There will be three steps

1. Server side (Back-end)
2. Data Base
3. Client side (Front-end)

1.Server side (Back-end):

Create a directory for your server-side code. You can name it "server-side":

Initializing npm
In the "server-side" directory, initialize npm by running the following command:

```
npm init -y
```

This command will generate a package.json file with default settings.

Initialize TypeScript in your project by creating a tsconfig.json file. Run:

```
tsc --init
```

Installing Required Libraries

Install the necessary libraries using npm. Run the following commands one by one:

- To install Node.js types for TypeScript:
  `npm install -D @types/node`
- To install the Express library:
  `npm install express`
- To install the CORS library:
  `npm install cors`
- To install the SQL library for Node.js (msnodesqlv8):
  `npm install msnodesqlv8`

Creating the Main Server File
Inside the "server-side" directory, create a file named index.ts

```
const express = require("express");
```

This imports the 'express' module, which provides a framework for building web applications in Node.js.

```
const cors = require('cors');
```

Imports the 'cors' module, which is used to handle Cross-Origin Resource Sharing and allows the server to respond to requests from different origins.

```
const app = express();
```

Creates an instance of the Express application.

```
app.use(cors());
```

Adds the 'cors' middleware to the Express application, enabling CORS for all routes.

```
const sql = require('msnodesqlv8');
```

Imports the 'msnodesqlv8' module, which is used to interact with SQL Server databases.

```
const
connectionString="server=serverName;Database=dbName;Trusted_Connection=Yes;Driv
er={SQL Server Native Client 11.0}";
```

Defines the connection string for the SQL Server database. This string includes the server name, database name, authentication mode, and driver details.

```
 app.get('/tasks', async({ req, res }:any) => {

      var query = 'SELECT * FROM Tasks';

      sql.query(connectionString,query, (err:any,rows:any)=>{
            console.log("Data coming from DB",rows);
            res.json(rows);
      })
      });
```

This code sets up an endpoint /tasks that handles GET requests. When a client makes a GET request to this endpoint, the code retrieves all rows from a database table named Tasks using an SQL query. The retrieved data is then sent back to the client as a JSON response, allowing clients to fetch information from the Tasks table.

```
 app.post('/tasks', async (req: any, res :any) => {

      const { name, status } = req.body;
      var query = `INSERT INTO Tasks (name, status) VALUES (?, ?)`;

      sql.query(connectionString,query,[name, status], (err:any,rows:any)=>{
            if (err) {
            console.log(err);
            } else {
            console.log("Data inserted in DB", rows);
            res.json(rows);
            }
      })
      });
```

This code defines a route /tasks that handles HTTP POST requests. When a POST request is made to this endpoint, the code extracts the name and status values from the request's body. It then constructs an SQL query to insert a new record into a database table named Tasks, using the extracted name and status values. The code executes this query, and if successful, it logs the inserted data to the console and responds with a JSON containing the inserted data. Essentially, this code allows clients to add new tasks by sending POST requests to the /tasks endpoint.

```
app.put('/tasks/:id', async (req :any, res:any) => {
      try {
            console.log("parameters value", req.params);
      const taskId = req.params.id;
      const { name, status } = req.body;

      const query = `UPDATE Tasks SET name = ?, status = ? WHERE id
=${taskId}`;


      sql.queryRaw(connectionString, query, [name,status],(err:any, rows:any)
=> {
      if (err) {
      console.log(err);
      } else {
```

```
        console.log("Data updated in DB", rows);
        res.json('Task Updated');
        }
        });
        } catch (error) {
        console.error(error);
        res.status(500).send('Server Error');
        }
        });
```

This code handles PUT requests to the /tasks/:id endpoint. When a PUT request is made, the provided id parameter identifies the task to be updated. The code extracts the name and status values from the request and constructs an SQL query to update the corresponding task's data in the Tasks table. Upon executing the query, successful updates are logged, and a JSON response confirms the task's update. In case of errors, appropriate error handling is performed, logging errors and returning a 500 status response. This code enables task updates through PUT requests to the specific /tasks/:id endpoint.

```
app.delete('/tasks/:id', async (req: any, res: any) => {
        try {
        const taskId = req.params.id;
        var query =  `DELETE FROM Tasks WHERE id = ${taskId}`
        sql.query(connectionString,query, (err:any,rows:any)=>{
                console.log("Data updated in DB",rows);
                res.json("Task deleted");
        })

        } catch (error) {
        console.error(error);
        res.status(500).send('Server Error');
        }
        });
```

This code defines an endpoint at /tasks/:id to handle HTTP DELETE requests. When a client sends a DELETE request to this endpoint, the id parameter in the URL specifies the task to be deleted. The code extracts this id and constructs an SQL query to delete the corresponding task from the Tasks table. The query is executed using the provided connection string, and upon successful deletion, a log message is generated, and a JSON response confirms the task's deletion. In the event of errors, the code handles them by logging the errors and responding with a 500 status to indicate a server error. This code allows clients to delete tasks by sending DELETE requests to the /tasks/:id endpoint.

Executes the SQL query using the connection string and handles the result.

