

INTEGRATION

CSE5006 – LAB 8

REPOSITORY:

[HTTPS://GITHUB.COM/CSE5006/LAB-8](https://github.com/CSE5006/LAB-8)

TABLE OF CONTENTS

1. Introduction	3
1.1 Clone and Fork	3
2. Persistence	6
2.1. Create Task	7
2.2. Automatically Load Tasks	9
2.3. Checking off Tasks	13
2.4. Exercise 1	15

1. INTRODUCTION

Welcome to this exciting new lab where we will embark on a journey of integration, taking our knowledge of React, Sequelize, and PostgreSQL and weaving them together to form a single cohesive application.

By now, you are already familiar with React, a JavaScript library for building user interfaces. You have experienced the magic of designing components, managing state, and mastering props to render interactive frontend applications. But a full-featured web application is more than just its frontend. It's about the way we manage and interact with data on the server-side, and this is where Sequelize and PostgreSQL step into the picture.

In the backend universe, you've explored the realms of Sequelize, an elegant promise-based Object-Relational Mapping (ORM) that empowers us to manage data in SQL databases using JavaScript. You've navigated the SQL-based database system PostgreSQL, understanding how it enables us to store, manage, and manipulate our data efficiently.

This lab is about bringing together the individual pieces of the puzzle. We will create a full-stack web application by connecting a React frontend to a backend powered by Sequelize and PostgreSQL. We will dive into how the client (React) communicates with the server (Sequelize with PostgreSQL), how to manage the flow of data, and how changes on the server reflect in our React application.

It might seem like a complex task, but we assure you it's an incredible learning journey that you're embarking on. By the end of this lab, you will have successfully created a full-stack web application integrating React, Sequelize, and PostgreSQL, enriching your development skills and getting you closer to becoming a seasoned full-stack developer.

Ready to dive in? Let's get started!

In this lab, we will use the VM provided in LMS. The username/password is vboxuser/vboxuser.

1.1 CLONE AND FORK

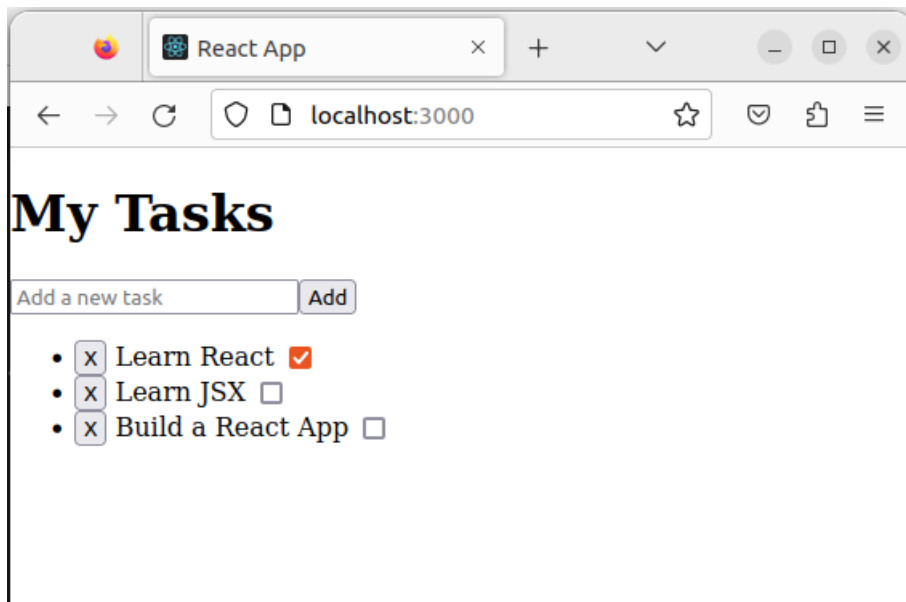
1. Fork the react-task-list-interactive project from <https://github.com/CSE5006/lab-8> as your own private GitHub repository.
2. Use the Git command line tool to clone a local copy of the repository.

```
vboxuser@CSE5006:~/Documents$ git clone https://github.com/CSE5006/lab-8
Cloning into 'lab-8'...
remote: Enumerating objects: 110, done.
remote: Counting objects: 100% (110/110), done.
remote: Compressing objects: 100% (61/61), done.
remote: Total 110 (delta 42), reused 97 (delta 32), pack-reused 0
Receiving objects: 100% (110/110), 325.73 KiB | 12.00 KiB/s, done.
Resolving deltas: 100% (42/42), done.
vboxuser@CSE5006:~/Documents$
```

3. Open a new terminal window and run the following command to start the server.

```
docker compose up --build
```

4. Point your web browser to <http://localhost:3000> in order to load the website after the container has finished starting.



Open up the entire lab directory in Visual Studio Code.

We have an entire express API and react frontend setup here to get started with.

