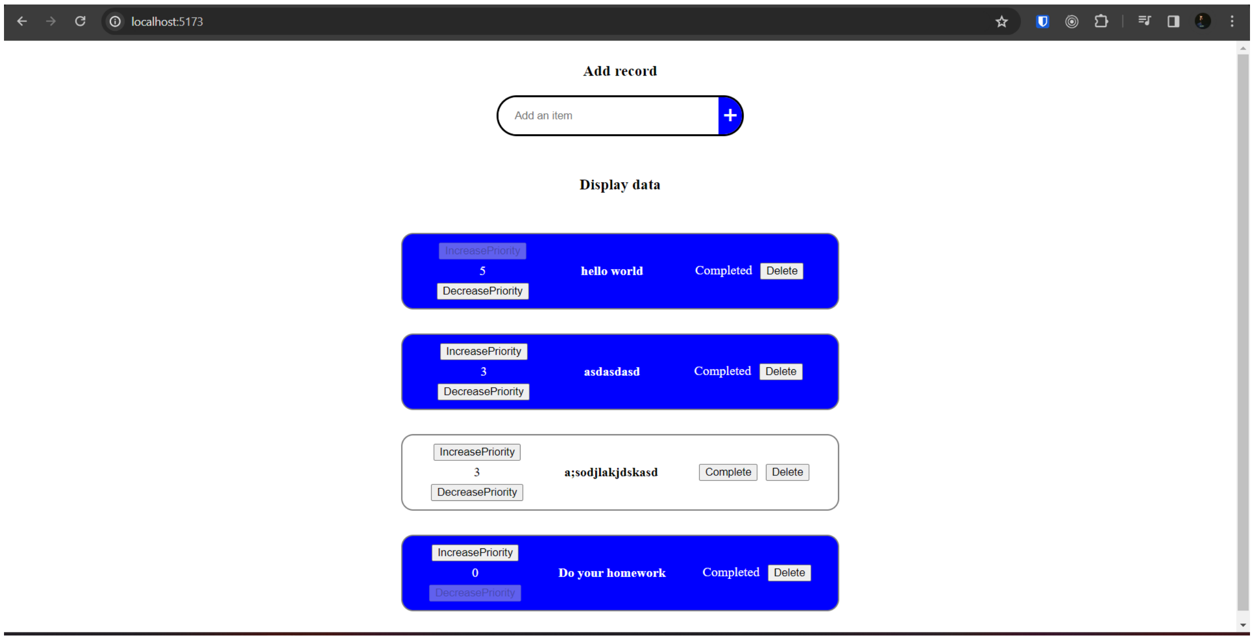
Assignment 1:: Simple Task Manager


localhost:5173

Add record

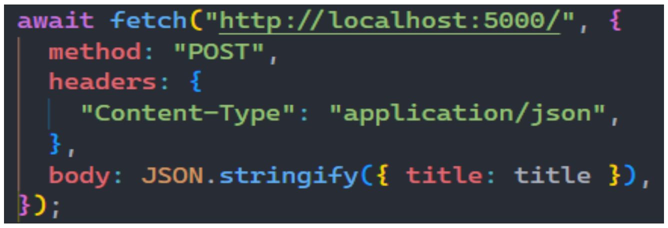
Add an item +

Display data

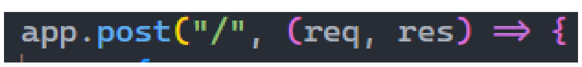
IncreasePriority 5 DecreasePriority	hello world	Completed	Delete
IncreasePriority 3 DecreasePriority	asdasdasd	Completed	Delete
IncreasePriority 3 DecreasePriority	a;sodjlakjdskasd	Complete	Delete
IncreasePriority 0 DecreasePriority	Do your homework	Completed	Delete

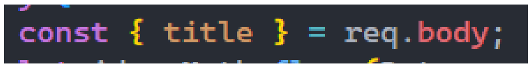
To represent tasks, I have used the help of **object** and **array** , or primarily I can say array of object. Understanding the question, what I can understand is, there should be an object because of multiple properties like **priority**, **completion** indicating the status. Since there can be multiple of such objects, I used an array to implement that. Firstly, **title** was taken as input from client side, and was sent to the server where other properties like id (integer), priority (integer) and complete (boolean) were added with some predefined values. Later with the endpoint that were defined using express, were used to operate on those records (property) stored in array and selected using id as id are set to be unique.

