Zensar Technologies

# NodeJS (Express + MongoDB) Assignments

## Module Patterns in Node.js

### 1. Basic Module Export and Import

**Assignment:** Create a basic Node.js module that exports a function and a constant value. Then, import and use this module in another file.

**Tasks:**

1. **Create a module file** (mathUtils.js) that exports a function add which takes two numbers and returns their sum, and a constant PI with the value 3.14.
2. **Create another file** (app.js) that imports the add function and PI constant from mathUtils.js and demonstrates their usage.

### 2. Module Pattern with Singleton

**Assignment:** Implement a singleton pattern using Node.js modules.

**Tasks:**

1. **Create a module file** (config.js) that exports a singleton object with properties host, port, and database. Ensure that multiple imports of this module return the same instance.
2. **Create a file** (app.js) that imports the config module multiple times and logs the instance to show that it is the same across imports.

### 3. Module Pattern with Revealing Module Pattern

**Assignment:** Implement the Revealing Module Pattern in Node.js.

**Tasks:**

1. **Create a module file** (userService.js) that uses the Revealing Module Pattern to expose methods getUser and createUser. getUser should return a fixed user object, and createUser should log a message with the user data.
2. **Create a file** (app.js) that imports the userService module and demonstrates using getUser and createUser.

### 4. Factory Module Pattern

**Assignment:** Implement a factory module pattern to create different types of users.

**Tasks:**

1. **Create a module file** (userFactory.js) that exports a function createUser which takes a type parameter (admin or guest) and returns different user objects based on the type.
2. **Create a file** (app.js) that imports the userFactory module and creates both admin and guest users, logging their details to the console.

### 5. Module Pattern with Dependency Injection

**Assignment:** Implement a module pattern with dependency injection.

**Tasks:**

1. **Create a module file** (logger.js) that exports a simple logger with a log method.
2. **Create a module file** (service.js) that accepts a logger as a dependency and uses it to log messages.
3. **Create a file** (app.js) that imports both logger and service, and demonstrates dependency injection by passing the logger instance to the service.

## Publishing and Consuming Modules & Scoped and Un-Scoped Modules

**1. Publishing and Consuming Modules**

**a. Publishing a Module**

**Assignment:**
Create a Node.js module and publish it to npm (you can use a private npm registry or test npm registry for this purpose).

**Tasks:**

1. Create a module file (`mathLib.js`) that exports a function `multiply` which takes two numbers and returns their product.
2. Create a `package.json` file for your module with appropriate metadata. Ensure it includes the module name, version, and entry point.
3. Publish the module to npm or a private registry.

Hint: You might want to use the `npm init` command to generate `package.json`, and `npm publish` to publish your module.

**b. Consuming a Published Module**

**Assignment:**
Create a Node.js application that consumes the published module.

**Tasks:**

1. Create a new directory for your Node.js application.
2. Create a `package.json` file for the application using `npm init`.
3. Install the published module (you can use `npm install <module-name>` if published to the public npm registry, or provide a path to the module if locally published).
4. Create a file (`app.js`) that imports and uses the `multiply` function from the published module.

Hint: Use `require('<module-name>')` to import the module.

**2. Scoped and Unscoped Modules**

**a. Unscoped Module**

**Assignment:**
Create an unscoped Node.js module and use it in another application.

**Tasks:**

1. Create a module file (`unscopedUtils.js`) that exports a function `subtract` which takes two numbers and returns their difference.
2. Create a `package.json` file for this module.
3. Create another directory for a new Node.js application.

4. Install the unscoped module locally by referencing the path to the module.
5. Create a file (`app.js`) that imports and uses the `subtract` function from the unscoped module.

Hint: Use `npm install <path-to-module>` to install the unscoped module locally.

**b. Scoped Module**

**Assignment:**
Create a scoped Node.js module and use it in another application.

**Tasks:**

1. Create a module file (`@my-scope/scopedUtils.js`) that exports a function `divide` which takes two numbers and returns their quotient.
2. Create a `package.json` file for this module with a scoped name (e.g., `@my-scope/scopedUtils`).
3. Publish the scoped module to npm or a private registry (ensure you use the appropriate scope when publishing).
4. Create another directory for a new Node.js application.
5. Install the scoped module using npm.
6. Create a file (`app.js`) that imports and uses the `divide` function from the scoped module.

Hint: Use `npm install @my-scope/scopedUtils` to install the scoped module.

## Core Modules
## Events – EventEmitter

**1. Basic Event Emitter Usage**
**Assignment:** Create a simple Node.js application that uses the EventEmitter class from the events module to emit and listen for custom events.
**Tasks:**

1. **Create a file** (eventEmitterDemo.js) that:
   - Imports the EventEmitter class from the events module.
   - Creates an instance of EventEmitter.
   - Defines a custom event (message) and attaches a listener that logs a message to the console.
   - Emits the custom event with a message.
2. **Run the eventEmitterDemo.js file** to see the event handling in action.

