**Chapter 1: Introduction to Node.js**

Tasks:

1. Install Node.js and verify the installation.

2. Create a "Hello World" program using Node.js.

3. Explain the event-driven, non-blocking I/O model of Node.js.

4. Use the Node.js REPL (Read-Eval-Print Loop) to execute JavaScript code interactively.

5. Write a program that reads and displays the contents of a text file using the `fs` module.

6. Explore the Node.js package ecosystem (npm) and install a package.

7. Explain the difference between `require` and ES6 `import` for importing modules.

8. Use the `process` object to access command-line arguments.

9. Demonstrate the use of global objects like `\_\_dirname` and `\_\_filename`.

10. Create a simple HTTP server using the `http` module.

**Chapter 2: Asynchronous Programming and Callbacks**

Tasks:

1. Understand the concept of callbacks in Node.js.

Hint: Callbacks are functions passed as arguments to other functions to be executed later.

2. Convert a synchronous function into an asynchronous one using callbacks.

Hint: Focus on using the `setTimeout` function to simulate asynchronous behavior.

3. Solve the "Callback Hell" problem using named functions and modularization.

Hint: Promises or async/await will be covered in later chapters.

4. Read and display the contents of multiple files in a specific order using callbacks.

Hint: Use the `fs.readFile` function inside each callback.

5. Implement a basic error handling mechanism in callback-based code.

Hint: Consider using the error parameter convention in callback functions.

6. Create a program that simulates fetching data from an API using callbacks.

Hint: You can use libraries like `axios` or the built-in `http` module.

7. Explain the concept of the "Event Loop" in Node.js.

Hint: Research how the event loop processes the callback queue.

8. Build a simple CLI (Command Line Interface) app that takes user input using callbacks.

Hint: Use the `readline` module to gather user input.

9. Develop a program that reads a directory's contents and lists files/directories using callbacks.

Hint: Utilize the `fs.readdir` function.

10. Combine multiple asynchronous tasks using the `async` library.

Hint: Look into the `async.parallel` or `async.series` functions.

**Chapter 3: Promises and Async/Await**

Tasks:

1. Understand the drawbacks of using callbacks and the need for Promises.

Hint: Focus on handling callback hell and improving code readability.

2. Create a Promise-based version of a callback-based function.

Hint: Use the `util.promisify` method or manually create a Promise wrapper.

3. Chain multiple asynchronous operations using Promises.

Hint: Return Promises from within `.then()` handlers.

4. Handle errors in Promise chains using `.catch()`.

Hint: Simulate an error condition and demonstrate error propagation.

5. Rewrite callback-based code using async/await syntax.

Hint: Use the `util.promisify` method for functions that return callbacks.

6. Convert a Promise-based function into an async function and compare the syntax.

Hint: Show how async/await simplifies code readability.

7. Use `Promise.all()` to concurrently fetch data from multiple APIs.

Hint: Create an array of Promises and await their resolution using `.all()`.

8. Implement a basic rate-limiting mechanism using Promises and `setTimeout`.

Hint: Restrict the rate of API requests to stay within defined limits.

9. Build a file processing script that copies content from one file to another using async/await.

Hint: Use `fs.promises.readFile` and `fs.promises.writeFile`.

10. Create a program that recursively calculates the total size of directories using Promises.

Hint: Implement a recursive function that walks through directories.

**Chapter 4: Working with Modules and Dependency Management**

Tasks:

1. Understand the concept of modules in Node.js.

Hint: Explore the difference between built-in modules and user-defined modules.

2. Create a simple module with functions and variables and import it into another file.

Hint: Use the `module.exports` and `require` syntax.

3. Organize code into multiple modules and demonstrate hierarchical module structure.

Hint: Consider creating a directory structure for better organization.

4. Use the `npm` command to initialize a new Node.js project.

Hint: Run `npm init` and follow the prompts.

5. Install external packages using npm and save them as project dependencies.

Hint: Use the `npm install` command with appropriate flags.

6. Understand the difference between `dependencies` and `devDependencies`.

Hint: Dependencies are required for the application to run, while devDependencies are for development purposes.

7. Create a basic Express.js application and run a server.

Hint: Use the `express` package and its minimal setup.