`app.listen(4000,()=>{...}`

Starts the Express server on port 4000. When the server is successfully started, it logs a message

After add code in index.ts file then run your back end project by running below command

```
tsc
node index
```

Your project will run on below url

http://localhost:4000/data

2. Database setup:

Step 1: Download and Install Microsoft SQL Server:

- Visit the official Microsoft SQL Server Downloads page: https://www.microsoft.com/en-us/sql-server/sql-server-downloads
- Choose the edition of SQL Server you want to download. For most users, the "Express" edition is a good choice as it's free and suitable for learning and small applications.
- Follow the on-screen instructions to download the installer executable (typically an .exe file).
- Run the downloaded installer executable.
- During the installation process, choose the appropriate installation type and follow the prompts. You can typically choose the default settings for most options.

Step 2: Connect to SQL Server using Windows Authentication:

- After installing SQL Server, you'll need to connect to it using a SQL Server management tool. One popular choice is "SQL Server Management Studio" (SSMS), which can be downloaded separately from:

  https://www.microsoft.com/en-us/sql-server/sql-server-downloads

- Open SQL Server Management Studio (SSMS) after installation.

- In the "Connect to Server" dialog:
    - ❖ Server Type: Choose "Database Engine."
    - ❖ Server Name: Enter the name of your SQL Server instance. This could be the server's name or its IP address.
    - ❖ Authentication: Choose "Windows Authentication."
    - ❖ Click the "Connect" button. If the connection is successful, you should be connected to the SQL Server instance.

Step 3: Create a Database and Table:

In SQL Server Management Studio:

- In the "Object Explorer" panel on the left, right-click on "Databases" and choose "New Database."
- Give your database a name, e.g., "taskList."
- Click "OK" to create the database.

With the "storeDb" database selected:

- Right-click on "Tables" under your new database and choose "New Table."
- Design your table by adding columns. For example, you can add columns like "id," "taskName," "status," etc.
- Set the appropriate data types and constraints for each column.
- Click the "Save" button to save the table.

Your "taskList" database with the "tasks" table is now set up. You can use this database to interact with your backend code.

3. Client Side (Front-end)

Create a New React App

Open your terminal or command prompt.

To create a new React app, you can use npx (a package runner tool that comes with npm) with the following command:

```
npx create-react-app my-react-app
```

Replace my-react-app with the desired name for your project. This command will set up a new React app in a folder with the specified name.

Install Tailwindcss for styling

```
npm install -D tailwindcss
npx tailwindcss init
```

Further details please visit https://tailwindcss.com/docs/guides/create-react-app

Editing the App
Open the src/App.js file in a text editor or an integrated development environment (IDE).

You can start editing the App component or create new components in separate files within the src directory.

```
const fetchTasks = () => {
      fetch('http://localhost:4000/tasks')
      .then(response => response.json())
      .then(data => {
      setTasks(data);
      })
      .catch(error => {
      console.error(error);
      });
  };
```

In above code fetchTasks responsible for retrieving task data from a specified server URL using a GET request. It converts the server's JSON response into data and updates the React component's tasks state with the retrieved information. Error handling is included to log errors to the console if encountered during the fetching process. This function essentially fetches task data from the server and updates the application's state to reflect the retrieved information.

```
 const handleDelete = (id) => {
      fetch(`http://localhost:4000/tasks/${id}`, {
      method: 'DELETE'
      })
      .then(response => {
      if (response.ok) {
      fetchTasks();
```

```
      } else {
      console.error('Failed to delete task');
      }
      })
      .catch(error => {
      console.error(error);
      });
  };
```

Above code defines a function called handleDelete responsible for deleting a task by sending an HTTP DELETE request to a specific server endpoint with the provided id. If the response from the server indicates success (HTTP status 200), the fetchTasks function is called to refresh the task list. Otherwise, if the response indicates an error, an error message is logged to the console. Any errors during the request are caught and logged as well. This function facilitates task deletion by communicating with the server and updating the task list accordingly.

```
const handleSaveTask = () => {
      if (selectedTask.id) {
      // Update existing task
      fetch(`http://localhost:4000/tasks/${selectedTask.id}`, {
      method: 'PUT',
      headers: {
      'Content-Type': 'application/json',
      },
      body: JSON.stringify(selectedTask),
      })
      .then(response => {
      if (response.ok) {
            fetchTasks();
            handleCloseModal();
      } else {
            console.error('Failed to update task');
      }
      })
      .catch(error => {
      console.error(error);
      });
      } else {
      const requestBody = {
      name: newTask.name,
```

```
      status: newTask.status
      };
      // Create new task
      fetch('http://localhost:4000/tasks', {
      method: 'POST',
      headers: {
      'Content-Type': 'application/json',
      },
      body: JSON.stringify(requestBody),
      })
      .then(response => {
      if (response.ok) {
            fetchTasks();
            handleCloseModal();
      } else {
            console.error('Failed to create task');
      }
      })
      .catch(error => {
      console.error(error);
      });
      }
```

The function handleSaveTask serves to either update an existing task or create a new one. If selectedTask.id is present, it sends an HTTP PUT request to update the task at http://localhost:4000/tasks/:id. The request includes the updated task data in JSON format, and if successful, it refreshes the task list using fetchTasks() and closes the modal dialog. If the response indicates an error, an error message is logged. If selectedTask.id is absent, indicating a new task, the function sends an HTTP POST request to http://localhost:4000/tasks with the new task data in JSON format. Similarly, it refreshes the task list, closes the modal, and logs errors if needed. This function manages task updates and creations by communicating with the server and updating the UI accordingly.

Run your front end project with following command

```
npm start
```