**2. Multiple Event Listeners**
**Assignment:** Extend the previous example to handle multiple listeners for the same event and ensure that all listeners are invoked when the event is emitted.
**Tasks:**

1. **Update the file** (eventEmitterDemo.js) to:
   - Add multiple listeners for the custom event (message), each logging a different message to the console.
   - Ensure that all listeners are executed when the event is emitted.
2. **Run the updated eventEmitterDemo.js file** to verify that all listeners are called.

**3. Event Listener Removal**
**Assignment:** Demonstrate how to remove event listeners using the removeListener method from the EventEmitter class.
**Tasks:**

1. **Create a file** (eventEmitterRemoveListener.js) that:
   - Imports the EventEmitter class from the events module.
   - Creates an instance of EventEmitter.
   - Defines and attaches multiple listeners for a custom event (notify).
   - Emits the event and logs a message indicating which listeners were invoked.
   - Removes one of the listeners.
   - Emits the event again and logs a message to show which listeners are still active.
2. **Run the eventEmitterRemoveListener.js file** to observe the effect of removing an event listener.

**4. EventEmitter and Asynchronous Operations**
**Assignment:** Use EventEmitter to handle asynchronous operations, such as reading a file and emitting an event when the operation completes.
**Tasks:**

1. **Create a file** (fileReadEmitter.js) that:
   - Imports the EventEmitter class and the fs module from Node.js.
   - Creates an instance of EventEmitter.
   - Sets up an event listener for a custom event (fileReadComplete).
   - Reads a file asynchronously using fs.readFile.
   - Emits the fileReadComplete event with the file contents when the file read operation completes.
2. **Run the fileReadEmitter.js file** to see the asynchronous event handling in action.

**5. Custom Event Classes**
**Assignment:** Define a custom class that extends the EventEmitter class and includes methods for emitting and handling custom events.
**Tasks:**

1. **Create a file** (customEventEmitter.js) that:
   - Defines a custom class CustomEmitter extending EventEmitter.
   - Adds a method triggerEvent that emits a custom event (customEvent).
   - Adds a method onCustomEvent that attaches a listener to handle customEvent.
   - Creates an instance of CustomEmitter, attaches the listener, and triggers the event.
2. **Run the customEventEmitter.js file** to verify that the custom event is properly handled.

## Core Modules
## Streams

**1. Reading from a Stream**
**Assignment:** Create a simple Node.js application that reads data from a readable stream and logs it to the console.
**Tasks:**

1. **Create a file** (readStreamDemo.js) that:
   - Imports the fs module from Node.js.
   - Creates a readable stream from a file (e.g., input.txt).

- Uses the data event to log chunks of data as they are read from the file.
- Handles the end event to log a message when the stream ends.

2. **Run the readStreamDemo.js file** to observe how data is read and logged from the file.

**2. Writing to a Stream**

**Assignment:** Create a Node.js application that writes data to a writable stream.

**Tasks:**

1. **Create a file** (writeStreamDemo.js) that:
   - Imports the fs module from Node.js.
   - Creates a writable stream to a file (e.g., output.txt).
   - Writes some data to the file using the write method.
   - Handles the finish event to log a message when writing is complete.
2. **Run the writeStreamDemo.js file** to verify that data is written to the file and the finish message is logged.

**3. Piping Data Between Streams**

**Assignment:** Demonstrate how to pipe data from a readable stream to a writable stream.

**Tasks:**

1. **Create a file** (pipeStreams.js) that:
   - Imports the fs module from Node.js.
   - Creates a readable stream from an input file (e.g., input.txt).
   - Creates a writable stream to an output file (e.g., output.txt).
   - Pipes the readable stream to the writable stream.
2. **Run the pipeStreams.js file** to observe how data from the input file is transferred to the output file using piping.

**4. Transform Streams**

**Assignment:** Create a transform stream that processes data as it is read from a readable stream and then writes it to a writable stream.

**Tasks:**

1. **Create a file** (transformStreamDemo.js) that:
   - Imports the stream and fs modules from Node.js.
   - Creates a custom transform stream class that converts data to uppercase.
   - Creates a readable stream from an input file (e.g., input.txt).
   - Creates a writable stream to an output file (e.g., output.txt).
   - Pipes the readable stream through the transform stream to the writable stream.
2. **Run the transformStreamDemo.js file** to see the transformation of data as it is processed and written to the output file.

**5. Implementing a Custom Readable Stream**

**Assignment:** Implement a custom readable stream that generates a sequence of numbers.

**Tasks:**

1. **Create a file** (customReadableStream.js) that:
   - Imports the stream module from Node.js.
   - Creates a custom readable stream class that generates numbers from 1 to 10.
   - Implements the _read method to push numbers to the stream.
   - Creates an instance of the custom readable stream and logs each number as it is read.
2. **Run the customReadableStream.js file** to observe the custom stream generating and logging the sequence of numbers.

