CODEBOOSTERS TECH - USE CASES SOLUTIONS

✅ Use Case 1: Event-Driven Real-Time Chat Message Logger Using Node.js

🧩 Goal:

Implement an **event-driven system** in Node.js where:

- A custom event is triggered every time a user sends a chat message.
- The message is logged to a file on the server.

✅ 1. Complete Code

📄 File: `chatLogger.js`

javascript

```
const EventEmitter = require('events');
const fs = require('fs');
const path = require('path');

// Create an instance of EventEmitter
class ChatEmitter extends EventEmitter {}
const chatEmitter = new ChatEmitter();

// Path to log file
const logFilePath = path.join(__dirname, 'chatLog.txt');

// Event listener for 'message' event
chatEmitter.on('message', (username, message) => {
    const timestamp = new Date().toISOString();
    const logEntry = `${timestamp} - ${username}: ${message}\n`;
```

```
    // Append message to file
    fs.appendFile(logFilePath, logEntry, (err) => {
        if (err) {
            console.error('Failed to write message to log file:', err);
        } else {
            console.log('Message logged:', logEntry.trim());
        }
    });
});

// Simulate messages from users
function sendMessage(username, message) {
    console.log(`${username} says: "${message}"`);
    chatEmitter.emit('message', username, message);
}

// Test Inputs
sendMessage('Alice', 'Hello, everyone!');
sendMessage('Bob', 'Hi Alice!');
```

📘 2. Line-by-Line Explanation

javascript

```
const EventEmitter = require('events');
```

- Imports Node.js's built-in `events` module, which lets you create and handle custom events.

javascript

```
const fs = require('fs');
const path = require('path');
```

- `fs` is used to write logs to a file.
- `path` ensures cross-platform compatibility when dealing with file paths.

javascript

```
class ChatEmitter extends EventEmitter {}
```

- We create a custom class `ChatEmitter` that extends `EventEmitter`. This allows us to define and use custom events like `'message'`.

javascript

```
const chatEmitter = new ChatEmitter();
```

- Instantiates our custom event emitter so we can emit and listen to events.

javascript

```
const logFilePath = path.join(__dirname, 'chatLog.txt');
```

- Defines the full path to the file `chatLog.txt` in the current directory.

javascript

```
chatEmitter.on('message', (username, message) => {
```

- Sets up an event listener that listens for the `'message'` event.
- When triggered, it receives the `username` and `message`.

javascript

```
    const timestamp = new Date().toISOString();
```

- Captures the current timestamp in ISO format for consistent log entries.

javascript

```
    const logEntry = `${timestamp} - ${username}: ${message}\n`;
```

- Formats the log entry with timestamp, username, and message.

javascript

```
    fs.appendFile(logFilePath, logEntry, (err) => {
```

- Appends the log entry to `chatLog.txt`. If the file doesn't exist, it's created.

```javascript
        if (err) {
            console.error('Failed to write message to log file:', err);
        } else {
            console.log('Message logged:', logEntry.trim());
        }
```

- Handles logging success or failure.

```javascript
function sendMessage(username, message) {
```

- Simulates a user sending a message.

```javascript
    console.log(`${username} says: "${message}"`);
    chatEmitter.emit('message', username, message);
```

- Logs the message to console and emits the `'message'` event, triggering the listener.

🗝️ 3. Important Keywords

Keyword	Description
`EventEmitter`	Node.js module to define and handle events.
`emit`	Fires a custom event.
`on`	Registers a listener for an event.

Keyword	Description
`fs.appendFile`	Appends data to a file, creating it if it doesn't exist.
`__dirname`	Built-in variable referring to current directory.
`path.join`	Joins paths safely for cross-platform use.

🧪 4. Test Inputs and Outputs

Test Inputs:

```javascript
sendMessage('Alice', 'Hello, everyone!');
sendMessage('Bob', 'Hi Alice!');
```

Console Output:

```yaml
Alice says: "Hello, everyone!"
Message logged: 2025-04-29T08:32:15.123Z - Alice: Hello, everyone!
Bob says: "Hi Alice!"
Message logged: 2025-04-29T08:32:15.456Z - Bob: Hi Alice!
```

File Output in `chatLog.txt`:

```makefile
2025-04-29T08:32:15.123Z - Alice: Hello, everyone!
2025-04-29T08:32:15.456Z - Bob: Hi Alice!
```

-

✅ Use Case 2: Timer-Based Order Reminder System Using Node.js

🧩 Goal:

Build a system for a **food delivery service** that:

- Sends out a **notification every 5 seconds**.
- Continues to do so until the order is marked as **delivered**.