This application was implemented using **React** and **Node**. On the Frontend side, I used a hook useState, since it works as a variable or array and also automatically renders. I primarily used useState to get the value of title or todo work for this project. Also to get the record from the server and store it. Also I used Fetch api to communicate client side to the server of the project. I used the stringify method from JSON to convert object type to string since string can only be transferred. The data were transferred in the body, which can also be seen below.


```
await fetch("http://localhost:5000/", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({ title: title }),
});
```

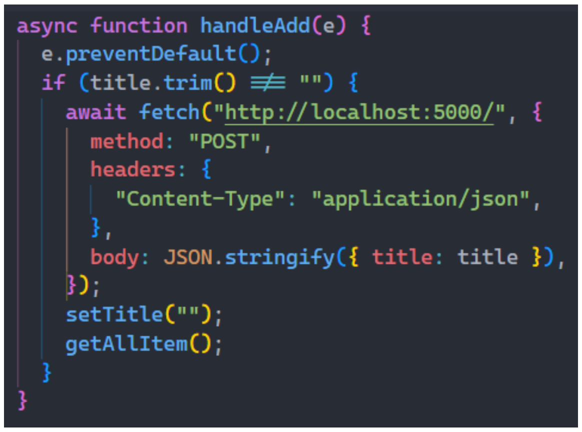
This body can be taken to its original form by destructuring the body located inside the request.


```
app.post("/", (req, res) => {
```

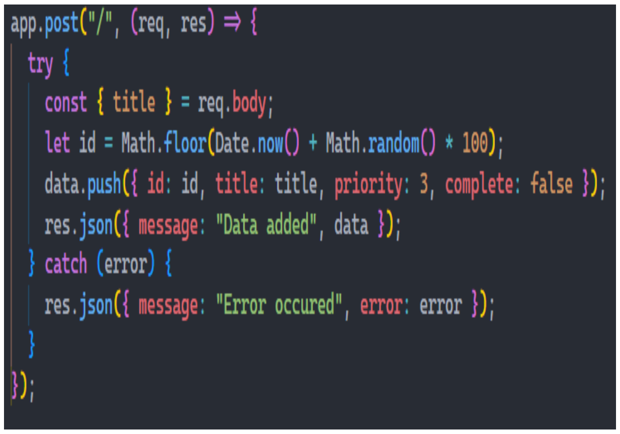

```
const { title } = req.body;
```

Task Addition and Deletion:

To add any task, title should be given, other necessary values would be populated in the backend side automatically before adding item to array. To add the item firstly from the client side, the information would be converted to string then sent to the end point as specified and on the server side, the body of request would be destructured to get the data. Also the **preventDefault** method helps to reload the page on submission of form.

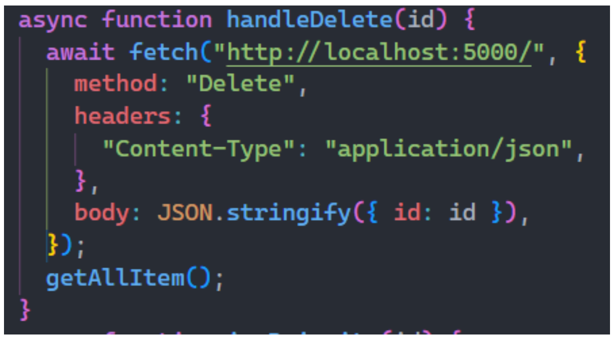


```
async function handleAdd(e) {
  e.preventDefault();
  if (title.trim() !== "") {
    await fetch("http://localhost:5000/", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ title: title }),
    });
    setTitle("");
    getAllItem();
  }
}
```