**6. Handling Stream Errors**
**Assignment:** Create a Node.js application that demonstrates handling errors in streams.
**Tasks:**

1. **Create a file** (errorHandlingStream.js) that:
    - Imports the fs module from Node.js.
    - Creates a readable stream from a non-existent file to trigger an error.
    - Creates a writable stream to a file (e.g., output.txt).
    - Pipes the readable stream to the writable stream and handles errors using the error event.
2. **Run the errorHandlingStream.js file** to observe how errors are handled and logged.

**7. Combining Multiple Streams**
**Assignment:** Combine multiple streams into a single readable stream.
**Tasks:**

1. **Create a file** (combineStreams.js) that:
    - Imports the stream module from Node.js.
    - Creates multiple readable streams from different sources (e.g., stream1.txt, stream2.txt).
    - Uses the stream.Readable class to create a combined readable stream.
    - Pipes the combined readable stream to a writable stream (e.g., combinedOutput.txt).
2. **Run the combineStreams.js file** to verify that data from multiple streams is combined and written to the output file.

## Core Modules
## File System, File System Watcher and Chokidar

**1. File System Operations**
**1.1 Reading and Writing Files**
**Assignment:** Create a Node.js application that performs basic file system operations, including reading from and writing to files.
**Tasks:**

1. **Create a file** (fileSystemDemo.js) that:
    - Imports the fs module from Node.js.
    - Reads the contents of an existing file (e.g., input.txt) asynchronously.
    - Logs the file contents to the console.
    - Writes a new line of text to a new file (e.g., output.txt).
2. **Run the fileSystemDemo.js file** to verify that the file contents are read and logged, and the new text is written to the output file.

**1.2 Appending to a File**
**Assignment:** Create a Node.js application that appends data to an existing file.
**Tasks:**

1. **Create a file** (appendToFile.js) that:
    - Imports the fs module from Node.js.
    - Appends a new line of text to an existing file (e.g., log.txt).
    - Logs a confirmation message to the console once the data is appended.

2. **Run the appendToFile.js file** to ensure the new line is appended to the file and the confirmation message is logged.

**1.3 Deleting and Renaming Files**
**Assignment:** Create a Node.js application that deletes and renames files.
**Tasks:**

1. **Create a file** (fileManagement.js) that:
   - Imports the fs module from Node.js.
   - Deletes an existing file (e.g., oldFile.txt).
   - Renames a file (e.g., temp.txt to newFile.txt).
   - Logs confirmation messages for the delete and rename operations.
2. **Run the fileManagement.js file** to verify that the file is deleted and renamed successfully, and the confirmation messages are logged.

**2. File Watcher and Chokidar**
**2.1 Basic File Watching with Chokidar**
**Assignment:** Use Chokidar to watch a directory for changes and log those changes to the console.
**Tasks:**

1. **Create a file** (fileWatcher.js) that:
   - Installs Chokidar (npm install chokidar).
   - Imports the chokidar module.
   - Sets up a watcher to monitor a directory (e.g., ./watchedDir).
   - Listens for file changes (additions, deletions, and modifications) and logs the events to the console.
2. **Create and modify files in the watched directory** and run the fileWatcher.js file to observe the logged events.

**2.2 Handling Specific Events with Chokidar**
**Assignment:** Create a Node.js application that handles specific events with Chokidar, such as file additions and modifications.
**Tasks:**

1. **Create a file** (eventWatcher.js) that:
   - Installs Chokidar (npm install chokidar).
   - Imports the chokidar module.
   - Sets up a watcher to monitor a directory (e.g., ./eventsDir).
   - Listens for specific events like add and change.
   - Logs messages indicating which files were added or changed.
2. **Create and modify files in the watched directory** and run the eventWatcher.js file to verify that the specific events are handled and logged.

**2.3 Debouncing File Changes**
**Assignment:** Use Chokidar to debounce file change events to avoid handling multiple events for a single change.
**Tasks:**

1. **Create a file** (debouncedWatcher.js) that:
   - Installs Chokidar (npm install chokidar).
   - Imports the chokidar module.
   - Sets up a watcher to monitor a directory (e.g., ./debounceDir).
   - Implements debouncing for file change events using a delay (e.g., 500ms) to handle only the final event after a series of changes.
   - Logs the debounced change events to the console.

2. **Create and modify files in the watched directory** and run the debouncedWatcher.js file to verify that changes are debounced and logged correctly.

**2.4 Monitoring Multiple Directories with Chokidar**
**Assignment:** Create a Node.js application that monitors multiple directories with Chokidar and logs changes for each directory.
**Tasks:**

1. **Create a file** (multiDirWatcher.js) that:
    - Installs Chokidar (npm install chokidar).
    - Imports the chokidar module.
    - Sets up watchers to monitor multiple directories (e.g., ./dir1 and ./dir2).
    - Listens for changes in each directory and logs the events with the directory name.
2. **Create and modify files in the watched directories** and run the multiDirWatcher.js file to verify that changes in all directories are logged correctly.

