (JSON parser)
app.use(express.json());


// Define a route with custom middleware
app.post('/login', authenticateUser, (req, res) => {
  res.send('User authenticated');
});
```

14. HANDLING REQUESTS AND RESPONSES

Concept:

- Handling **HTTP requests** and **responses** in Express.js is done through route handlers.
- The request object (req) contains information about the HTTP request (e.g., body, query params), while the response object (res) is used to send a response back.

Use Case:

- Handling form submissions and sending back JSON responses.

Interview Questions:

1. How do you access the request body in Express.js?
2. How can you send a JSON response in Express.js?

Coding Script:

```
// Handling POST requests with JSON response

app.post('/user', (req, res) => {

  const userData = req.body;

  res.status(201).json({ message: 'User created', userData });

});
```

15. ERROR HANDLING IN EXPRESS.JS

Concept:

- Proper **error handling** ensures that unexpected issues in your application are handled gracefully and appropriate HTTP status codes are returned.

Techniques:

1. **Error-handling Middleware**: A middleware that catches errors passed using next().
2. **Custom Error Classes**: Define custom error classes for more structured error handling.

Use Case:

- Handling a database error when a record is not found.

Interview Questions:

1. How does error handling work in Express.js?
2. How do you create custom error classes in Node.js?

Coding Script:

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

// Example of throwing a custom error
function getUser(id) {
  const user = users.find(u => u.id === id);
  if (!user) {
    throw new Error('User not found');
  }
  return user;
}
```

16. TEMPLATING ENGINES (EJS, PUG)

Concept:

- **Templating engines** allow dynamic generation of HTML pages based on data.
- **EJS** (Embedded JavaScript) and **Pug** (formerly Jade) are popular templating engines in Express.js.

Use Case:

- Render dynamic HTML views for a user profile page using EJS.

Interview Questions:

1. What is the difference between EJS and Pug?
2. How do you use EJS for rendering dynamic views in Express.js?

Coding Script:

```
// Setting up EJS in Express.js
app.set('view engine', 'ejs');

app.get('/profile', (req, res) => {
  const user = { name: 'John', age: 30 };
  res.render('profile', { user });
});
```

17. RESTFUL API DESIGN

Concept:

- **RESTful API** design follows the principles of Representational State Transfer (REST) architecture.
- It uses standard HTTP methods (GET, POST, PUT, DELETE) to create, read, update, and delete resources.

Key Principles:

1. **Stateless**: Each request from the client to the server must contain all the necessary information to understand and process the request.
2. **Client-Server Architecture**: Separation of concerns between client and server.
3. **Cacheable**: Responses should indicate if they can be cached or not.
4. **Uniform Interface**: Consistent and standard approach to interactions between the client and server.
5. **Resource-Based**: The API exposes resources, which can be represented as URIs (Uniform Resource Identifiers).

Use Case:

- Designing an API for managing a collection of books, where each book has attributes such as title, author, and publishedYear.

Interview Questions:

1. What are the principles of RESTful API design?
2. How do you ensure that a RESTful API is stateless?

Coding Script:

```javascript
// Example of a RESTful API using Express.js

// Get a list of books (GET)
app.get('/books', (req, res) => {
  const books = [
    { title: 'Book 1', author: 'Author 1' },
    { title: 'Book 2', author: 'Author 2' }
  ];
  res.status(200).json(books);
});

// Add a new book (POST)
app.post('/books', (req, res) => {
  const { title, author } = req.body;
  const newBook = { title, author };
  res.status(201).json(newBook);
});
```

18. INTRODUCTION TO DATABASE APIS

Concept:

- **Database APIs** allow Node.js applications to interact with databases.
- Two common types of databases are **SQL databases** (e.g., MySQL, PostgreSQL) and **NoSQL databases** (e.g., MongoDB).
- **Database ORM (Object-Relational Mapping)** tools such as **Mongoose** (for MongoDB) and **Sequelize** (for SQL databases) are used to simplify database interaction.

Use Case:

- Storing user data (name, email, password) in a MongoDB database using Mongoose.

Interview Questions:

1. What is the difference between SQL and NoSQL databases?
2. How does ORM help in interacting with databases?

Coding Script:

```
// MongoDB with Mongoose example


const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/myapp', { useNewUrlParser: true });


const userSchema = new mongoose.Schema({

  name: String,

  email: String

});


const User = mongoose.model('User', userSchema);


User.create({ name: 'John Doe', email: 'john@example.com' })

  .then(user => console.log(user))

  .catch(err => console.log(err));
```

19. USING MONGODB WITH MONGOOSE

Concept:

- **Mongoose** is an ODM (Object Data Modeling) library that provides a straight-forward schema-based solution to model MongoDB data.
- It simplifies interaction with MongoDB by allowing developers to define object schemas and automatically handle the mapping between application objects and MongoDB documents.

Key Features:

1. **Schemas**: Define the structure of the data (fields and types).
2. **Models**: Create models that interact with collections in MongoDB.
3. **Validation**: Provide built-in validation to ensure that data is valid before it is saved to the database.

Use Case:

- Creating a blog system where blog posts are stored in a MongoDB collection.

Interview Questions:

1. What is Mongoose, and how does it interact with MongoDB?
2. What are the advantages of using Mongoose over the native MongoDB driver?

