CODEBOOSTERS TECH - GRAPHQL + JSON FORMAT EXAMPLE

Full Step-by-Step Explanation

1. Importing Required Libraries

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

- **express**: A Node.js framework to build web servers quickly.
- `require("express")`: Loads the Express library so you can use it.

```javascript
const { graphqlHTTP } = require("express-graphql");
```

- **express-graphql**: Middleware to connect GraphQL with Express.
- `{ graphqlHTTP }`: Destructuring to get only `graphqlHTTP` function, which connects GraphQL and Express.

```javascript
const { buildSchema } = require("graphql");
```

- **graphql**: The core library for GraphQL schema and types.
- `buildSchema` : A function to **create a schema** using GraphQL's schema language (a special syntax).

```javascript
const fs = require("fs").promises;
```

- **fs** (File System): A built-in Node.js module to read and write files.
- `.promises` : Here you use **the async version** of `fs` methods, so you can use `await` .

```javascript
const path = require("path");
```

- **path**: A Node.js built-in module to handle and transform file paths safely (works on all OS).

---

2. Setup Express app

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

- `app` is now your **Express application**.
- You will attach routes (like `/graphql`) on it.

---

3. Define the Students JSON File Location

```javascript
const studentsFilePath = path.join(__dirname, "students.json");
```

- `__dirname` : The folder where this JS file is running.
- `"students.json"` : The file where student data will be stored.
- `path.join()` : Joins these parts to create a **correct, cross-platform file path**.

---

4. Define Utility Functions

a) Load Students

```javascript
const loadStudents = async () => {
  try {
    const data = await fs.readFile(studentsFilePath, "utf8");
    return JSON.parse(data);
  } catch (err) {
    console.error("Error loading students:", err);
    return [];
  }
};
```

- `loadStudents()` : Reads `students.json` file.
- `fs.readFile()` : Asynchronously reads the file content.
- `JSON.parse()` : Converts the file string into a JS object/array.
- **If error** (e.g., file missing), it catches the error and returns an **empty array** `[]` .

---

b) Save Students

```javascript
```

```
const saveStudents = async (students) => {
  try {
    await fs.writeFile(
      studentsFilePath,
      JSON.stringify(students, null, 2),
      "utf8",
    );
  } catch (err) {
    console.error("Error saving students:", err);
  }
};
```

- `saveStudents(students)` : Writes students array to the file.
- `JSON.stringify()` : Converts students array to **pretty printed JSON** (2 spaces indent).
- `fs.writeFile()` : Asynchronously writes it to disk.
- **If error** (e.g., no permission), it logs the error.

---

5. Define the GraphQL Schema

javascript

```
const schema = buildSchema(`
  type Query {
    getStudents: [Student]
    getStudent(id: ID!): Student
  }

  type Mutation {
    addStudent(name: String!, age: Int!, department: String!, enrollmentYear: Int!):
Student
    updateStudent(id: ID!, name: String, age: Int, department: String,
enrollmentYear: Int): Student
    deleteStudent(id: ID!): String
  }

  type Student {
    id: ID!
```

```
    name: String
    age: Int
    department: String
    enrollmentYear: Int
  }
`);
```

- **Query**:
  - `getStudents`: Returns a list of all students.
  - `getStudent(id: ID!)`: Returns a single student by ID.
- **Mutation**:
  - `addStudent(...)`: Add new student.
  - `updateStudent(...)`: Update an existing student.
  - `deleteStudent(id)`: Delete a student.
- **Student**:
  - Object type with `id`, `name`, `age`, `department`, `enrollmentYear`.

> **Important**: `!` after a type (e.g., `ID!`) means **required**.

---

6. Global Variable

javascript

```
let students = [];
```

- Initially an **empty array**.
- Will be loaded with real data from `students.json`.

---

7. Load Students on Server Startup

javascript

```
loadStudents()
  .then((loadedStudents) => {
    students = loadedStudents;
```

- When server starts, **load students** from file.

---

8. Define Resolvers

Resolvers are **functions that are called** when GraphQL queries/mutations are executed.

Queries:

javascript

```
getStudents: () => {
  return students;
},
```

- Return all students.

---

javascript

```
getStudent: ({ id }) => {
  return students.find((student) => student.id === parseInt(id));
},
```

- `id` is string type in GraphQL.
- `parseInt(id)` : convert to number.
- `find()` finds the student.

Mutations:

```javascript
addStudent: async ({ name, age, department, enrollmentYear }) => {
  const id = students.length
    ? Math.max(...students.map((s) => s.id)) + 1
    : 1;
```

- Generate new unique ID.
- If no students, ID = 1.
- Otherwise, maximum ID + 1.

```javascript
const newStudent = { id, name, age, department, enrollmentYear };
students.push(newStudent);
await saveStudents(students);
return newStudent;
```

- Create new student.
- Save updated array.

```javascript
updateStudent: async ({ id, name, age, department, enrollmentYear }) => {
  const studentId = parseInt(id);
  const student = students.find((s) => s.id === studentId);
  if (!student) return null;
```

- Find the student by ID.
- If not found, return `null`.

javascript

```
student.name = name || student.name;
student.age = age || student.age;
student.department = department || student.department;
student.enrollmentYear = enrollmentYear || student.enrollmentYear;
```

- Update only provided fields, others stay the same.

javascript

```
await saveStudents(students);
return student;
```

- Save after update.

javascript

```
deleteStudent: async ({ id }) => {
  const studentId = parseInt(id);
  console.log("Loaded students:", students);
  console.log("Deleting student with ID:", studentId);

  const index = students.findIndex((student) => student.id === studentId);
  if (index === -1) {
    console.log("Student not found with ID:", studentId);
    return "Student not found.";
  }
```

- Find index of student to delete.
- If not found, return a message.

```javascript
students.splice(index, 1);
await saveStudents(students);
return \`Student with ID ${studentId} deleted successfully.\`;
```

- Remove student from array.
- Save updated array.