✅ 1. Complete Code

📄 File: `orderReminder.js`

javascript

```
let orderDelivered = false;
let reminderCount = 0;

// Simulate sending a notification
function sendNotification(orderId) {
    console.log(`Reminder ${++reminderCount}: Order #${orderId} is still being
prepared.`);
}

// Function to mark the order as delivered after a delay
function markAsDelivered(orderId, delayInSeconds) {
    setTimeout(() => {
        orderDelivered = true;
        console.log(`Order #${orderId} has been delivered.`);
    }, delayInSeconds * 1000);
}

// Main reminder system
function startReminder(orderId) {
```

```
    const reminderInterval = setInterval(() => {
        if (orderDelivered) {
            clearInterval(reminderInterval);
            console.log(`Stopping reminders for Order #${orderId}.`);
        } else {
            sendNotification(orderId);
        }
    }, 5000); // every 5 seconds
}

// Test Run
const orderId = 101;
startReminder(orderId);
markAsDelivered(orderId, 16); // simulate delivery after 16 seconds
```

📘 2. Line-by-Line Explanation

javascript

```
let orderDelivered = false;
let reminderCount = 0;
```

- `orderDelivered`: a flag to track whether the order has been delivered.
- `reminderCount`: counts how many reminders have been sent.

javascript

```
function sendNotification(orderId) {
    console.log(`Reminder ${++reminderCount}: Order #${orderId} is still being
prepared.`);
}
```

- This simulates sending a notification.
- Uses `++reminderCount` to increase the reminder count before displaying it.

javascript

```
function markAsDelivered(orderId, delayInSeconds) {
    setTimeout(() => {
        orderDelivered = true;
        console.log(`Order #${orderId} has been delivered.`);
    }, delayInSeconds * 1000);
}
```

- Simulates the order being delivered after a certain number of seconds.
- Uses `setTimeout` to delay the delivery simulation.

javascript

```
function startReminder(orderId) {
```

- Starts the reminder system for a specific order.

javascript

```
    const reminderInterval = setInterval(() => {
```

- Uses `setInterval` to send a notification every 5 seconds.

javascript

```
        if (orderDelivered) {
            clearInterval(reminderInterval);
            console.log(`Stopping reminders for Order #${orderId}.`);
        } else {
            sendNotification(orderId);
        }
    }, 5000);
}
```

- Checks if the order is delivered:
  - If **yes**, it stops sending reminders by calling `clearInterval`.
  - If **no**, it continues to call `sendNotification`.

javascript

```
const orderId = 101;
startReminder(orderId);
markAsDelivered(orderId, 16); // simulate delivery after 16 seconds
```

- Test input:
  - Starts reminders for order `101`.
  - After 16 seconds, marks it as delivered.

🗝️ 3. Important Keywords Used

Keyword	Description
`setInterval()`	Calls a function repeatedly at fixed intervals (5 seconds here).
`setTimeout()`	Calls a function once after a delay (used for simulating delivery).
`clearInterval()`	Stops the repeated function calls from `setInterval`.
`++reminderCount`	Increments the count before using it in the log.

🧪 4. Test Inputs and Outputs

Test Input:

javascript

```
startReminder(101);
markAsDelivered(101, 16); // delivery after 16 seconds
```

Console Output:

yaml

```
Reminder 1: Order #101 is still being prepared.
Reminder 2: Order #101 is still being prepared.
```

```
Reminder 3: Order #101 is still being prepared.
Order #101 has been delivered.
Stopping reminders for Order #101.
```

- The reminder runs every 5 seconds.
- The delivery occurs after ~16 seconds (around the 4th reminder mark).
- The system stops sending reminders once delivered.

✅ Use Case 3: Simple HTTP API Returning Employee Details (Node.js)

🧩 Goal:

Create a lightweight HTTP server that:

- Listens for GET requests.
- Returns employee details in **JSON** format.
- No external libraries used (pure Node.js).