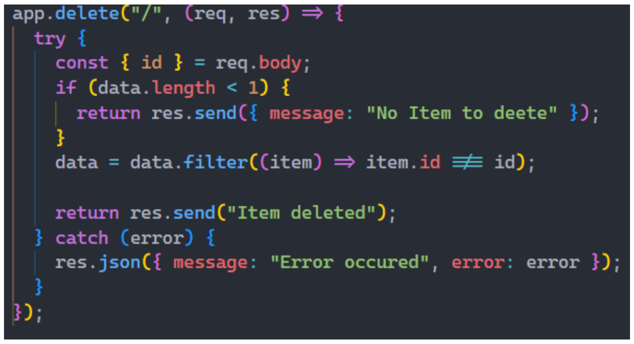


```
app.post("/", (req, res) => {
  try {
    const { title } = req.body;
    let id = Math.floor(Date.now() + Math.random() * 100);
    data.push({ id: id, title: title, priority: 3, complete: false });
    res.json({ message: "Data added", data });
  } catch (error) {
    res.json({ message: "Error occured", error: error });
  }
});
```

Similarly to delete any task, I used the help of ID rather than name since the name can be repeated, ID was made by the sum of date time and random number so the chance to be common would be very low.



```
async function handleDelete(id) {
  await fetch("http://localhost:5000/", {
    method: "Delete",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({ id: id }),
  });
  getAllItem();
}
```



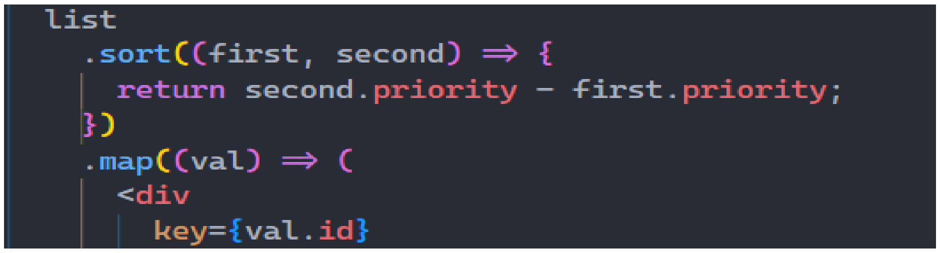
```
app.delete("/", (req, res) => {
  try {
    const { id } = req.body;
    if (data.length < 1) {
      return res.send({ message: "No Item to deete" });
    }
    data = data.filter((item) => item.id !== id);

    return res.send("Item deleted");
  } catch (error) {
    res.json({ message: "Error occured", error: error });
  }
});
```

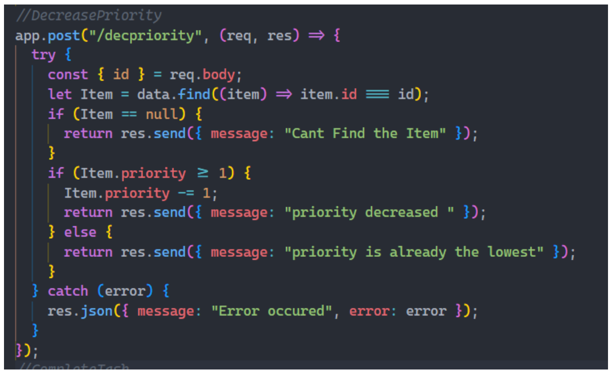
Sending name instead of id as input from the client side and destructuring and filtering by the name would do the operation to delete any item by name.