1.2. REVERSE PROXYING WITH NGINX

Let's say that we have a server with the domain name www.example.com, with our API and frontend running on ports 5000 (line 33) and 3000 (line 22), respectively.

```
EXPLORER
LAB-8
  > api
  > frontend
  > nginx
  .gitignore
  docker-compose.yml
  README.md

docker-compose.yml
1  version: "3.8"
2
3  services:
4    nginx:
5      build:
6        context: ./nginx
7        dockerfile: Dockerfile
8      ports:
9        - 80:80
10     depends_on:
11       - frontend
12       - api
13     links:
14       - frontend
15       - api
16
17     frontend:
18       build:
19         context: ./frontend
20         dockerfile: Dockerfile
21       ports:
22         - 3000:3000
23       depends_on:
24         - api
25       volumes:
26         - ./frontend:/app
27
```

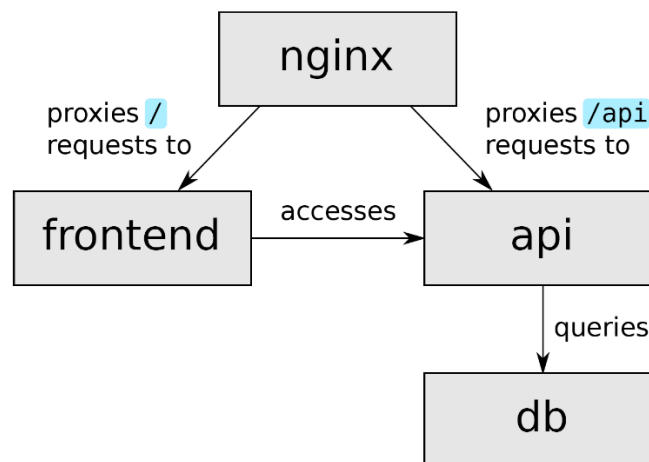
From the browser's perspective, these are two different web addresses: www.example.com:5000 and www.example.com:3000. What we want is to use www.example.com (default port 80) for both, but select the server depending on the path. To achieve this, we can use a reverse proxy, such as NGINX. NGINX will run on port 80 and reroute incoming requests to the appropriate internal server. For example, we can write a rule that says all HTTP requests with paths beginning with /api should go to localhost:5000, and everything else should go to localhost:3000.

Here's what the part of the NGINX configuration for this would look like (you don't need to copy this code anywhere, it's included here for reference only) – it is also found in default.conf:

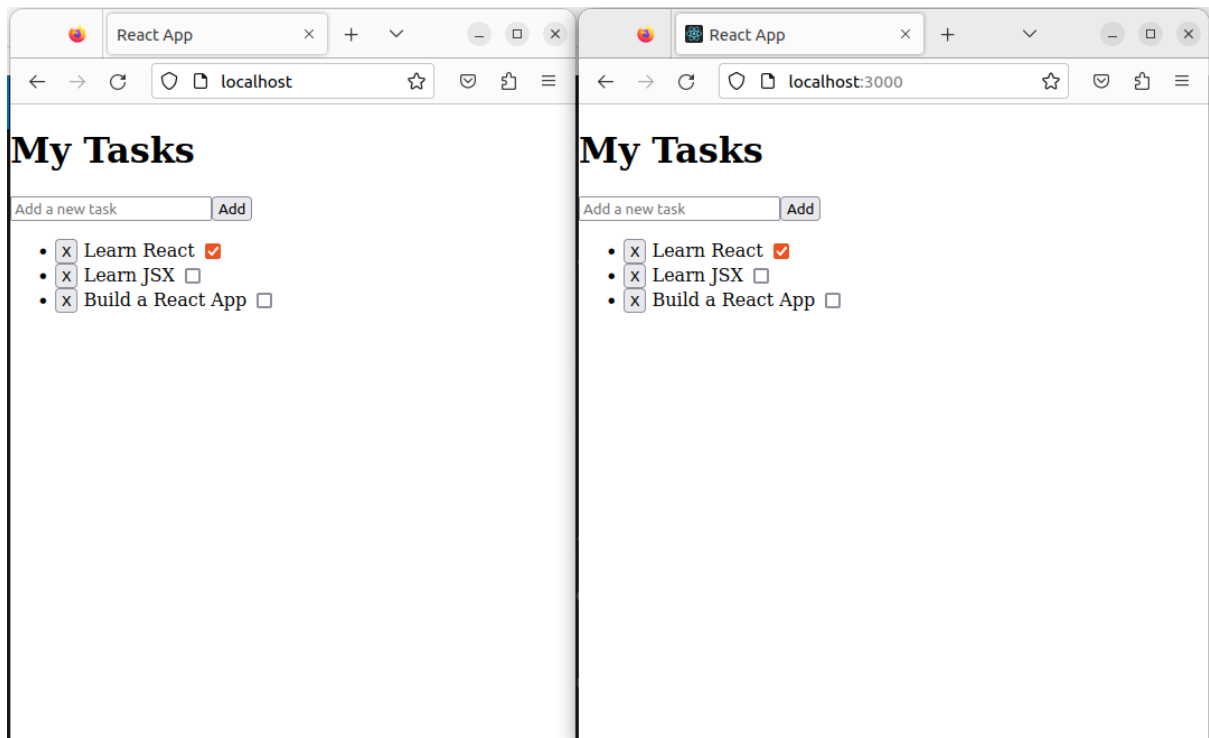
```
location / {  
    proxy_pass http://frontend:3000;  
}  
  
location /api/ {  
    proxy_pass http://api:5000;  
}
```

An appropriately configured NGINX Docker image is already available for you in the nginx/ subdirectory of blog-lab08. There is no need to change anything in there, but feel free to look around.

Since the number of services has grown somewhat since last lab, it may help to visualise the services and the links between them as a diagram, and compare this to what is written in docker-compose.yml.