## Core Modules
## HTTP Module

**1. Basic HTTP Server**
**Assignment:** Create a simple HTTP server that responds with "Hello, Node.js!" to all incoming requests.
**Tasks:**

1. **Create a file** named basicHttpServer.js that:
    - Imports the http module.
    - Creates an HTTP server that listens on port 4000.
    - Responds with "Hello, Node.js!" to any request.
2. **Run the basicHttpServer.js file** and test it using a web browser or curl to ensure the response is as expected.

**2. HTTP Server with Static Files**
**Assignment:** Create an HTTP server that serves static HTML files from a directory.
**Tasks:**

1. **Create a file** named staticFileServer.js that:
    - Imports the http, fs, and path modules.
    - Creates an HTTP server that listens on port 5000.
    - Serves static HTML files from a directory called public:
        - If a request is made to /, serve index.html from the public directory.
        - Serve any other file requested from the public directory.
2. **Create an HTML file** named index.html inside a directory called public.
3. **Run the staticFileServer.js file** and test serving the HTML file using a web browser.

**3. HTTP Server with JSON Response**
**Assignment:** Create an HTTP server that responds with a JSON object based on the request path.
**Tasks:**

1. **Create a file** named jsonResponseServer.js that:
    - Imports the http module.
    - Creates an HTTP server that listens on port 6000.
    - Responds with JSON objects for specific paths:

- /api/user should respond with {"name": "Sachin Tendulkar", "age": 51}.
- /api/product should respond with {"id": 1, "name": "Widget", "price": 19.99}.

  - Respond with a 404-status code for any other paths.
2. **Run the jsonResponseServer.js file** and test the JSON responses using a web browser or curl.

**4. Handling Query Parameters**

**Assignment:** Create an HTTP server that responds based on query parameters.

**Tasks:**

1. **Create a file** named queryParamsServer.js that:
   - Imports the http and url modules.
   - Creates an HTTP server that listens on port 7000.
   - Handles GET requests to /greet and responds with "Hello, [name]!" where [name] is a query parameter. If no name is provided, respond with "Hello, Guest!".
2. **Run the queryParamsServer.js file** and test the server using URLs like http://localhost:7000/greet?name=Alice.

**5. HTTP Server with POST Data Handling**

**Assignment:** Create an HTTP server that handles POST requests and echoes back the received data.

**Tasks:**

1. **Create a file** named postDataServer.js that:
   - Imports the http module.
   - Creates an HTTP server that listens on port 8000.
   - Handles POST requests to /submit by collecting the data and responding with the received data.
2. **Run the postDataServer.js file** and test sending POST data using curl or Postman.

**6. HTTP Server with Multiple Routes**

**Assignment:** Create an HTTP server that handles multiple routes and responds with different messages based on the path.

**Tasks:**

1. **Create a file** named multiRouteServer.js that:
   - Imports the http module.
   - Creates an HTTP server that listens on port 9000.
   - Responds with specific messages based on the request path:
     - /home should respond with "Home Page".
     - /contact should respond with "Contact Us".
     - /about should respond with "About Us".
   - Respond with "404 Not Found" for any other paths.
2. **Run the multiRouteServer.js file** and test the different routes using a web browser or curl.

**7. HTTP Server with Custom Headers**

**Assignment:** Create an HTTP server that includes custom headers in its responses.

**Tasks:**

1. **Create a file** named headerServer.js that:
   - Imports the http module.
   - Creates an HTTP server that listens on port 10000.
   - Responds to all requests with the following custom headers:
     - X-Custom-Header: MyCustomHeader
     - X-Powered-By: Node.js
   - Responds with "Custom Headers Applied" in the response body.

2. **Run the headerServer.js file** and check the headers using browser developer tools or curl.

**8. HTTP Server with Error Handling**

**Assignment:** Create an HTTP server that handles errors and responds with a 500 status code for internal server errors.

**Tasks:**

1. **Create a file** named errorHandlingServer.js that:
   - Imports the http module.
   - Creates an HTTP server that listens on port 11000.
   - Handles requests and simulates an error by throwing an exception.
   - Responds with a 500-status code and "Internal Server Error" message.
2. **Run the errorHandlingServer.js file** and test error handling by sending requests to the server.

## Express

**1. Creating Server and Client using Express**

**Assignment:**

**A. Create an Express Server**

**Tasks:**

1. **Create a file** named server.js that:
   - Imports the express module.
   - Initializes an Express application.
   - Sets up routes to handle the following requests:
     - GET / should respond with "Welcome to the Express Server!".
     - GET /api/data should respond with a JSON object { "message": "Here is your data" }.
   - Listens on port 3000 and logs a message when the server is running.
2. **Run the server.js file** and ensure the server is listening on port 3000.

**B. Create a Simple Client to Communicate with the Server**

**Tasks:**

1. **Create a file** named client.js that:
   - Imports the axios module (you may need to install it using npm install axios).
   - Sends GET requests to the following endpoints:
     - http://localhost:3000/ and logs the response.
     - http://localhost:3000/api/data and logs the response.
2. **Run the client.js file** and ensure that it correctly logs responses from the server.