Task Prioritization:

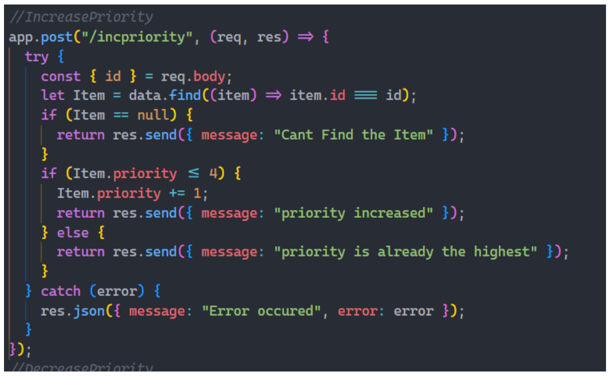
To list the item according to priority level , first I took a list that was populated by data with the help of useState with the help of getAllData function. This list was first sorted in descending order and then mapped to get each item and display information according to priority. Priority which initially was set to be 3 was updated by endpoint **incpriority** and **decpriority**.



```
list
  .sort((first, second) => {
    return second.priority - first.priority;
  })
  .map((val) => (
    <div
      key={val.id}
```



```
//DecreasePriority
app.post("/decpriority", (req, res) => {
  try {
    const { id } = req.body;
    let Item = data.find((item) => item.id === id);
    if (Item == null) {
      return res.send({ message: "Cant Find the Item" });
    }
    if (Item.priority >= 1) {
      Item.priority -= 1;
      return res.send({ message: "priority decreased " });
    } else {
      return res.send({ message: "priority is already the lowest" });
    }
  } catch (error) {
    res.json({ message: "Error occured", error: error });
  }
});
```

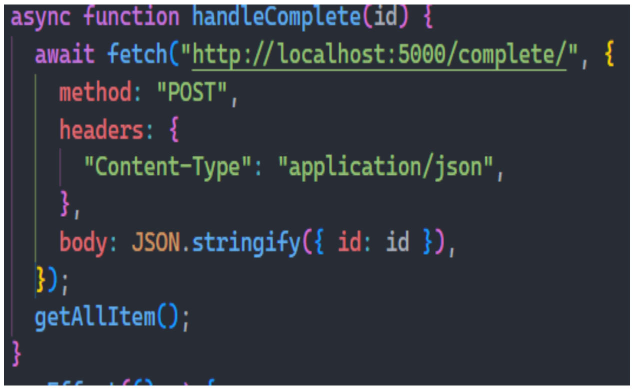


```
//IncreasePriority
app.post("/incpriority", (req, res) => {
  try {
    const { id } = req.body;
    let Item = data.find((item) => item.id === id);
    if (Item == null) {
      return res.send({ message: "Cant Find the Item" });
    }
    if (Item.priority <= 4) {
      Item.priority += 1;
      return res.send({ message: "priority increased" });
    } else {
      return res.send({ message: "priority is already the highest" });
    }
  } catch (error) {
    res.json({ message: "Error occured", error: error });
  }
});
```

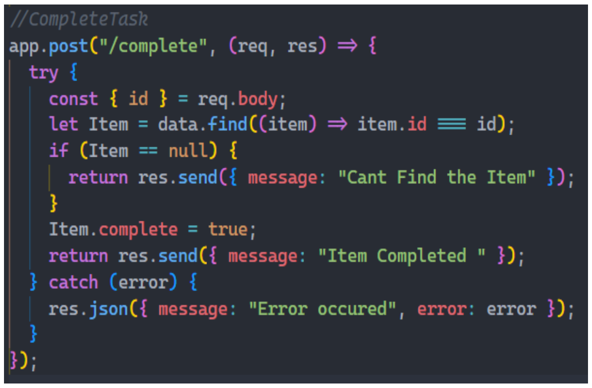
These were called by buttons which clicked on functions in the frontend, specific buttons would also be disabled if priority was 5 or 0 to mark the priority is already maximum or the minimum.