Now open the browser and try to open the application using <http://localhost> and compared the result with <http://localhost:3000>



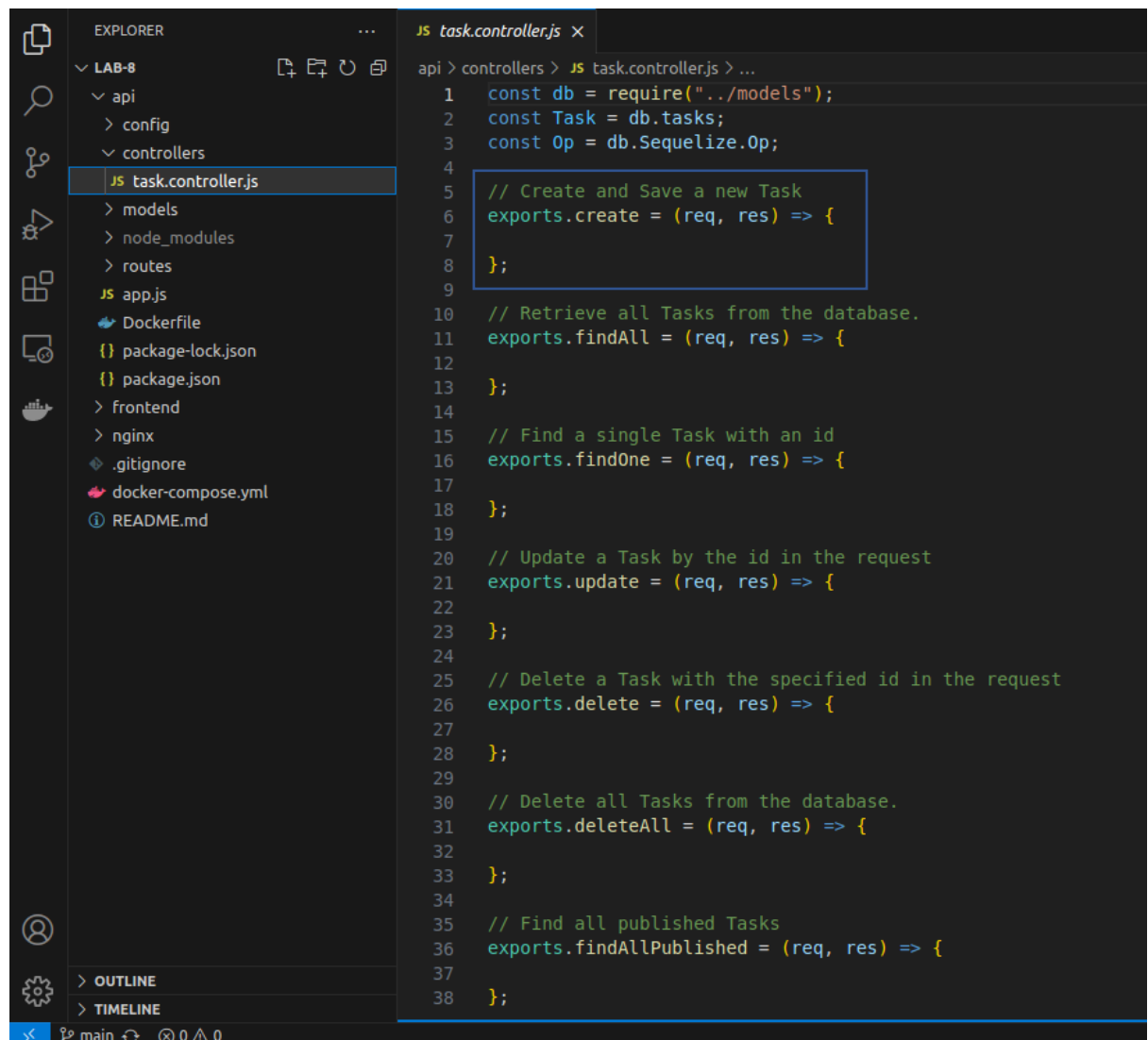
You may try to add new task, remove a task from the list or put the check mark on the list as well. Once you have finished with the lists, try to refresh the application by using the refresh button on your browser or press “F5” key on your keyboard. What do you find?

2. PERSISTENCE

Putting everything we’ve learned together will allow us to build a task list that persists between closing our browser! We will be making use of the fetch API in React to make the necessary requests in the backend.

2.1. CREATE TASK

1. Add the API logic to the create endpoint in task.controller.js, directing the server to put a record in the database:



The screenshot shows the Visual Studio Code editor with the Explorer sidebar on the left and the task.controller.js file open in the main editor. The Explorer sidebar shows a project structure with folders like LAB-8, api, controllers, models, node_modules, routes, and files like app.js, Dockerfile, package-lock.json, package.json, frontend, nginx, .gitignore, docker-compose.yml, and README.md. The task.controller.js file is highlighted in the Explorer. The main editor shows the code for task.controller.js, which includes database connections and various API endpoints. A blue box highlights the create endpoint logic.

```
api > controllers > JS task.controller.js > ...
1  const db = require("../models");
2  const Task = db.tasks;
3  const Op = db.Sequelize.Op;
4
5  // Create and Save a new Task
6  exports.create = (req, res) => {
7
8  };
9
10 // Retrieve all Tasks from the database.
11 exports.findAll = (req, res) => {
12
13 };
14
15 // Find a single Task with an id
16 exports.findOne = (req, res) => {
17
18 };
19
20 // Update a Task by the id in the request
21 exports.update = (req, res) => {
22
23 };
24
25 // Delete a Task with the specified id in the request
26 exports.delete = (req, res) => {
27
28 };
29
30 // Delete all Tasks from the database.
31 exports.deleteAll = (req, res) => {
32
33 };
34
35 // Find all published Tasks
36 exports.findAllPublished = (req, res) => {
37
38 };
```