**2. Create Express App with Different View Engines**

**Assignment:**

**A. Create an Express App Using EJS as a View Engine**

**Tasks:**

1. **Create a file** named appEJS.js that:
   - Imports the express module and the ejs view engine.
   - Initializes an Express application.
   - Sets ejs as the view engine.

   - Creates a route GET / that renders a view named index.ejs with a title variable.
   - Creates a directory named views and within it, create an index.ejs file that displays the title variable.
2. **Run the appEJS.js file** and ensure that the route renders the index.ejs view with the provided title.

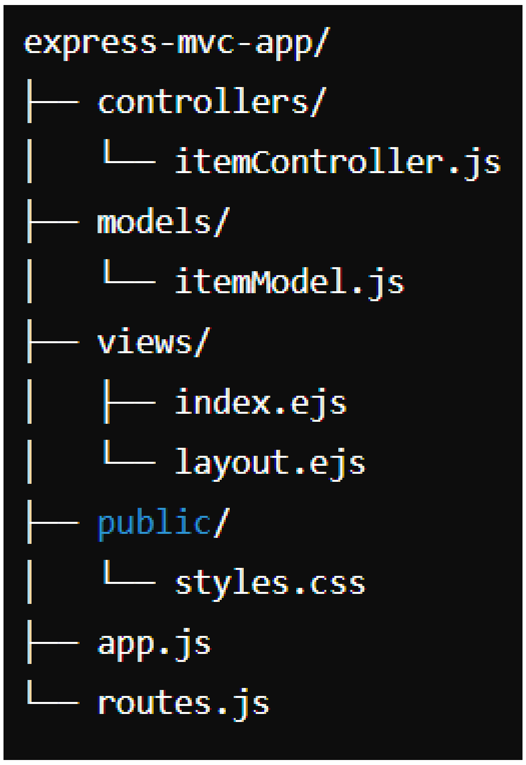
**B. Create an Express App Using Pug as a View Engine**
**Tasks:**
1. **Create a file** named appPug.js that:
   - Imports the express module and the pug view engine.
   - Initializes an Express application.
   - Sets pug as the view engine.
   - Creates a route GET / that renders a view named index.pug with a title variable.
   - Creates a directory named views and within it, create an index.pug file that displays the title variable.
2. **Run the appPug.js file** and ensure that the route renders the index.pug view with the provided title.

**C. Create an Express App Using Handlebars as a View Engine**
**Tasks:**
1. **Create a file** named appHandlebars.js that:
   - Imports the express module and the express-handlebars view engine.
   - Initializes an Express application.
   - Sets handlebars as the view engine.
   - Creates a route GET / that renders a view named index.handlebars with a title variable.
   - Creates a directory named views and within it, create an index.handlebars file that displays the title variable.
2. **Run the appHandlebars.js file** and ensure that the route renders the index.handlebars view with the provided title.

## Express MVC Application

**Assignment: Building an Express MVC Application**
**Objective:** Create a basic Express application that follows the MVC pattern. The application will include functionality to manage a list of items.

**Tasks:**
1. **Set Up the Project:**
   - **Create a new directory** for your project, e.g., express-mvc-app.
   - **Initialize the project** with npm:
   - **Install the required packages**:
2. **Create the Project Structure:**
   - **Create the following directory structure:**



```
express-mvc-app/
├── controllers/
│   └── itemController.js
├── models/
│   └── itemModel.js
├── views/
│   ├── index.ejs
│   └── layout.ejs
├── public/
│   └── styles.css
├── app.js
└── routes.js
```

3. **Implement the Model:**
   - **Create a file** named models/itemModel.js that:
     - Defines a basic in-memory array to store items. (array of item name)
     - Implements functions to get all items and add a new item.

4. **Implement the Controller:**
   - **Create a file** named controllers/itemController.js that:
     - Imports the model functions.
     - Defines functions to handle HTTP requests:
       - getItems: Retrieves and renders the list of items.
       - addItem: Handles adding a new item.

5. **Implement the Routes:**
   - **Create a file** named routes.js that:
     - Defines routes for the application:
       - GET /: Display the list of items.
       - POST /add: Add a new item.

6. **Create the Views:**
   - **Create a file** named views/index.ejs that:
     - Displays the list of items in an HTML table.
     - Includes a form to add a new item.
   - **Create a file** named views/layout.ejs that:
     - Defines a basic HTML layout including a header, body, and footer.
     - Includes a section for content injection.

7. **Set Up the Express Application:**
   - **Create a file** named app.js that:
     - Sets up the Express application.

- Configures middleware for serving static files, parsing request bodies, and setting up EJS as the view engine.
- Imports and uses the routes defined in routes.js.

