**Project Document**

**Backend**

**Server Setup**

* This is a Node.js Express server that listens for HTTP requests on a specified port. Here's an overview of what the code does:
* First, it loads the necessary dependencies: dotenv for loading environment variables from a .env file, express for creating an HTTP server, cors for enabling Cross-Origin Resource Sharing (CORS) middleware, and several custom route modules for handling requests related to user authentication and task management.
* It establishes a connection to the database using connection() function.
* The Express middleware functions express.json() and cors() are called, which allow parsing of incoming JSON payloads and cross-origin requests, respectively.
* The server sets up routes for handling HTTP requests related to user authentication and task management by registering the appropriate middleware functions exported by the route modules.
* Finally, the server starts listening for incoming HTTP requests on the specified port number using the app.listen() method. If the server starts up successfully, a message is logged to the console.

**User schema**

* Import required packages/modules such as mongoose, jwt, Joi, and passwordComplexity.
* Create a Mongoose schema for the user collection with fields such as firstName, lastName, email, and password.
* Add a method to the userSchema called "generateAuthToken" which generates a JSON Web Token (JWT) using the user's id and a private key from the environment variables.
* Create a Mongoose model "User" based on the userSchema.
* Define a validation function using the Joi package to validate user input data. This function checks that the firstName, lastName, email, and password fields are present, and the email is in the correct format, and the password meets the password complexity requirements.
* Export the User model and the validation function for use in other parts of the application.

**Task schema**

* This code exports a module that defines a taskSchema using the Mongoose schema constructor to specify the structure of a Task document in a MongoDB database. The taskSchema has two fields: Title of type String and user\_id of type String.
* The module exports three Mongoose models: createTask, updateTask, and deleteTask, each of which corresponds to a CRUD operation that can be performed on a Task document. However, these models are not used in the code and may be unnecessary.
* The module also exports a validate function that takes an object as input and uses Joi to validate that the object has a Title field of type String and a user\_id field of type String. If the input object is invalid, the function will throw a validation error.

**Routes**

**Login (Authentication)**

* This is a server-side JavaScript code that handles a POST request to the endpoint "/".
* The first step is to import necessary dependencies such as the router module from the Express framework, the User model from the user.js file, the bcrypt library for password encryption, and the Joi module for data validation.
* After that, a POST request handler is created using the router module. Inside the handler, the request body is validated using the validate function, which uses Joi schema to ensure that the email and password fields are present and formatted correctly.
* Then, the code attempts to find a user with the email provided in the request body using the User model's findOne method. If no user is found, it returns a 401 status code with an error message.
* If a user is found, the password is compared with the encrypted password stored in the database using the bcrypt library's compare method. If the passwords do not match, the code returns a 401 status code with an error message.
* If the passwords match, a JSON web token is generated using the user's ID and the private key stored in the .env file. The token and the user's ID are then sent as a response to the client.
* Finally, the validate function is defined to ensure that the email and password fields are present and formatted correctly. If the fields are not present or formatted incorrectly, a validation error is returned.
* The code exports the router module to make it available for use in other files.

**Sign Up**

* This is a Node.js module that exports a router object that handles HTTP POST requests to create a new user.
* First, the router imports the User model and a validate function from "../models/user" and the bcrypt library.
* Then, the router defines a route handler function that listens for a POST request to the root URL path "/".
* Inside the route handler, it calls validate function to validate the user input in the request body against a Joi schema defined in the User model.
* If the validation fails, it sends a response with a 400 Bad Request status and an error message.
* If the validation passes, it searches the database for a user with the same email as the one provided in the request body. If it finds one, it sends a response with a 409 Conflict status and an error message indicating that a user with the given email already exists.
* If it doesn't find a user with the given email, it generates a salt and hash from the provided password using bcrypt, and then creates a new instance of User model with the user input and hashed password. Then it saves the new user to the database and sends a response with a 201 Created status and a success message.
* If there is any error during the process, it sends a response with a 500 Internal Server Error status and an error message.