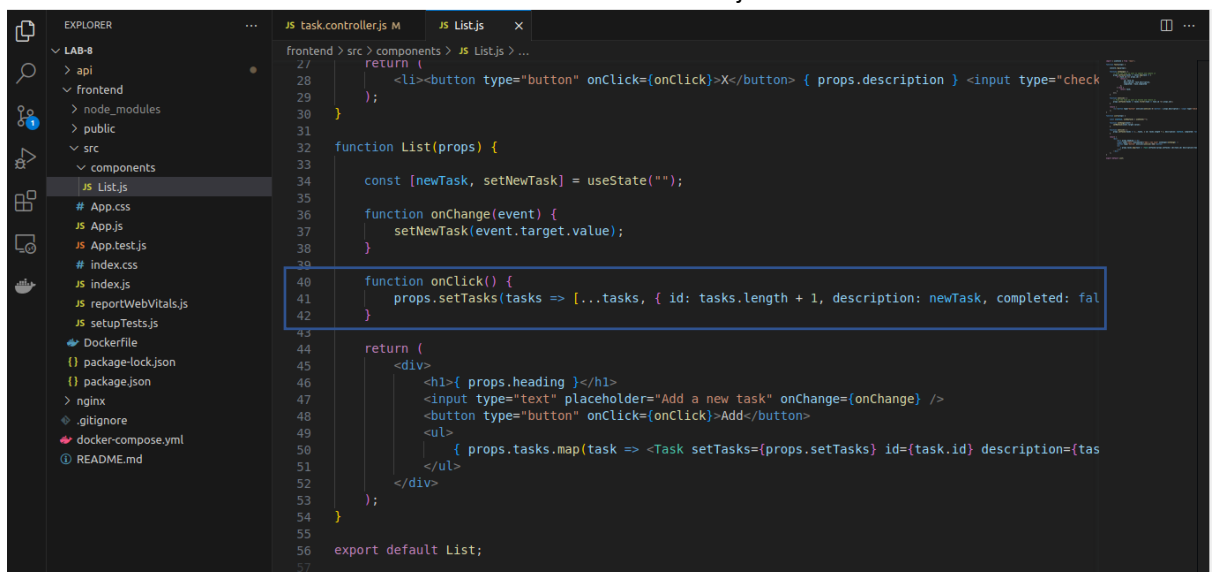
```

exports.create = (req, res) => {
  const task = {
    description: req.body.description,
    completed: req.body.completed || false
  };

  Task.create(task)
    .then(data => {
      res.send(data);
    })
    .catch(err => {
      res.status(500).send({
        message:
          err.message || "Some error occurred"
      });
    });
};

```

2. Time to infuse the fetch function call into the React frontend in List.js.



```

27  return (
28    <li><button type="button" onClick={onClick}>X</button> { props.description } <input type="checkbox"
29  );
30  }
31  }
32  function List(props) {
33  }
34  const [newTask, setNewTask] = useState("");
35  function onChange(event) {
36    setNewTask(event.target.value);
37  }
38  function onClick() {
39  }
40  props.setTasks(tasks => [...tasks, { id: tasks.length + 1, description: newTask, completed: false }]);
41  }
42  }
43  return (
44    <div>
45      <h1>{ props.heading }</h1>
46      <input type="text" placeholder="Add a new task" onChange={onChange} />
47      <button type="button" onClick={onClick}>Add</button>
48      <ul>
49        { props.tasks.map(task => <Task setTasks={props.setTasks} id={task.id} description={task.description} />)}
50      </ul>
51    </div>
52  );
53  }
54  export default List;
55

```

Replace the onClick function in the List component with:


```

function onClick() {
  fetch('http://localhost/api/tasks', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({ description: newTask, completed: false })
  })
  .then(response => response.json())
  .then(data => {
    props.setTasks(tasks => [...tasks, data]);
  })
  .catch((error) => {
    console.error('Error:', error);
  });

  setNewTask(""); // Clear the input field
}

```

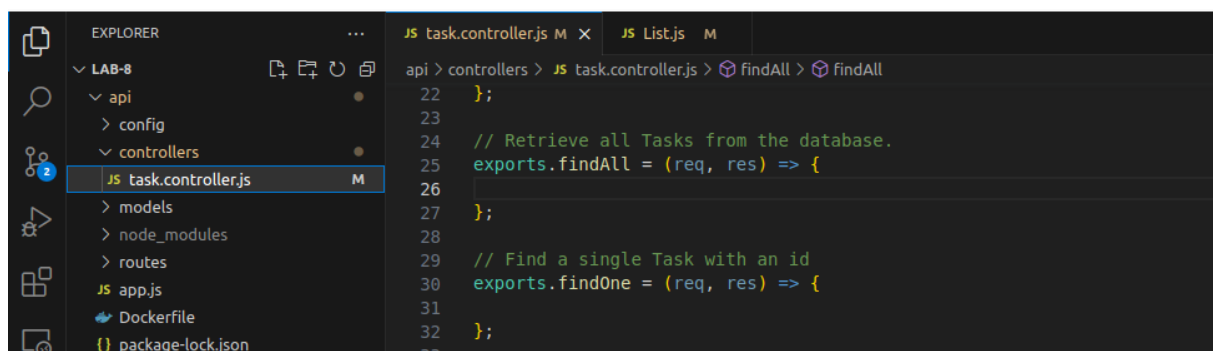
Wasn't that easy? Onto the next!

Note: in this step, you have created a method to store the data in the database. However, you can't see the data when you refresh the app since you haven't created the method to load the data from the database. The next following step will guide you how to load the data from the database.

Tasks: Create a new task, named "Wash the laundry". Once you have created it, refresh your application. Can you see your new task?