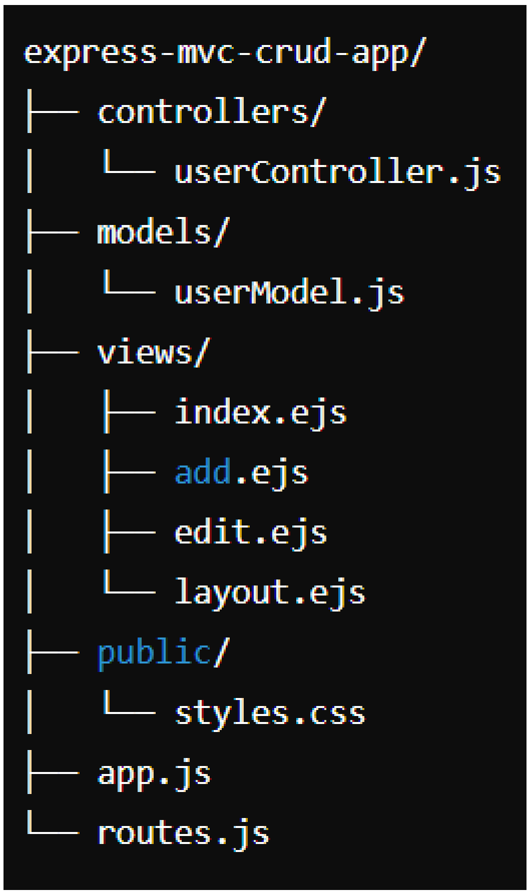
## Express MVC CRUD Application with in-memory Database

**Assignment: Building an Express MVC CRUD Application**
**Objective:** Create a basic Express application that implements CRUD operations using an in-memory database. The application will manage a list of users.

**Tasks:**

1. **Set Up the Project:**
   - **Create a new directory** for your project, e.g., express-mvc-crud-app.
   - **Initialize the project** with npm:
   - **Install the required packages**:

2. **Create the Project Structure:**
   - **Create the following directory structure:**



```
express-mvc-crud-app/
├── controllers/
│   └── userController.js
├── models/
│   └── userModel.js
├── views/
│   ├── index.ejs
│   ├── add.ejs
│   ├── edit.ejs
│   └── layout.ejs
├── public/
│   └── styles.css
├── app.js
└── routes.js
```

3. **Implement the Model:**
   - **Create a file** named models/userModel.js that:
     - Defines an in-memory array to store user objects.
     - Implements functions to get all users, get a user by ID, add a new user, update a user, and delete a user.

4. **Implement the Controller:**
   - **Create a file** named controllers/userController.js that:
     - Imports the model functions.
     - Defines functions to handle HTTP requests for CRUD operations:
       - getAllUsers: Retrieves and renders the list of users.
       - getUserById: Retrieves and renders a single user.
       - createUser: Handles adding a new user.
       - updateUser: Handles updating an existing user.
       - deleteUser: Handles deleting a user.

5. **Implement the Routes:**
   - **Create a file** named routes.js that:
     - Defines routes for CRUD operations:
       - GET /: Display the list of users.
       - GET /user/add: Show the form to add a new user.
       - POST /user/add: Add a new user.
       - GET /user/edit/:id: Show the form to edit a user.
       - POST /user/edit/:id: Update a user.
       - POST /user/delete/:id: Delete a user.

6. **Create the Views:**
   - **Create a file** named views/index.ejs that:
     - Displays the list of users in an HTML table.
     - Includes links to edit and delete each user.
   - **Create a file** named views/add.ejs that:
     - Provides a form to add a new user.
   - **Create a file** named views/edit.ejs that:
     - Provides a form to edit an existing user.
   - **Create a file** named views/layout.ejs that:
     - Defines a basic HTML layout including a header, body, and footer.
     - Includes a section for content injection.

7. **Set Up the Express Application:**
   - **Create a file** named app.js that:
     - Sets up the Express application.
     - Configures middleware for serving static files, parsing request bodies, and setting up EJS as the view engine.
     - Imports and uses the routes defined in routes.js.

## MongoDB Basics

**(Note: - You can try following assignments on Mongo Shell and then with MongoDB or mongoose code)**

**1. Data Modeling**

- **Assignment**: Design a data model for a blog application where users can have multiple posts, and each post can have multiple comments. Consider relationships between users, posts, and comments. Create the necessary schema for the model using Mongoose in Node.js.

**2. Create Database**

- **Assignment**: Write a Node.js script using Mongoose to create a new MongoDB database named myblog. Ensure the database connection is properly handled.

**3. Drop Database**

- **Assignment**: Write a Node.js script to drop the myblog database created in the previous assignment. Ensure that the operation is confirmed before proceeding with the drop.

**4. Create Collection**

- **Assignment**: Write a Node.js script to create a collection named users in the myblog database. Define a schema for the collection that includes fields for username, email, password, and created_at.

**5. Drop Collection**

- **Assignment**: Write a Node.js script to drop the users collection from the myblog database. Ensure the operation is confirmed before proceeding with the drop.

