Documentation for “To Do App”

Table of Contents

REST API Endpoints

GET:

```
app.get("/api/tasks", async (req, res) => {

 try {
  const { sortBy} = req.query;
  let sortOption = {};

  if (sortBy === "dueDate") {···
  } else if (sortBy === "dateCreated") {
    sortOption = {dateCreated: 1};
  }



  const tasks = await Task.find({}).sort(sortOption);

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



  res.json(tasks);

 } catch (error) {
   console.error("Error:", error);  //Look for http error status cod
   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 tasks into a given order from the user input (Default, Due Date, Date Created).

POST:

```
app.post("/api/tasks/todo", async (req, res) => {
  try {
    const {title, description, dueDate } = req.body;


    const taskData = {title, description, dueDate };
    const createTask = new Task(taskData);
    const newTask = await createTask.save();


      res.json({ message: "Task created successfully!", task: newTask});

  } catch (error) {
    console.error("Error:", error);  //Look for http error status codes 200 = ok for example
    res.status(500).json({message: "Error creating task!"});
  }
});
```

Responsible for creating a new task and saving it to the database.

PATCH:

```
app.patch ("/api/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 “completes” the task and allows it to be rendered in the appropriate column on the “Dashboard.html” screen.

PATCH:

```
app.patch ("/api/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 making the task not complete!"});
  }
});
```

Responsible for updating the completed value of each task to “false”, which makes the task “not complete” and allows it to be rendered in the appropriate column on the “Dashboard.html” screen.

DELETE:

```
app.delete("/api/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!"});

    }
    (parameter) res: Response<any, Record<string, any>, number>
    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 removing tasks in the database using the ID that was parsed into the function.

EDIT:

```
app.put("/api/tasks/update/:id", async (req, res) => {

  try {
    const taskId = req.params.id;
    const {title, description, dueDate } = req.body;

    const taskData = {title, description,dueDate};
    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 updating the task!"});

  }
});
```

Responsible for editing the task values inside the database, such as: Title, Description or Due Date.

Dependencies

- **"cors": "^2.8.5"** Enables CORS for Cross Origin Resource Sharing – Allows only certain access from specified domains to access and interact with the server.

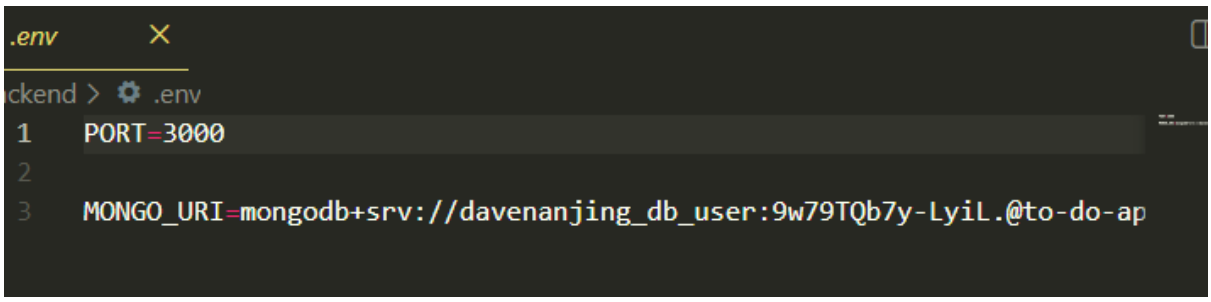
- **"dotenv": "^17.2.3"** Loads environmental variable into the server file from the .env file – allows us to store sensitive info in a different hidden file.

- **"express": "^5.1.0"** Express is a backend web framework for Node.JS - used to create APIs with less code, and also to carry out other backend functions with ease and minimal setup.

- **"mongoose": "^9.0.0"** Mongoose is the library of tools we use to access the MongoDB database.

- **"nodemon": "^3.1.11"** It is the development tool that allows auto restarting of the server.

Credentials



Mongo_URI – This is the connection string for secure access to MongoDB database. The access portal into the database.

Port 3000 (on the device where the server is live).

Procedures for Feedback

Collection Methods:

- Feedback collected from emails and surveys.
- User specific testing on certain parts of the app / certain features.
- Feedback collected via a form either on the website or when emailed to them.

Process:

1. Ask for feedback.
2. Feedback received.
3. Feedback considered, evaluated and approved for changes
4. Review and follow up with the client who provided feedback
5. Timeline created on project management platform and tasks /updates assigned to developers.
6. Code is updated/changed, when ready for testing
7. Code is tested through various tests (QA, Security, Accessibility, Performance etc.)
8. Seek final review from all relevant stakeholders. Gather approval on tested changes.
9. Changes pushed live into production
10. Start process again by seeking feedback on the 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 occur.