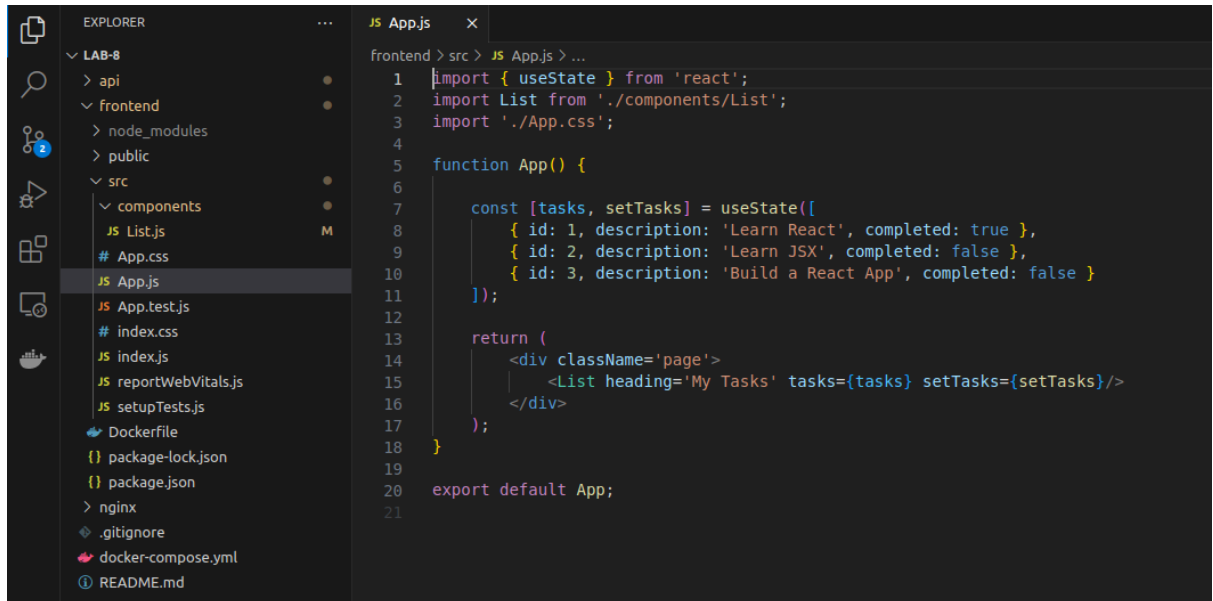
2.2. AUTOMATICALLY LOAD TASKS

1. Once again, spin the API logic into the findAll endpoint in task.controller.js, enabling the server to gift tasks to the frontend:



```
// Retrieve all Tasks from the database.
exports.findAll = (req, res) => {
  Task.findAll()
    .then(data => {
      res.send(data);
    })
    .catch(err => {
      res.status(500).send({
        message: err.message || "Some error occurred"
      });
    });
};
```

2. Now, let's tweak the code in Frontend.



```
1 import { useState } from 'react';
2 import List from './components/List';
3 import './App.css';
4
5 function App() {
6
7     const [tasks, setTasks] = useState([
8         { id: 1, description: 'Learn React', completed: true },
9         { id: 2, description: 'Learn JSX', completed: false },
10        { id: 3, description: 'Build a React App', completed: false }
11    ]);
12
13    return (
14        <div className='page'>
15            <List heading='My Tasks' tasks={tasks} setTasks={setTasks}/>
16        </div>
17    );
18
19
20    export default App;
21
```

Instead of using predefined list, modify App.js to summon the useEffect hook and load the tasks when the page is born anew:

```
import { useState, useEffect } from 'react'; // import useEffect
import List from './components/List';
import './App.css';

function App() {

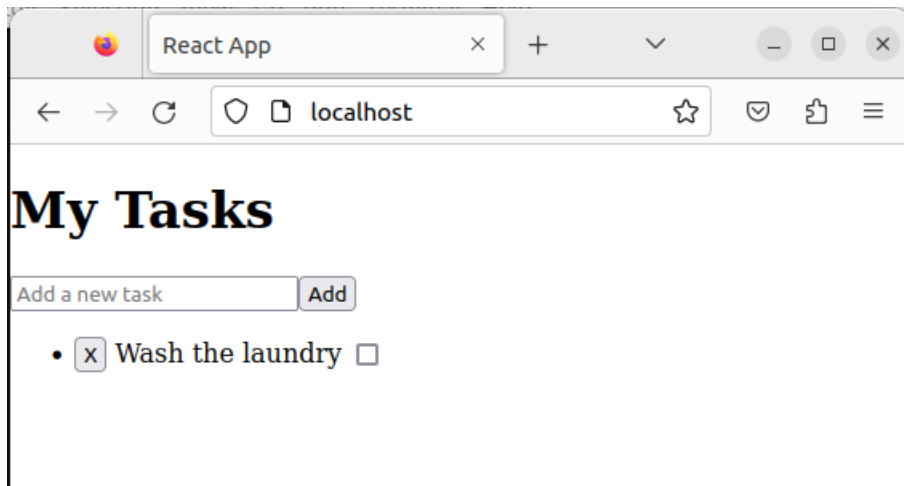
    const [tasks, setTasks] = useState([]);

    useEffect(() => {
        fetch('http://localhost/api/tasks')
            .then(response => response.json())
            .then(data => setTasks(data))
            .catch((error) => {
                console.error('Error:', error);
            });
    }, []);

    return (
        <div className='page'>
            <List heading='My Tasks' tasks={tasks} setTasks={setTasks}/>
        </div>
    );
}

export default App;
```