**6. Data Types**

- **Assignment**: Create a products collection in MongoDB with fields that demonstrate various data types, including string, number, boolean, array, object, and date. Write a Node.js script to insert documents that include examples of these data types.

**7. Insert Document**

- **Assignment**: Write a Node.js script to insert multiple documents into the users collection. Each document should represent a user with fields for username, email, password, and created_at.

**8. Query Document**

- **Assignment**: Write a Node.js script to query the users collection and retrieve:
  - All users.
  - A user by their username.
  - Users who registered after a specific date.
  - Users with an email domain of "example.com".

**9. Update Document**

- **Assignment**: Write a Node.js script to update documents in the users collection:
  - Update the email field for a specific user identified by username.
  - Set a verified field to true for all users who registered before a specific date.

**10. Delete Document**

- **Assignment**: Write a Node.js script to delete documents from the users collection:
  - Delete a specific user by username.
  - Delete all users who haven't verified their email (consider the verified field from the previous assignment).

## Create express app with different view engines using express-generator (pug, ejs, hbs, hjs, twig, vash etc)

https://expressjs.com/en/starter/generator.html

## Express MVC CRUD Application with MongoDB Database

**Assignment Overview:**
Build a simple Express MVC (Model-View-Controller) CRUD application that interacts with a MongoDB database. Application should allow users to create, read, update, and delete records.

**Assignment Requirements:**

1. **Project Setup:**
   - Set up an Express project.
   - Install necessary dependencies (express, mongoose, body-parser, dotenv, express-handlebars or any templating engine of your choice).
   - Create a .env file to store environment variables such as the MongoDB connection string.

2. **Models:**
   - Create a user model using Mongoose.
   - The **User** model should have the following fields:
     - **name: String, required.**
     - **email: String, required, unique.**
     - **age: Number, optional.**

3. **Controllers:**
   - Implement controller methods for:
     - Listing all users.
     - Displaying a form for adding a new user.
     - Creating a new user.
     - Displaying a form for editing a user.
     - Updating a user.
     - Deleting a user.

4. **Views:**
   - Implement views using handlebars (or any templating engine).
   - Create views for:
     - Listing users (index.handlebars).
     - Adding a new user (add.handlebars).
     - Editing a user (edit.handlebars).
   - Layout should include a navigation menu to access different parts of the application.

5. **Routes:**
   - Set up routes in userRoutes.js to handle:
     - Displaying all users.
     - Displaying the form to add a user.

- Submitting the form to add a user.
- Displaying the form to edit a user.
- Submitting the form to update a user.
- Deleting a user.

6. **MongoDB Connection:**
   - Connect to a MongoDB database using Mongoose.
   - Use environment variables to configure the connection string.

7. **Error Handling:**
   - Implement basic error handling for database operations (e.g., display an error message if a user cannot be found).

8. **CRUD Operations:**
   - Implement the following CRUD operations:
     - **Create:** Add a new user to the database.
     - **Read:** List all users from the database.
     - **Update:** Edit and update an existing user.
     - **Delete:** Remove a user from the database.

## RESTful Service using Express

**Assignment: Creating a RESTful Service Using Express**
**Objective:** Build a RESTful API for managing a collection of books. Implement CRUD (Create, Read, Update, Delete) operations.

**Instructions:**

1. **Setup Project:**
   - Initialize a new Node.js project.
   - Install Express and other necessary packages.
   - Create the initial project structure.

2. **Define Book Model:**
   - Use an in-memory data structure (like an array) to store book data. Each book should have id, title, author, and publishedDate.

3. **Create RESTful Endpoints:**
   - **GET /books** - Retrieve a list of all books.
   - **GET /books/**

- Retrieve a specific book by id.
   - **POST /books** - Create a new book.
   - **PUT /books/**

- Update a specific book by id.
   - **DELETE /books/**

- Delete a specific book by id.

4. **Handle Errors:**
   - Ensure proper error handling for scenarios such as invalid id, missing fields, etc.

5. **Test API:**
   - Use Thunder Client (VS Code Extension) or a similar tool to test the API endpoints.

**Project Structure:**
- app.js - Main application file
- routes/ - Directory for route handlers
  - books.js - Routes for book-related operations

**Create React app using Vite or Angular Standalone App as an Client app to consume Books RESTful Service**

## WebSocket

**1. WebSocket Basics in Node.js**

**Assignment 1: Create a Basic WebSocket Server**
**- Objective:** Set up a basic WebSocket server that listens for client connections and logs incoming messages to the console.

**Instructions:**
1. Create a Node.js project and install the `ws` WebSocket library.
2. Implement a WebSocket server that listens on port `8080`.
3. Log messages received from clients to the console.
4. Send a welcome message to clients when they connect.

**Assignment 2: Create a WebSocket Client**
**- Objective:** Implement a simple WebSocket client that connects to the server and sends a message.

**Instructions:**
1. Create an HTML file with a script to connect to the WebSocket server.
2. Send a message to the server upon connecting.
3. Log incoming messages from the server to the console.