**CRUD Task**

**Create Task**

* Import the necessary modules: The code imports the Express router and the createTask model from the ../models/Task module.
* Define the route: The code defines an HTTP POST route for creating a new task at the root endpoint ('/').
* Request body validation: The code uses the validate function from the ../models/Task module to validate the request body sent by the client. If the request body is invalid, the server sends a 400 Bad Request response with an error message.
* Create and save the task: If the request body is valid, the server creates a new task by creating an instance of the createTask model and saving it to the database. The title and user\_id fields of the task are set to the corresponding values in the request body.
* Send the response: If the task is successfully created and saved, the server sends a 201 Created response with a success message. If there is an error, the server sends a 500 Internal Server Error response with an error message.
* Export the router: The code exports the router so that it can be used in other parts of the application.

**Update Task**

* The express module is required, and an instance of the Router class is created.
* The updateTask model and the validate function are imported from the ../models/Task module.
* The router listens to a PATCH request to the endpoint specified in the router.patch() method.
* When a request is received, the function defined as the second argument to the router.patch() method is executed.
* The validate function is called to validate the request body, and if there's an error, a response with a 400 status code and an error message is sent back to the client.
* If the request body is valid, the updateTask model's updateOne() method is called to update the task with the specified id in the request body with the Title provided.
* If the update is successful, a response with a 201 status code and a success message is sent back to the client.
* If there's an error, a response with a 500 status code and an error message is sent back to the client.

**Get User Task**

* This is an implementation of a GET API endpoint that fetches a list of tasks belonging to a particular user. Here are the steps:
* First, we import the necessary modules, in this case, express and createTask model from the ../models/Task.js file.
* We define a GET endpoint for the / route using router.get("/", async (req, res) => {...});.
* Inside the try block, we extract the user\_id from the request body and set some pagination parameters: page, limit, and skip. These parameters will be used to limit the number of results we get from the database and to provide pagination.
* We use createTask.find({ user\_id: userId }).skip(skip).limit(limit); to query the database and find all tasks that belong to the specified user id. The skip and limit parameters are used to implement pagination.
* Finally, we return the list of tasks as a JSON response: res.json(tasks);.
* If there's an error, we catch it and send an appropriate error response.
* **Delete Task**This is an Express route for deleting a task.
* First, it imports the express module and creates a new router object.
* It also imports the deleteTask function from the ../models/Task module.
* When a DELETE request is sent to this route, it executes an asynchronous function that does the following:
* It calls the deleteTask function to delete the task with the given \_id.
* If the deletion is successful, it sends a 201 status code with a success message in the response.
* If there's an error, it sends a 500 status code with an error message in the response.
* Finally, the route is exported as a module.

**Frontend**

**Login**

* This is a React functional component that renders a login form for users to log in to their account. The component starts by importing necessary dependencies such as React, useState, axios, and Link from React Router. It also imports a CSS module that contains styles for the component.
* The component sets up a state with two empty strings for the user's email and password. The state is updated with every keystroke in the input fields using the handleChange function, which uses event delegation to get the input element's name and value and update the state using the useState hook.
* The component's main function is the handleSubmit function, which is called when the form is submitted. The function first prevents the default form submission behavior and then sends a POST request to the authentication endpoint using the axios library. If the email address is invalid, the function sets an error message, which is rendered in the component.
* If the request is successful, the response object contains a token and the user's ID, which are stored in the browser's localStorage object. The user is then redirected to the homepage.
* The component returns a div with a className of "login\_container", which contains two nested divs with className "login\_form\_container". The left div contains the login form, while the right div contains a link to the sign-up page. The form contains input fields for the email and password, a submit button, and an error message that is displayed if there is an error with the input.
* The link to the sign-up page is a React Router link component, and the button in the link has a className of "white\_btn" for styling.