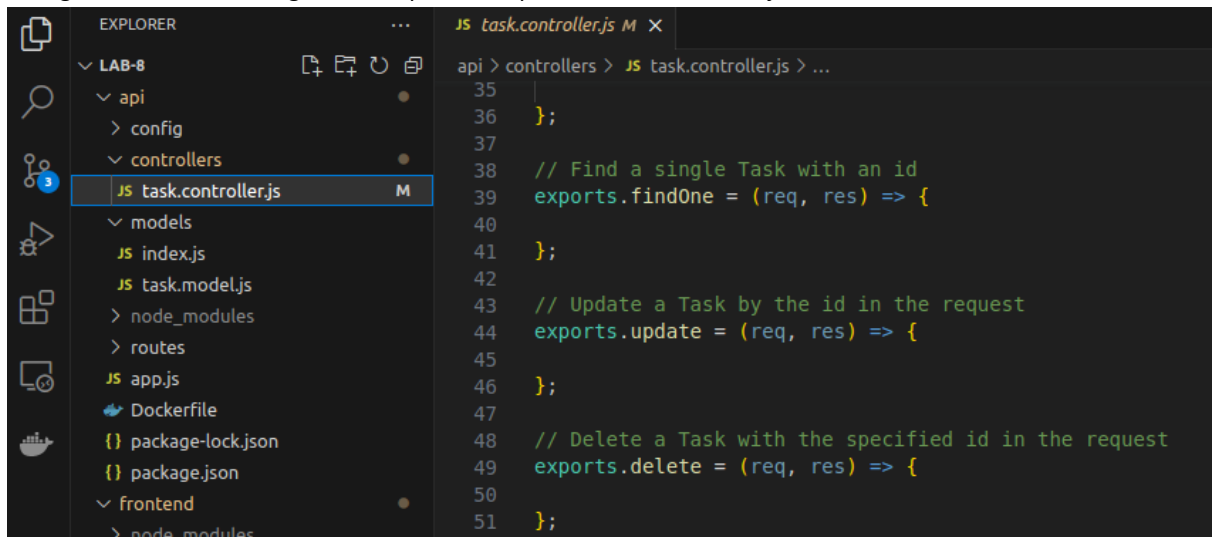
Try refreshing the page after adding a new task. Close the tab, reopen it, and be dazzled as your tasks persist!



2.3. CHECKING OFF TASKS

What good is a task list if you can't celebrate completed tasks with triumphant ticks? Let's get that working!

1. Yet again, stir in the API logic to the update endpoint in task.controller.js.

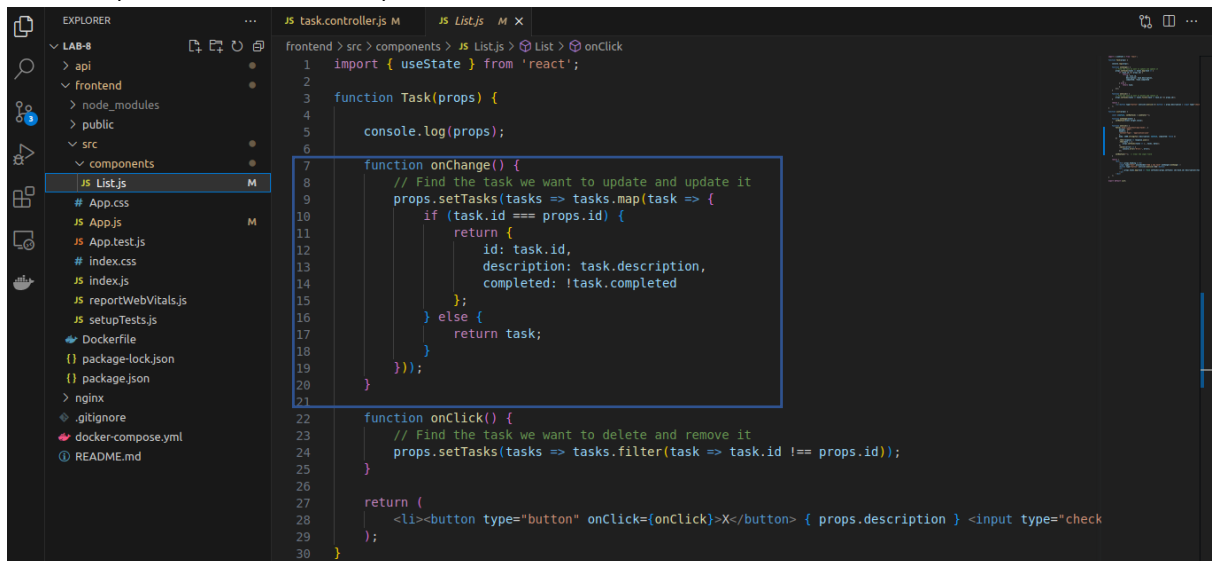


```
35 |  
36 | };  
37 |  
38 | // Find a single Task with an id  
39 | exports.findOne = (req, res) => {  
40 |  
41 | };  
42 |  
43 | // Update a Task by the id in the request  
44 | exports.update = (req, res) => {  
45 |  
46 | };  
47 |  
48 | // Delete a Task with the specified id in the request  
49 | exports.delete = (req, res) => {  
50 |  
51 | };
```

Now we will activate the update function in API, enabling the server to pass tasks to the frontend. Modify the update method by using the following script:

```
// Update a Task by the id in the request  
exports.update = (req, res) => {  
  const id = req.params.id;  
  
  Task.update(req.body, {  
    where: { id: id }  
  })  
  .then(num => {  
    if (num == 1) {  
      res.send({  
        message: "Task was updated successfully."  
      });  
    } else {  
      res.send({  
        message: `Cannot update Task`  
      });  
    }  
  })  
  .catch(err => {  
    res.status(500).send({  
      message: "Error updating Task with id=" + id  
    });  
  });  
};
```