Task Completion:

Similar to other endpoints, there is also a button to send requests to the backend that would update the status of complete to be true from false that would indicate the completion of task. Similarly on frontend, the button would be changed to completed text and the background color.



```
async function handleComplete(id) {
  await fetch("http://localhost:5000/complete/", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({ id: id }),
  });
  getAllItem();
}
```



```
//CompleteTask
app.post("/complete", (req, res) => {
  try {
    const { id } = req.body;
    let Item = data.find((item) => item.id === id);
    if (Item == null) {
      return res.send({ message: "Cant Find the Item" });
    }
    Item.complete = true;
    return res.send({ message: "Item Completed " });
  } catch (error) {
    res.json({ message: "Error occured", error: error });
  }
});
```

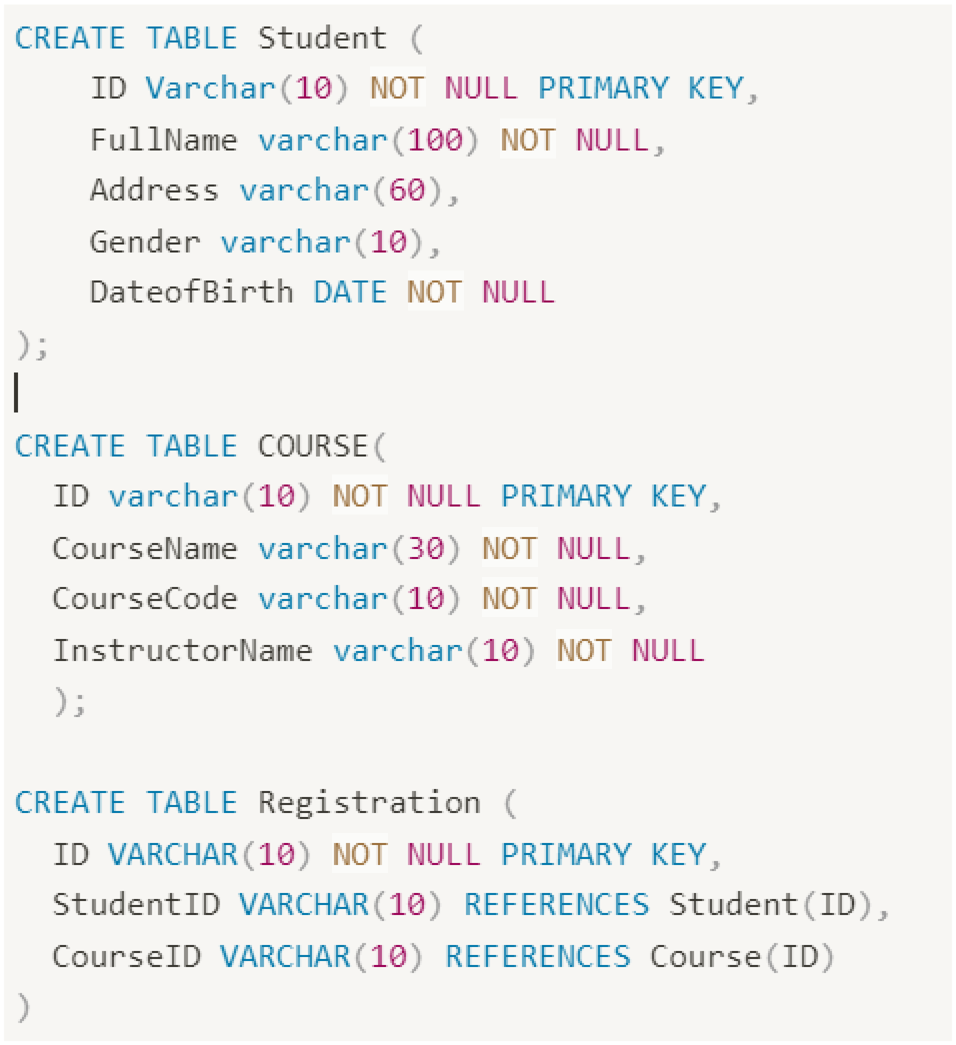
Assignment - 2 :: Student Course Registration System

For this question, From the given entities I assumed following attributes :

- Students (ID, FullName, Address, Gender, DateofBirth)
- Courses (ID, CourseName, CourseCode, InstructorName)
- Registration (ID, StudentID, CourseID)

where ID represents **Primary Key** and StudentID and CourseID are **Foreign Key** referring to ID in students and courses table. Registration referred or linked to Course by foreign key CourseID and Student by foreign key StudentID.