---

9. Setup GraphQL Middleware

```javascript
app.use(
  "/graphql",
  graphqlHTTP({
    schema: schema,
    rootValue: root,
    graphiql: true,
  }),
);
```

- `/graphql` : Endpoint for your GraphQL server.
- `graphiql: true` : You get an **interactive UI** in the browser to test queries.

---

10. Start Server

```javascript
app.listen(4000, () => {
  console.log("Server running at http://localhost:4000/graphql");
});
```

- Start the Express server on **port 4000**.

INPUT Types and OUTPUTS

Query 1: Get All Students

Input (GraphQL query):

```graphql
query {
  getStudents {
    id
    name
    age
    department
    enrollmentYear
  }
}
```

Output (Example):

```json
{
  "data": {
    "getStudents": [
      {
        "id": 1,
        "name": "Alice",
        "age": 20,
        "department": "Computer Science",
        "enrollmentYear": 2022
      }
    ]
  }
}
```

Query 2: Get Single Student

Input:

```graphql
query {
  getStudent(id: "1") {
    id
    name
    age
    department
    enrollmentYear
  }
}
```

Output:

```json
{
  "data": {
    "getStudent": {
      "id": 1,
      "name": "Alice",
      "age": 20,
      "department": "Computer Science",
      "enrollmentYear": 2022
    }
  }
}
```

Mutation 1: Add Student

Input:

```graphql
```

```graphql
mutation {
  addStudent(name: "Bob", age: 22, department: "Mechanical", enrollmentYear: 2021) {
    id
    name
  }
}
```

Output:

```json
{
  "data": {
    "addStudent": {
      "id": 2,
      "name": "Bob"
    }
  }
}
```

Mutation 2: Update Student

Input:

```graphql
mutation {
  updateStudent(id: "2", age: 23) {
    id
    name
    age
  }
}
```

Output:

```json
```

```
{
  "data": {
    "updateStudent": {
      "id": 2,
      "name": "Bob",
      "age": 23
    }
  }
}
```

Mutation 3: Delete Student

Input:

graphql

```
mutation {
  deleteStudent(id: "2")
}
```

Output:

json

```
{
  "data": {
    "deleteStudent": "Student with ID 2 deleted successfully."
  }
}
```

📌 Summary

✅ Your server can:

- **Get all** students.

- **Get one** student by ID.
- **Add** a new student.
- **Update** an existing student.
- **Delete** a student.

✅ Data is **saved persistently** into `students.json`.

✅ You get a **nice UI** (GraphiQL) at http://localhost:4000/graphql.

📚 Quick Keywords Glossary with Emojis and Examples

Keyword	Meaning	Real-Life Example
const ⚛️	Declare a **constant** (cannot be reassigned)	Imagine **your phone number —** once it's set, you can't change it. You can use it to identify yourself, but you can't change it.
let 🔄	Declare a **mutable** variable	Think of **a shopping cart**. You add items (variable changes) as you shop. It can change depending on your actions.
require() 📦	Import a Node.js module	It's like **bringing a toolbox** (external module) to your workspace, where you can grab tools (functions) for specific tasks.
async ⏳	Declare an **asynchronous** function	Like **cooking dinner** while chatting with a friend. You don't wait to finish cooking before talking; you handle both at once.

Keyword	Meaning	Real-Life Example
await 🕰️	Wait for a promise to resolve	Imagine **waiting for a bus —** you wait (asynchronously) until it arrives. You're not stuck, you just wait until it's ready.
try/catch 🚨	Handle errors in a block of code	Think of **driving a car**: You try to go down a street, but if there's an obstacle, you **catch** it and find another route.
Promise 🎁	Represents an async value in the future	It's like **ordering a product online**. The store promises to deliver it later. You're not sure when, but it will come.
path.join() 🌐	Safely join parts of a file path	Like **walking through a path**: If you're in a forest (a folder), you take different routes (file names) until you reach your destination (file).
JSON.parse() 🔍	String → Object	It's like **turning a text message** with details into a full **map with directions**. You convert it into usable data.
JSON.stringify() 📝	Object → String	Imagine **drawing a map**. Afterward, you **convert** it into a written description (string) so others can read it easily.
buildSchema 🏗️	Build GraphQL schema from string	Like **drafting blueprints** for a house. The blueprint (schema) defines what the house (your GraphQL API) should look like.
graphiql 💻	UI tool to test GraphQL APIs	Think of it as a **testing lab** for a recipe. You input different ingredients (queries) and see the outcome right away.
destructuring 🧩	Extract parts from an object easily	Like **unpacking a suitcase**: You take out specific items (properties) one by one, instead of digging through the whole bag.