2. We can now change the onChange function in List.js for the Task component to employ the API to make sure the update method can be implemented.



```
1 import { useState } from 'react';
2
3 function Task(props) {
4   console.log(props);
5
6
7   function onChange() {
8     // Find the task we want to update and update it
9     props.setTasks(tasks => tasks.map(task => {
10       if (task.id === props.id) {
11         return {
12           id: task.id,
13           description: task.description,
14           completed: !task.completed
15         };
16       } else {
17         return task;
18       }
19     }));
20   }
21
22   function onClick() {
23     // Find the task we want to delete and remove it
24     props.setTasks(tasks => tasks.filter(task => task.id !== props.id));
25   }
26
27   return (
28     <li><button type="button" onClick={onClick}>X</button> { props.description } <input type="checkbox"
29   );
30 }
```

```
function onChange() {
  const updatedTask = {
    id: props.id,
    description: props.description,
    completed: !props.completed
  };

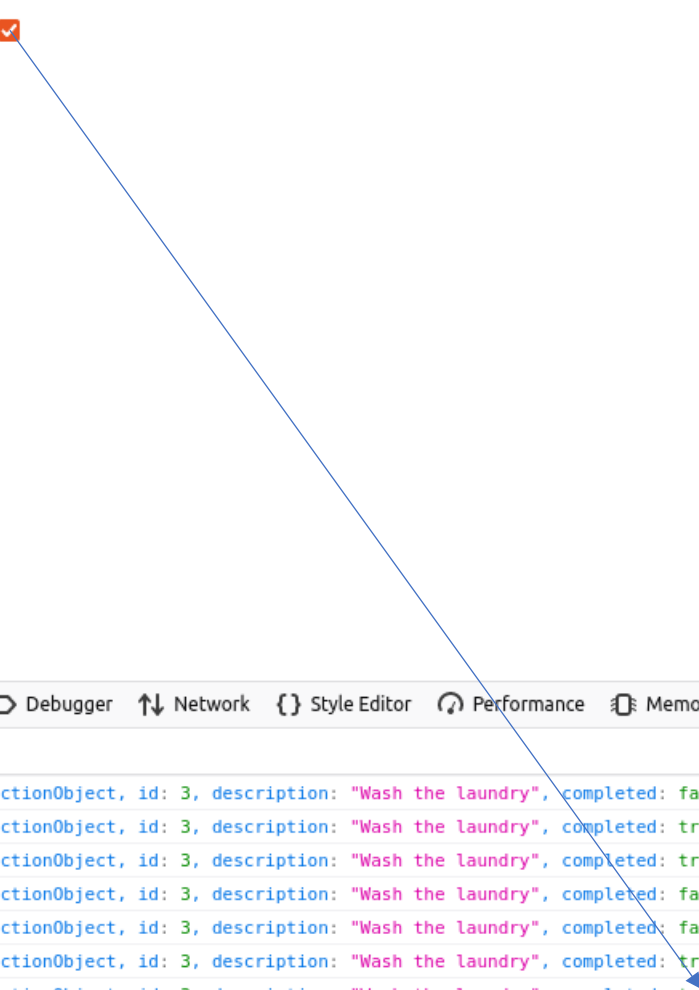
  fetch(`http://localhost/api/tasks/${props.id}`, {
    method: 'PUT',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(updatedTask)
  })
  .then(response => response.json())
  .then(() => {
    props.setTasks(tasks => tasks.map(task => {
      if (task.id === props.id) {
        return updatedTask;
      } else {
        return task;
      }
    }));
  })
  .catch((error) => {
    console.error('Error:', error);
  });
}
```

Try fiddling with the checkbox on some tasks, then refresh and watch your changes endure!

You can also open the “Developer Tools” by pressing “F12”.

My Tasks

- ☐ Wash the laundry ☒



Inspector Console Debugger Network Style Editor Performance Memory Storage

Filter Output

```
▶ Object { setTasks: BoundFunctionObject, id: 3, description: "Wash the laundry", completed: false }
▶ Object { setTasks: BoundFunctionObject, id: 3, description: "Wash the laundry", completed: true }
▶ Object { setTasks: BoundFunctionObject, id: 3, description: "Wash the laundry", completed: true }
▶ Object { setTasks: BoundFunctionObject, id: 3, description: "Wash the laundry", completed: false }
▶ Object { setTasks: BoundFunctionObject, id: 3, description: "Wash the laundry", completed: false }
▶ Object { setTasks: BoundFunctionObject, id: 3, description: "Wash the laundry", completed: true }
▶ Object { setTasks: BoundFunctionObject, id: 3, description: "Wash the laundry", completed: true }
```