Respective table for the database was created by executing following queries:



```
CREATE TABLE Student (
    ID Varchar(10) NOT NULL PRIMARY KEY,
    FullName varchar(100) NOT NULL,
    Address varchar(60),
    Gender varchar(10),
    DateofBirth DATE NOT NULL
);

CREATE TABLE COURSE(
  ID varchar(10) NOT NULL PRIMARY KEY,
  CourseName varchar(30) NOT NULL,
  CourseCode varchar(10) NOT NULL,
  InstructorName varchar(10) NOT NULL
  );

CREATE TABLE Registration (
  ID VARCHAR(10) NOT NULL PRIMARY KEY,
  StudentID VARCHAR(10) REFERENCES Student(ID),
  CourseID VARCHAR(10) REFERENCES Course(ID)
)
```

Here, the attributes were declared as NOT NULL to say that there must be value or it can't be empty. **Foreign keys** were implemented on Registration by the help of REFERENCES keyword followed by table name and attribute to be assigned on registration table.

The tables can also be said to have a one to one relation between **Courses** to **Registration** and **Student** to **Registration**, due to which, to get common value between courses and students, we had to use a **Registration** table.

Queries

1. Retrieve a list of all students enrolled in a specific course.

```
SELECT S.FullName,S.ID as 'StudentID',C.COurseName
FROM Student AS S
JOIN Registration AS R
ON S.ID = R.StudentID
JOIN Course AS C
ON R.CourseID = C.ID
where C.CourseName="Physics";
```

Join gives us the freedom to link two tables by common values. They can also be cascaded to add more than 2 tables. To get a list of all students that have enrolled in a specific course, firstly, I wrote what was to be displayed and from which table that column should come. Using Join, table student and registration were joined since they had the StudentID (from registration) and ID (from student) as common and were referenced by the help of foreign key, and similarly for Course.

2. Get the details of a student's course registrations, including the course name and Instructor.

```
SELECT * FROM
Student AS S
Full JOIN
Registration AS R
ON S.ID = R.StudentID
Full JOIN Course AS C
ON R.CourseID = C.ID;
```

This question is the same as question 1 but with join, we get only the common elements while using full join, we get all the records that may be common between 2 tables or not. So, here I used full join instead.

3. Find courses with no registered students.

```
SELECT C.ID,C.CourseName,C.CourseCode,C.InstructorName
FROM COURSE as C
Left JOIN Registration as R
ON C.ID = R.CourseID
WHERE R.CourseID IS NULL;
```

In this question, I have to find a course that no student had enrolled in, which means we had to use the course and registration table and find the records other than that were common and strictly only on the course table. So, I used the **left join** to get all the records from the course table while we returned those whose courseID from the registration table was null.

4. Insert a new student into the database.

```
Insert into Student values
(10236,"Dummy123","Kathmandu","Male",'2002-10-30');
```

This statement inserts the values serially to the table Student.

5. Add a new course to the system.

```
Insert into Course values
(115,"Engineering Mathematics-3","SH251","Khadga Sing Gupta"),
(110,"Engineering Mathematics-2","SH251","Rabi Singh Pandey");
```

Above query, inserts multiple values to the table courses with serial input values.

6. Register a student for a course.

```
Insert into Registration values (108,10231,110);
```

This question has asked us to register a student for a course, that is a record to be added to the registration table.

7. Remove a student from a course.

```
DELETE FROM Registration
WHERE StudentID = (SELECT ID FROM Student WHERE FullName = 'Amrit Bista')
AND CourseID = (SELECT ID FROM Course WHERE CourseName = 'Engineering
Mathematics-2');
```

In this question,the first part states there is something to be deleted from the table named Registration and with which clause, we identify that the item should have what particular value. Since I tried to delete the course Engineering Mathematics-2 from Amrit Bista, I selected the respective ID and substituted them in the where clause and joined those both conditions since, both the criteria should be matched. Same operation can also be shortened if the ID in the registration table was known.

Submitted By:
Amrit Raj Bista
aamritbistaa@gmail.com