**Sign Up**

* This is a React functional component named Signup that handles user sign-up functionality. It uses the useState hook to define the component state with an initial state object that has keys for firstName, lastName, email, and password. It also defines error state and initializes it to an empty string.
* It also imports the axios library for handling HTTP requests and Link and useNavigate components from the react-router-dom library for navigation. It also imports styles from an external CSS file named styles.module.css.
* The component renders a sign-up form that contains input fields for firstName, lastName, email, and password. It also contains a button for submitting the form.
* The handleChange function is called each time any of the input fields' values change. It takes an event object as an argument and uses the currentTarget property of the event to get the input field's name and value. It then updates the component state using the setData function with the new values.
* The handleSubmit function is called when the user submits the sign-up form. It takes an event object as an argument and uses the preventDefault method to prevent the form from submitting and refreshing the page. It then checks whether the email entered by the user is valid or not using a regular expression. If it is not valid, it sets the error state to a message asking the user to enter a valid email address and returns from the function. If the email is valid, it makes an HTTP POST request to the server with the user's sign-up data. If the request is successful, it logs a message to the console and navigates the user to the login page using the navigate function. If the request fails with an error response, it sets the error state to the error message received from the server.
* The component then returns the JSX that defines the structure and layout of the sign-up form.

**Main Dashboard**

Based on the code you provided, it looks like a task management application with the following features:

* Add a new task to the list with a title and user id.
* Edit an existing task by opening an edit modal and changing the task title.
* Delete a task from the list.
* Toggle the completion status of a task.
* Filter the tasks by all, completed, or incomplete.
* Search for tasks by title.
* Pagination for tasks displayed on the screen.
* User authentication and logout feature.

The tasks are stored in an array called tasks, which is initially empty. When the component mounts, the useEffect hook runs and calls the fetchTasks function, which makes a GET request

to the server to get the user's tasks from the API. The response is stored in the tasks state array using the setTasks function.

The component has several state variables:

1. data: an object representing the new task to be added to the list.
2. filter: a string representing the current filter setting.
3. search: a string representing the current search query.
4. editModalOpen: a boolean indicating whether the edit modal is currently open.
5. editModalTask: an object representing the task being edited in the edit modal.
6. error: a string representing any errors that occur while adding a new task.

The handleNewTaskChange function updates the data state object as the user types in the title input field. The handleNewTaskSubmit function is called when the user submits the form to add a new task. It sends a POST request to the server with the new task data, then adds the new task to the tasks array if it does not already exist.

* The handleFilterChange and handleSearchChange functions update the filter and search state variables, respectively, as the user interacts with the filter and search input fields.
* The handleDelete function removes a task from the tasks array when the user clicks the delete button for that task.
* The handleEdit function opens the edit modal and sets the editModalTask state object to the task being edited.
* The handleEditModalClose function closes the edit modal.
* The handleEditModalSubmit function is called when the user submits the edit modal form. It updates the tasks array with the new task data and closes the edit modal.
* The handleToggleComplete function toggles the completed property of a task when the user clicks the checkbox for that task.

The filteredTasks and searchedTasks variables are derived state variables that filter the tasks array based on the filter and search state variables.

The component renders a navbar with a logout button, a task input form, a filter dropdown, a search input field, a pagination component, and a list of tasks. The task list displays the task title, a checkbox indicating the completion status of the task, an edit button, and a delete button for each task. When the edit button is clicked, the handleEdit function is called and the edit modal is opened with the task title pre-filled.

**App Component**

The App component is a functional component. The first line of the function gets the token from localStorage and assigns it to the user variable.

The component returns a Routes component, which contains a collection of Route components. Here are the routes and their explanations:

{user && <Route path="/" exact element={<Main />} />}: If the user is logged in (user is truthy), render a Route component that matches the exact path / and renders the Main component.

<Route path="/signup" exact element={<Signup />} />: Always render a Route component that matches the exact path /signup and renders the Signup component.

{!user && <Route path="/login" exact element={<Login />} />}: If the user is not logged in (user is false), render a Route component that matches the exact path /login and renders the Login component.

{user && <Route path="/login" exact element={<Navigate to="/" />} />}: If the user is logged in, render a Route component that matches the exact path /login and immediately navigates to the home page (/) using the Navigate component.

{!user && <Route path="/" element={<Navigate to="/login" replace />} />}: If the user is not logged in, render a Route component that matches any path and immediately navigates to the login page (/login) using the Navigate component. The replace prop is used to replace the current URL in the history stack, so that the back button won't take the user back to the home page.

Export the App component.