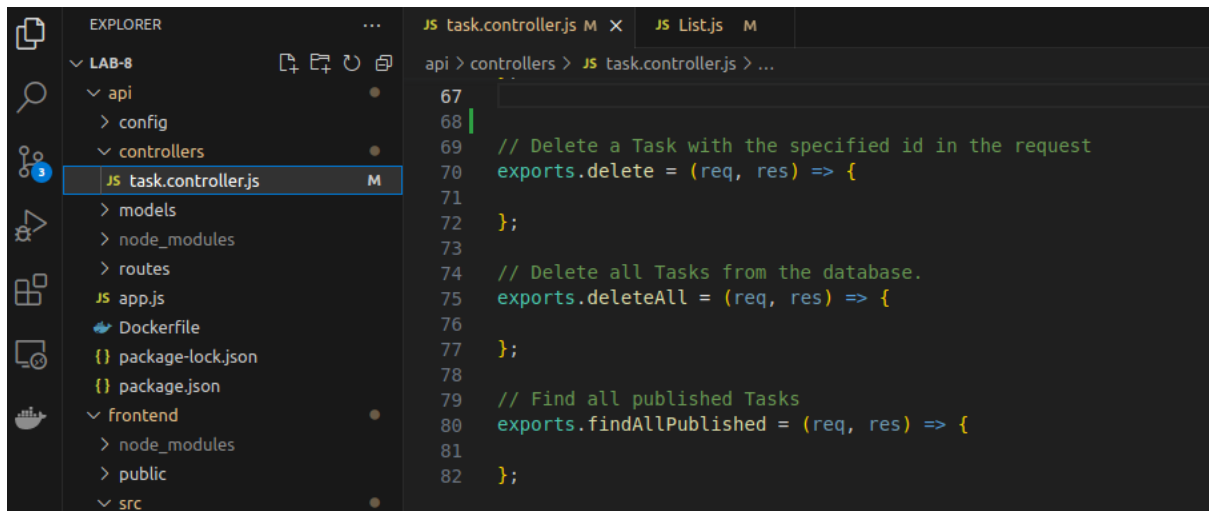
>>

2.4. EXERCISE 1

I think you’ve got the right idea. Now go and make the delete button for tasks work with the database! Make sure you check your code by refreshing the page to see if the tasks you’ve deleted are gone from the database.

Answer:

Open `api/controllers/task.controller.js` and locate the delete method.

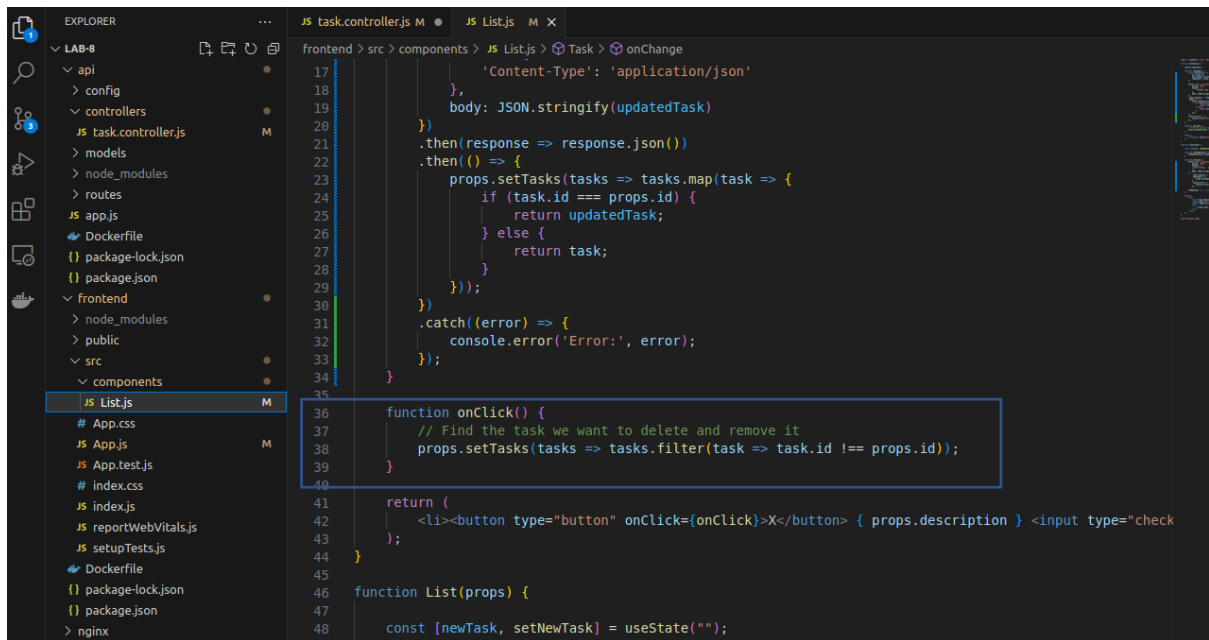


Update the delete method with the following script

```
69 // Delete a Task with the specified id in the request  
70 exports.delete = (req, res) => {  
71     const id = req.params.id;  
72  
73     Task.destroy({  
74         where: { id: id }  
75     })  
76     .then(num => {  
77         if (num == 1) {  
78             res.send({  
79                 message: "Task was deleted successfully!"  
80             });  
81         } else {  
82             res.send({  
83                 message: `Cannot delete Task`  
84             });  
85         }  
86     })  
87     .catch(err => {  
88         res.status(500).send({  
89             message: "Could not delete Task with id=" + id  
90         });  
91     });  
92 };
```

Now, update the frontend, so that the delete button will call the delete method in API

Open frontend/src/components/List.js



Update this function using the following script:

```

36 function onClick() {
37   // Find the task we want to delete and remove it
38   fetch('http://localhost/api/tasks/${props.id}', {
39     method: 'DELETE',
40   })
41   .then(() => {
42     // remove it from the state
43     props.setTasks(tasks => tasks.filter(task => task.id !== props.id));
44   })
45   .catch((error) => {
46     console.error('Error:', error);
47   });
48 }
49
50 return (
51   <li><button type="button" onClick={onClick}>X</button> { props.description } <input type="checkbox"
52 </li>
53 );

```

Save your project, and refresh your app in the browser.