✅ 1. Complete Code

📄 File: `employeeServer.js`

javascript

```javascript
const http = require('http');
const url = require('url');
```

```
// Sample employee data
const employees = [
    { id: 1, name: 'Alice', department: 'Engineering' },
    { id: 2, name: 'Bob', department: 'HR' },
    { id: 3, name: 'Charlie', department: 'Marketing' }
];

// Create HTTP server
const server = http.createServer((req, res) => {
    const parsedUrl = url.parse(req.url, true);

    // Set response headers
    res.setHeader('Content-Type', 'application/json');

    if (parsedUrl.pathname === '/employees' && req.method === 'GET') {
        // Return all employees
        res.writeHead(200);
        res.end(JSON.stringify(employees));
    } else {
        // Not Found
        res.writeHead(404);
        res.end(JSON.stringify({ error: 'Route not found' }));
    }
});

// Server listens on port 3000
server.listen(3000, () => {
    console.log('Server running at http://localhost:3000');
});
```

📘 2. Line-by-Line Explanation

javascript

```
const http = require('http');
const url = require('url');
```

- Loads Node.js built-in `http` module to create a web server.
- `url` is used to parse the request path and query parameters.

```javascript
const employees = [
    { id: 1, name: 'Alice', department: 'Engineering' },
    { id: 2, name: 'Bob', department: 'HR' },
    { id: 3, name: 'Charlie', department: 'Marketing' }
];
```

- Hardcoded sample employee data stored as an array of objects.

```javascript
const server = http.createServer((req, res) => {
```

- Creates the HTTP server. Every request triggers the callback.

```javascript
    const parsedUrl = url.parse(req.url, true);
```

- Parses the incoming URL into parts: pathname, query, etc.

```javascript
    res.setHeader('Content-Type', 'application/json');
```

- Sets the response content type to JSON.

```javascript
    if (parsedUrl.pathname === '/employees' && req.method === 'GET') {
```

- Checks if the request is a GET to `/employees`.

```javascript
```

```
        res.writeHead(200);
        res.end(JSON.stringify(employees));
```

- Sends a 200 OK status and the list of employees as JSON.

```javascript
    } else {
        res.writeHead(404);
        res.end(JSON.stringify({ error: 'Route not found' }));
    }
});
```

- Handles all other requests by returning a 404 with an error message.

```javascript
server.listen(3000, () => {
    console.log('Server running at http://localhost:3000');
});
```

- Starts the server on port 3000.

🗝️ 3. Important Keywords Used

Keyword	Description
`http.createServer()`	Creates a Node.js HTTP server.
`res.setHeader()`	Sets response headers like content type.
`res.writeHead()`	Sets HTTP status code.
`res.end()`	Ends the response and sends data back to the client.
`url.parse()`	Parses the request URL into pathname and query.

Keyword	Description
`JSON.stringify()`	Converts JavaScript objects to JSON strings.

🧪 4. Test Inputs and Outputs

Test Input:

- Start server with:

```bash
node employeeServer.js
```

- Then access in browser or API tool:

```bash
http://localhost:3000/employees
```

Output (HTTP 200):

```json
[
    { "id": 1, "name": "Alice", "department": "Engineering" },
    { "id": 2, "name": "Bob", "department": "HR" },
    { "id": 3, "name": "Charlie", "department": "Marketing" }
]
```

Invalid Route Example:

```bash
http://localhost:3000/foo
```

Output (HTTP 404):

```json
{ "error": "Route not found" }
```

✅ Use Case 4: GraphQL API for Querying and Mutating Employee Data

🧩 Goal:

Create a **GraphQL API** that allows:

- Querying all employees or a single employee by `id`.
- Adding a new employee using a **mutation**.

✅ 1. Complete Code

📁 Folder Structure:

```pgsql
graphql-employee-api/
├── index.js
├── schema.js
├── data.js
├── package.json
```

📦 Step 1: Install Dependencies

Run the following command to set up:

```bash
npm init -y
npm install express express-graphql graphql
```

📄 File: `data.js` — Sample In-Memory Data

```javascript
const employees = [
  { id: '1', name: 'Alice', department: 'Engineering' },
  { id: '2', name: 'Bob', department: 'HR' }
];

module.exports = { employees };
```