Coding Script:

```
// Defining a schema and model in Mongoose
const blogPostSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: { type: String, required: true },
  author: { type: String, required: true }
});

const BlogPost = mongoose.model('BlogPost', blogPostSchema);

// Creating a new blog post
const newPost = new BlogPost({
  title: 'My First Post',
  content: 'This is the content of my first blog post.',
  author: 'John Doe'
});

newPost.save()
  .then(post => console.log(post))
  .catch(err => console.log(err));
```

20. SQL DATABASES AND ORM (SEQUELIZE)

Concept:

- **SQL Databases** like MySQL and PostgreSQL use structured data and support SQL queries to interact with the data.
- **Sequelize** is an ORM (Object Relational Mapping) for Node.js that supports SQL databases and allows developers to interact with databases using JavaScript objects instead of raw SQL queries.

Key Features:

1. **Model Definition**: Define models using JavaScript that correspond to tables in the database.
2. **CRUD Operations**: Perform Create, Read, Update, Delete operations using model methods.
3. **Associations**: Define relationships between tables (e.g., One-to-Many, Many-to-Many).

Use Case:

- Managing an e-commerce application with a product database using Sequelize for ORM.

Interview Questions:

1. What is Sequelize, and how does it help in interacting with SQL databases?
2. How do you define associations between models in Sequelize?

Coding Script:

```
// Sequelize Model definition
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize('mysql://user:password@localhost:3306/mydb');

const User = sequelize.define('User', {
  name: {
    type: DataTypes.STRING,
    allowNull: false
  },
  email: {
    type: DataTypes.STRING,
    unique: true,
    allowNull: false
  }
});

// Syncing model with database
sequelize.sync()
  .then(() => console.log('User table created'))
  .catch(err => console.log(err));

// Creating a new user
```

```
User.create({ name: 'Jane Doe', email: 'jane@example.com' })
  .then(user => console.log(user))
  .catch(err => console.log(err));
```

21. DATABASE TRANSACTIONS

Concept:

- **Database transactions** allow multiple operations to be executed as a single unit, ensuring data integrity.
- If one operation fails, the entire transaction is rolled back, ensuring the database is not left in an inconsistent state.

Use Case:

- Ensuring that both the user registration and payment processes are completed successfully before committing the changes to the database.

Interview Questions:

1. What is a database transaction, and why is it important?
2. How do you handle transactions in Sequelize?

Coding Script:

```
// Sequelize transaction example
const sequelize = require('sequelize');

sequelize.transaction(async (t) => {
  try {
    const user = await User.create({ name: 'John', email: 'john@example.com' }, { transaction: t });
    await Payment.create({ amount: 100, userId: user.id }, { transaction: t });
    await t.commit();
  } catch (error) {
    await t.rollback();
    console.log('Transaction failed: ', error);
  }
});
```

22. RESTFUL API PRINCIPLES

Concept:

- RESTful APIs follow a stateless architecture, using HTTP methods to manipulate resources and ensure separation between client and server.
- REST principles ensure scalability and maintainability of APIs by adhering to a consistent and uniform design pattern.

Key Principles:

1. **Stateless**: Every request must contain all the information needed for the server to process it.
2. **Cacheable**: Responses must explicitly indicate whether they can be cached.
3. **Layered System**: APIs should be designed in layers to improve scalability and separation of concerns.

Use Case:

- Designing an API for an online marketplace to allow users to list products, place orders, and manage their profiles.

Interview Questions:

1. What is the key difference between REST and SOAP web services?
2. What is meant by statelessness in RESTful APIs?

Coding Script:

```
// Example of RESTful API for managing products

// GET request to retrieve products
app.get('/products', (req, res) => {
  const products = [
    { id: 1, name: 'Product 1', price: 100 },
    { id: 2, name: 'Product 2', price: 200 }
  ];
  res.json(products);
});

// POST request to add a new product
app.post('/products', (req, res) => {
 const { name, price } = req.body;
  const newProduct = { id: 3, name, price };
  res.status(201).json(newProduct);
});
```

23. AUTHENTICATION AND AUTHORIZATION (JWT, OAUTH)

Concept:

- **Authentication** ensures that the user is who they claim to be (e.g., via JWT or OAuth).
- **Authorization** determines what authenticated users are allowed to do.

Techniques:

1. **JWT (JSON Web Tokens)**: A popular method for handling authentication in stateless applications.
2. **OAuth**: A protocol that allows third-party services to exchange authentication tokens on behalf of a user.

Use Case:

- Using JWT to authenticate users and provide access to protected API routes.

Interview Questions:

1. What is the difference between authentication and authorization?
2. How does JWT authentication work in a Node.js application?

Coding Script:

```
// JWT Authentication example in Express.js

const jwt = require('jsonwebtoken');
const secretKey = 'your-secret-key';

// Middleware to check for JWT token
function authenticateJWT(req, res, next) {
  const token = req.header('Authorization');
  if (!token) {
    return res.status(403).send('Access Denied');
  }
  jwt.verify(token, secretKey, (err, user) => {
    if (err) {
      return res.status(403).send('Invalid Token');
    }
    req.user = user;
    next();
  });
}

// Login route
app.post('/login', (req, res) => {
  const username = req.body.username;
  const user = { name: username };
  const token = jwt.sign(user, secretKey);
  res.json({ token });
});

// Protected route
app.get('/profile', authenticateJWT, (req, res) => {
  res.send('Profile Info');
});
```