**2. Basic Chat Application**

**Assignment 3: Build a Simple Chat Server**
**- Objective:** Create a WebSocket server that allows multiple clients to communicate with each other.

**Instructions:**
1. Extend the WebSocket server to broadcast messages to all connected clients.
2. Store client connections in an array and iterate over them to send messages.

**Assignment 4: Build a Chat Client**
- **Objective:** Create a client that allows users to send and receive messages in a chat application.

**Instructions:**
1. Extend the HTML client to include an input field and a button for sending messages.
2. Display received messages in a chat window.

**3. Advanced Chat Application Features**

**Assignment 5:** Add User Identification
- **Objective:** Enhance the chat application by adding user identification and displaying the user name with each message.

**Instructions:**
1. Modify the server to handle user names.
2. Modify the client to allow users to enter their names before sending messages.

## hapi

**1. Setting Up the Hapi Server**

**Assignment 1: Create a Basic Hapi Server**
- **Objective:** Set up a basic Hapi server that serves a static HTML file.

**Instructions:**
1. Initialize a new Node.js project and install Hapi.js.
2. Implement a basic Hapi server that serves an HTML page.

**2. Serving Dynamic Content**

**Assignment 2: Serve Dynamic Content Using Hapi.js**
- **Objective:** Create a route that dynamically serves content based on request parameters.

**Instructions:**
1. Add a route to the Hapi server that renders dynamic content.

**3. Template Rendering**

**Assignment 3: Use Hapi.js with a Templating Engine**
- **Objective:** Set up a Hapi server to render HTML templates using a templating engine like `@hapi/vision` with `handlebars` or `pug`.

**Instructions:**
1. Install `@hapi/vision` and a templating engine (`handlebars` or `pug`).
2. Configure Hapi to use the templating engine for rendering views.

**4. Adding Form Handling**

**Assignment 4: Handle Form Submissions**
**- Objective:** Create a form and handle form submissions in Hapi.js.

**Instructions:**
1. Create a form in an HTML file.
2. Add a route to handle form submissions.

## RESTful Service using hapi

**Creating a RESTful Service Using Hapi.js**
**Objective**: Implement a simple RESTful service using Hapi.js. This service will handle CRUD operations for a resource (e.g., books), which will be stored in an in-memory database.

**1. Setup Your Project**
**Assignment 1: Initialize Project and Install Dependencies**
- **Instructions**:
  1. Initialize a new Node.js project.
  2. Install @hapi/hapi.

**2. Implement Basic Hapi Server**
**Assignment 2: Create a Basic Hapi Server**
- **Instructions**:
  1. Create a basic Hapi server setup.
  2. Ensure the server is running and responds with a simple message.

**3. Implement CRUD Operations**
**Assignment 3: Create CRUD Routes**
- **Instructions**:
  1. Implement routes to handle CRUD operations for a books resource.
  2. Use an in-memory array to store the books.

**4. Test the RESTful Service**
**Assignment 4: Test CRUD Operations**
- **Instructions**:
  1. Use tools like curl, Postman, or httpie to test the CRUD operations.
  2. Verify that you can create, read, update, and delete books using the API endpoints

## Micro services

**Objective:** Implement a simple microservices architecture using Node.js. Each microservice will handle a specific part of a larger system, and they will communicate with each other using HTTP.

**1. Setup Your Project**

**Assignment 1: Initialize Projects for Microservices**

**- Instructions:**
  1. Initialize two separate Node.js projects for microservices.
  2. Name them `user-service` and `order-service`.
  3. Install necessary dependencies (`express` for the server).

**2. Implement User Service**

**Assignment 2: Create a Basic User Service**

**- Instructions:**
  1. Implement a simple `user-service` that manages user data.
  2. Include endpoints to get a user by ID and create a user.

**3. Implement Order Service**

**Assignment 3: Create a Basic Order Service**

**- Instructions:**
  1. Implement an `order-service` that manages orders.
  2. Include endpoints to get an order by ID and create an order.
  3. The order should reference a user ID.

**4. Implement Service Communication**

**Assignment 4: Enable Communication Between Services**

**- Instructions:**
  1. Update the `order-service` to fetch user information from the `user-service` when creating an order.
  2. Use `axios` to make HTTP requests between services.

**5. Test the Microservices**

**Assignment 5: Test CRUD Operations**

- Instructions:
  1. Use tools like `curl`, `Postman`, or `httpie` to test the endpoints of both services.
  2. Verify that creating a user and an order works as expected, and that the order includes user details.

**6. Implement API Gateway**

**Assignment 4: Create API Gateway**
**Instructions:**

Implement an api-gateway that routes requests to user-service and order-service.
Include endpoints for fetching users and orders through the API Gateway.

**7. Test the API Gateway**
**Assignment 5: Test API Gateway Endpoints**

- **Instructions**:
    1. Use tools like curl, Postman, or httpie to test the endpoints of the API Gateway.
    2. Verify that the API Gateway correctly routes requests to the user-service and order-service.