📄 File: `schema.js` — GraphQL Schema

```javascript
const { GraphQLObjectType, GraphQLString, GraphQLSchema, GraphQLList, GraphQLID,
GraphQLNonNull } = require('graphql');
const { employees } = require('./data');

// Employee Type
const EmployeeType = new GraphQLObjectType({
  name: 'Employee',
  fields: () => ({
    id: { type: GraphQLID },
    name: { type: GraphQLString },
    department: { type: GraphQLString }
  })
});

// Root Query
```

```
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    employees: {
      type: new GraphQLList(EmployeeType),
      resolve() {
        return employees;
      }
    },
    employee: {
      type: EmployeeType,
      args: { id: { type: GraphQLID } },
      resolve(_, args) {
        return employees.find(emp => emp.id === args.id);
      }
    }
  }
});

// Mutation
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addEmployee: {
      type: EmployeeType,
      args: {
        id: { type: new GraphQLNonNull(GraphQLID) },
        name: { type: new GraphQLNonNull(GraphQLString) },
        department: { type: new GraphQLNonNull(GraphQLString) }
      },
      resolve(_, args) {
        const newEmp = { id: args.id, name: args.name, department: args.department
};
        employees.push(newEmp);
        return newEmp;
      }
    }
  }
});

module.exports = new GraphQLSchema({
  query: RootQuery,
```

```
  mutation: Mutation
});
```

📄 File: `index.js` — Express Server

```javascript
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const schema = require('./schema');

const app = express();

app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true // enable GraphiQL UI
}));

app.listen(4000, () => {
  console.log('GraphQL server running at http://localhost:4000/graphql');
});
```

📘 2. Line-by-Line Explanation

`data.js`

- Contains hardcoded employee data stored in-memory.

`schema.js`

```javascript
const EmployeeType = new GraphQLObjectType({
```

- Defines a GraphQL type called `Employee` with fields `id`, `name`, and `department`.

```javascript
const RootQuery = new GraphQLObjectType({
```

- Adds two queries:
  - `employees`: returns all employees.
  - `employee(id: ID)`: fetches one by ID.

```javascript
const Mutation = new GraphQLObjectType({
```

- Adds `addEmployee` mutation that takes `id`, `name`, and `department` and pushes it into the list.

```javascript
module.exports = new GraphQLSchema({ query: RootQuery, mutation: Mutation });
```

- Exports the schema with both query and mutation functionality.

`index.js`

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

- Middleware to handle GraphQL requests using Express.

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

- Registers the `/graphql` endpoint and enables the built-in IDE GraphiQL.

🗝️ 3. Important Keywords

Keyword	Description
`GraphQLObjectType`	Defines types like `Employee`, `Query`, `Mutation`.
`GraphQLList`	Represents an array of items in GraphQL.
`GraphQLNonNull`	Marks a field as required.
`resolve()`	Resolver function to fetch the actual data.
`express-graphql`	Middleware that connects Express with GraphQL.

🧪 4. Test Inputs and Outputs

Test via browser at:

```bash
http://localhost:4000/graphql
```

📤 Query All Employees

```graphql
query {
  employees {
    id
    name
    department
  }
}
```

✅ Output:

```json
```

```
{
  "data": {
    "employees": [
      { "id": "1", "name": "Alice", "department": "Engineering" },
      { "id": "2", "name": "Bob", "department": "HR" }
    ]
  }
}
```

📤 Query One Employee by ID

graphql

```
query {
  employee(id: "2") {
    name
    department
  }
}
```

✅ Output:

json

```
{
  "data": {
    "employee": {
      "name": "Bob",
      "department": "HR"
    }
  }
}
```

📤 Add a New Employee

graphql

```
mutation {
  addEmployee(id: "3", name: "Charlie", department: "Sales") {
    id
    name
  }
}
```

✅ **Output:**

```
json

{
  "data": {
    "addEmployee": {
      "id": "3",
      "name": "Charlie"
    }
  }
}
```

✅ Use Case 5: GraphQL Schema for Product Management (Query + Mutation)

🧩 Goal:

Design a GraphQL API for an e-commerce platform to:

- **Query** all products or a product by its ID.
- **Add** new products using a mutation.

✅ 1. Complete Code

We'll keep the structure similar to the previous use case.