8. Implement route handling in an Express.js application.

Hint: Define routes using HTTP methods like `get`, `post`, etc.

9. Use middleware in an Express.js application for tasks like logging and error handling.

Hint: Explore the concept of middleware functions in the Express.js documentation.

10. Build a RESTful API with Express.js that performs CRUD operations on a simple resource.

Hint: Define routes for creating, reading, updating, and deleting resource items.

**Chapter 5: Web Development with Express.js**

Tasks:

1. Create a static file server using Express.js to serve HTML, CSS, and client-side JavaScript files.

2. Implement middleware for handling form data and parsing JSON in incoming requests.

3. Build a basic authentication system using Express.js middleware.

4. Develop a session management system using libraries like `express-session`.

5. Implement a templating engine (e.g., EJS or Handlebars) to dynamically generate HTML.

6. Build a user registration and login system with password hashing and validation.

7. Create a RESTful API with authentication and authorization using Express.js.

8. Handle file uploads using libraries like `multer` in an Express.js application.

9. Implement error handling and create custom error middleware for better user experience.

10. Build a real-time chat application using WebSockets and the `socket.io` library.

**Chapter 6: Data Storage and Databases**

Tasks:

1. Connect to a MySQL database using the `mysql` package.

2. Perform basic CRUD operations (Create, Read, Update, Delete) on MySQL tables.

3. Use the `sequelize` library for ORM (Object-Relational Mapping) in Node.js with MySQL.

4. Connect to a MongoDB database using the `mongoose` library.

5. Define and perform CRUD operations on MongoDB collections using Mongoose.

6. Implement data validation and schema definition using Mongoose models.

7. Explore caching techniques using tools like `redis` for improving application performance.

8. Set up a basic caching mechanism for frequently accessed data using Redis.

9. Implement data pagination and sorting in database queries.

10. Build a RESTful API that interacts with a database for resource management.

**Chapter 7: Authentication and Authorization**

Tasks:

1. Understand the difference between authentication and authorization.

2. Implement local authentication using Passport.js and a username/password strategy.

3. Integrate third-party authentication (e.g., Google, Facebook) using Passport.js.

4. Implement token-based authentication using JWT (JSON Web Tokens).

5. Develop role-based authorization for controlling user access to specific routes.

6. Implement password reset functionality with email verification.

7. Secure sensitive information using environment variables and configuration.

8. Use middleware to restrict access to authenticated users and specific roles.

9. Implement OAuth 2.0 authentication for API access with Express.js.

10. Explore best practices for securing Express.js applications and preventing common security vulnerabilities.

**Chapter 8: Real-time Applications with Socket.IO**

Tasks:

1. Understand the concept of WebSockets and their role in real-time communication.

2. Set up a basic Socket.IO server and client for real-time messaging.

3. Implement chat functionality using Socket.IO for instant messaging.

4. Develop a real-time collaborative drawing application using Socket.IO.

5. Build a live notifications system using Socket.IO to alert users of updates.

6. Integrate Socket.IO with Express.js to create a real-time dashboard.

7. Implement rooms and namespaces to manage separate communication channels.

8. Secure Socket.IO connections and prevent unauthorized access.

9. Optimize Socket.IO for performance and handle a large number of connections.

10. Develop a real-time multiplayer game using Socket.IO.

**Chapter 9: Web APIs and RESTful Services**

Tasks:

1. Understand the principles of REST (Representational State Transfer).

2. Create a simple RESTful API using Express.js with different routes for CRUD operations.

3. Implement input validation and error handling in the API.

4. Use query parameters and route parameters to customize API responses.

5. Implement versioning in RESTful APIs using URL versioning or headers.

6. Explore HATEOAS (Hypermedia as the Engine of Application State) in RESTful APIs.

7. Implement rate limiting and throttling for API requests.

8. Use authentication and authorization mechanisms in RESTful APIs.

9. Implement filtering, sorting, and pagination for API responses.

10. Document your RESTful API using tools like Swagger or Postman.

**Chapter 10: Automated Testing and Debugging**

Tasks:

1. Understand the importance of automated testing in software development.

2. Write unit tests for individual functions using testing frameworks like `Mocha` and `Chai`.

3. Create test suites to cover various scenarios and edge cases.

4. Implement integration tests for routes and middleware in an Express.js application.

5. Use mocking and stubbing techniques to isolate dependencies in tests.

6. Explore continuous integration (CI) with tools like Travis CI or GitHub Actions.

7. Integrate code coverage tools to measure test coverage percentage.

8. Implement end-to-end (E2E) tests using tools like `Cypress` or `Puppeteer`.

9. Debug Node.js applications using built-in debugging tools and VS Code.

10. Use logging and error tracking tools (e.g., `winston`, `Sentry`) for production applications.

**Chapter 11: Performance Optimization**

Tasks:

1. Understand the importance of performance optimization in web applications.

2. Use profiling tools (e.g., `node --inspect`, `clinic`) to identify performance bottlenecks.

3. Optimize database queries using indexing, query optimization, and caching.

4. Implement browser caching and compression for static assets.

5. Minimize HTTP requests and optimize loading of assets.

6. Use a load testing tool (e.g., `Artillery`, `JMeter`) to simulate user traffic.

7. Optimize server-side rendering (SSR) for improved initial page load time.

8. Implement server clustering and load balancing for scalability.

9. Optimize memory usage and garbage collection in Node.js applications.

10. Implement performance monitoring to track application metrics and response times.

**Chapter 12: Serverless Computing with AWS Lambda**

Tasks:

1. Understand the concept of serverless computing and its benefits.

2. Set up an AWS account and create an IAM role for Lambda functions.

3. Develop a simple AWS Lambda function using Node.js.

4. Integrate Lambda functions with API Gateway to create serverless APIs.

5. Implement data storage using AWS services like DynamoDB or S3.

6. Trigger Lambda functions using events from services like S3 or SNS.

7. Secure Lambda functions and API Gateway using authentication and authorization.

8. Manage environment variables and configuration in serverless applications.

9. Use the AWS Serverless Application Model (SAM) for defining infrastructure as code.

10. Explore serverless best practices and considerations for production deployments.

**Chapter 13: GraphQL and Data Fetching**

Tasks:

1. Understand the differences between REST and GraphQL for data fetching.

2. Set up a basic GraphQL server using tools like `apollo-server` or `express-graphql`.

3. Define GraphQL schemas and types to represent your data.

4. Implement queries and mutations to fetch and modify data.

5. Use resolvers to connect GraphQL queries to actual data sources.

6. Explore pagination and filtering in GraphQL queries.

7. Implement authentication and authorization in GraphQL resolvers.

8. Integrate GraphQL with database systems like MongoDB or SQL databases.

9. Compare and contrast RESTful APIs and GraphQL in terms of data fetching.

10. Build a real-world application using GraphQL for efficient data retrieval.

**Chapter 14: Deployment and DevOps**

asks:

1. Understand the deployment process and its importance in application lifecycle.

2. Choose a hosting platform (e.g., Heroku, AWS, DigitalOcean) for your Node.js application.

3. Configure environment variables for different deployment environments.

4. Set up continuous integration and deployment (CI/CD) using tools like Jenkins or GitHub Actions.

5. Deploy a Node.js application to a cloud hosting platform (e.g., Heroku, AWS Elastic Beanstalk).

6. Implement zero-downtime deployment strategies for maintaining application availability.

7. Use containerization tools like Docker for packaging and deploying applications.

8. Set up monitoring and alerting using tools like New Relic or Prometheus.

9. Implement automatic scaling and load balancing for production applications.

10. Perform rollbacks and handle errors during deployments effectively.

**Chapter 15: Advanced Topics and Future Directions**

Tasks:

1. Explore advanced JavaScript concepts like closures, scopes, and the event loop.

2. Understand the role of microservices architecture in building scalable applications.

3. Implement server-side rendering (SSR) using frameworks like Next.js.

4. Explore WebAssembly and its potential applications in Node.js.

5. Implement machine learning and artificial intelligence in Node.js applications.

6. Build a GraphQL gateway for orchestrating multiple microservices.

7. Use Node.js for building real-time data streaming applications.

8. Explore the concept of serverless microservices and event-driven architectures.

9. Implement internationalization (i18n) and localization in Node.js applications.

10. Investigate emerging trends and future directions of Node.js and web development.