Documentation for “To Do App”

Contents

Rest API Endpoints:

- POST:

```
// Create a new task and add it to the array
app.post("/tasks/todo", async (req, res) => {
  try {
    const { title, description, dueDate } = req.body;

    const taskData = { title, description, dueDate }; // grabs data
    const createTask = new Task(taskData); // Creates a new "Task" model with the data grab
    const newTask = await createTask.save();

    res.json({ task: newTask, message: "New task created successfully!" });
  } catch (error) {
    console.error("Error:", error);
    res.status(500).json({ message: "Error creating tasks!" });
  }
});
```

➢ Responsible for: Creating a new task and saving it to the database

- GET:

```
app.get("/tasks", async (req, res) => {
  try {
    const { sortBy } = req.query; // ?sortBy=duedate or ?sortBy=DateCreated

    let sortOption = {};

    if (sortBy === "dueDate") {
      sortOption = { dueDate: 1 }; //Ascending\
    } else if (sortBy === "dateCreated") {
      sortOption = { dateCreated: 1 };
    }

    const tasks = await Task.find({});
    res.json(tasks);
  } catch (error) {
    console.error("Error:", error);
    res.status(500).json({ message: "Error grabbing tasks!" });
  }
});
```

➢ Responsible for: Grabbing all the tasks from the database to display them on “dashboard.html”. It is also responsible for sorting the task into a given order(Default, Due Date, Date Created.

- PATCH(make task complete):

```
// to complete task
app.patch("/tasks/complete/:id", async (req, res) => {
  try {
    const { completed } = req.body;
    const taskId = req.params.id;

    const completedTask = await Task.findByIdAndUpdate(taskId, { completed }, { new: true });

    if (!completedTask) {
      return res.status(404).json({ message: "task not found" });
    }
    res.json({ task: completedTask, message: "task set to complete" });
  } catch (error) {
    console.error("Error:", error);
    res.status(500).json({ message: "Error completing the task!" });
  }
});
```

- Responsible for: Updating the completed value of each task to “true”, which in turn “completes” the task and allows it to be rendered in the appropriate column in the dashboard.

- PATCH (Make task not complete):

```
// to not complete task
app.patch("/tasks/notComplete/:id", async (req, res) => {
  try {
    const { completed } = req.body;
    const taskId = req.params.id;

    const taskNotComplete = await Task.findByIdAndUpdate(taskId, { completed }, { new: true });

    if (!taskNotComplete) {
      return res.status(404).json({ message: "task not found" });
    }
    res.json({ task: taskNotComplete, message: "task set to 'not complete'" });
  } catch (error) {
    console.error("Error:", error);
    res.status(500).json({ message: "Error setting the task to 'not complete'" });
  }
});
```

- Responsible for: Updating the completed value of each task to “false”, which in turn “uncompletes” the task and allows it to be rendered in the appropriate column in the dashboard.

- DELETE:

```
// To Delete the task
app.delete("/tasks/delete/:id", async (req, res) => {
  try {
    const taskId = req.params.id;

    const deletedTask = await Task.findByIdAndDelete(taskId);

    if (!deletedTask) {
      return res.status(404).json({ message: "Task not found" });
    }

    res.json({ task: deletedTask, message: "Task deleted successfully" });
  } catch (error) {
    console.error("Error:", error);
    res.status(500).json({ message: "Error deleting the task!" });
  }
});
```

➢ Responsible for: remove task in the database using the ID

- PUT:

```
// To edit task

app.put("/tasks/update/:id", async (req, res) => {
  try {
    const taskId = req.params.id;
    const { title, description, dueDate } = req.body;

    const taskData = { title, description, dueDate };
    console.log(taskData);
    const updatedTask = await Task.findByIdAndUpdate(taskId, taskData, { new: true });

    if (!updatedTask) {
      return res.status(404).json({ message: "Task not found" });
    }

    res.json({ task: updatedTask, message: "Task updated successfully!" });
  } catch (error) {
    console.error("Error:", error);
    res.status(500).json({ message: "Error editing the task!" });
  }
});
```

➢ Responsible for: Edit task in the database

Depedencies:

Dependency Versions:

- **Cors(^2.8.5)** > Enables CORS for cross Origin Resource Sharing – Allows only certain access from specified domain to access and interact the server.
- **Dotenv(^17.2.1)** > Loads environmental variables into the project from the .env file
- **Express(^5.1.0)** > Web framework for Node.js – Used for all APIs
- **Mongoose(^8.17.1)** > Framework/library which allows easy interaction with MongoDB database.
- **Nodemon(^3.1.10)** > Dev tool that allows auto -restarting of the server

Installation of Dependencies

- **RUN:** 'cd backend'
- **RUN:** 'npm install' in the 'backend' folder > installs all packages listed in the package.json file

Credentials:

```
MONGO_URI=mongodb+srv://ossyokis:dsT0YicXoITmd2qs@todoapp.dmk7uoa.mongodb.net/?retryWrites=true&w=majority&appName=ToDoApp
PORT=3000
```

- MONGO_URL: connection string for secure access to MongoDB database
- PORT: Port number (on the device) the app listens on (during development)

Procedure for Feedback:

Collection Methods:

- Users can give feedback via comments/forums on the app website
- Users can submit feedback via feedback form also on the website
- User feedback can be obtained through specific user testing

Process:

1. Feedback considered, evaluated and approved for changes
2. Timeline created on the project management platform tasks/updates assigned to developers
3. Code is updated/changed, then ready for testing
4. Code is tested through various test (QA, Security etc)
5. Seek final approval from all relevant stakeholders, approval on tested change
6. Changes pushed live into production
7. Seek feedback on changes made.

Procedure for code commit changes:

1. Document the process for code commit changes. This involves creating a feature branch from the develop branch, making and testing changes locally, and committing them with descriptive messages.

2. Push the branch to the remote repository, then create a pull request for review. Address any feedback received, and once approved, merge the changes into the develop branch.

3. After thorough testing, merge into the master branch. Post-merge, delete the feature branch and pull the latest changes from develop. Ensure CI/CD pipelines run tests and deploy automatically.

4. Follow best practices by committing frequently, using clear messages, participating in reviews, maintaining tests, and updating documentation. For issues like merge conflicts, resolve them locally, and contact the repository admin if access issues occurs