📁 Folder Structure:

```pgsql
graphql-product-api/
├── index.js
├── schema.js
├── data.js
├── package.json
```

📦 Step 1: Install Dependencies

```bash
npm init -y
npm install express express-graphql graphql
```

📄 File: `data.js`

```javascript
const products = [
  { id: 'p1', name: 'Laptop', price: 999.99 },
  { id: 'p2', name: 'Smartphone', price: 699.49 }
];

module.exports = { products };
```

File: `schema.js`

javascript

```
const {
  GraphQLObjectType,
  GraphQLString,
  GraphQLSchema,
  GraphQLFloat,
  GraphQLList,
  GraphQLID,
  GraphQLNonNull
} = require('graphql');

const { products } = require('./data');

// Product Type
const ProductType = new GraphQLObjectType({
  name: 'Product',
  fields: () => ({
    id: { type: GraphQLID },
    name: { type: GraphQLString },
    price: { type: GraphQLFloat }
  })
});

// Root Query
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    products: {
      type: new GraphQLList(ProductType),
      resolve() {
        return products;
      }
    },
    product: {
      type: ProductType,
      args: { id: { type: GraphQLID } },
      resolve(_, args) {
        return products.find(p => p.id === args.id);
      }
```

```
    }
  }
});

// Mutation
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addProduct: {
      type: ProductType,
      args: {
        id: { type: new GraphQLNonNull(GraphQLID) },
        name: { type: new GraphQLNonNull(GraphQLString) },
        price: { type: new GraphQLNonNull(GraphQLFloat) }
      },
      resolve(_, args) {
        const newProduct = { id: args.id, name: args.name, price: args.price };
        products.push(newProduct);
        return newProduct;
      }
    }
  }
});

module.exports = new GraphQLSchema({
  query: RootQuery,
  mutation: Mutation
});
```

📄 File: `index.js`

javascript

```
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const schema = require('./schema');

const app = express();
```

```
app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));

app.listen(4001, () => {
  console.log('GraphQL server running at http://localhost:4001/graphql');
});
```

📘 2. Line-by-Line Explanation

`ProductType`

javascript

```
const ProductType = new GraphQLObjectType({
```

Defines the `Product` GraphQL type with `id`, `name`, and `price`.

`RootQuery`

javascript

```
fields: {
  products: {
    type: new GraphQLList(ProductType),
    resolve() {
      return products;
    }
  },
```

- `products`: returns all product items.
- `product(id: ID)`: returns one product by ID.

`Mutation`

javascript

```
addProduct: {
  type: ProductType,
  args: {
    id: { type: new GraphQLNonNull(GraphQLID) },
    name: { type: new GraphQLNonNull(GraphQLString) },
    price: { type: new GraphQLNonNull(GraphQLFloat) }
  },
```

- Defines the mutation `addProduct`, which accepts `id`, `name`, and `price` to add a product to the list.

🗝️ 3. Important GraphQL Keywords

Keyword	Description
`GraphQLFloat`	Used for decimal values (product price).
`GraphQLList`	Represents an array of types.
`GraphQLNonNull`	Makes a field required.
`resolve()`	Resolver function to fetch data.

🧪 4. Test Inputs and Outputs

Start server:

```bash
node index.js
```

Navigate to http://localhost:4001/graphql

📤 Query All Products

```graphql
query {
  products {
    id
    name
    price
  }
}
```

✅ Output:

```json
{
  "data": {
    "products": [
      { "id": "p1", "name": "Laptop", "price": 999.99 },
      { "id": "p2", "name": "Smartphone", "price": 699.49 }
    ]
  }
}
```

📤 Add New Product

```graphql
mutation {
  addProduct(id: "p3", name: "Headphones", price: 199.95) {
    id
    name
    price
  }
}
```

✅ Output:

```json
```

```
{
  "data": {
    "addProduct": {
      "id": "p3",
      "name": "Headphones",
      "price": 199.95
    }
  }
}
```

📤 Get Product by ID

graphql

```
query {
  product(id: "p3") {
    name
    price
  }
}
```

✅ Output:

json

```
{
  "data": {
    "product": {
      "name": "Headphones",
      "price": 199.95
    }
  }
}
```

✅ Use Case 6: Validate Account Number (Exactly 10 Digits) in GraphQL

🧩 Goal:

- Create a GraphQL API that:
  - Accepts an **account number**.
  - **Validates** it must be exactly **10 digits**.
  - Throws an **error** if invalid.

✅ 1. Complete Code

📁 Folder Structure:

```pgsql
graphql-account-validation/
├── index.js
├── schema.js
├── package.json
```

📦 Step 1: Install Dependencies

```bash
npm init -y
```

```
npm install express express-graphql graphql
```

📄 File: `schema.js`

javascript

```javascript
const {
  GraphQLObjectType,
  GraphQLString,
  GraphQLSchema,
  GraphQLNonNull
} = require('graphql');

// Account Type
const AccountType = new GraphQLObjectType({
  name: 'Account',
  fields: () => ({
    accountNumber: { type: GraphQLString },
    message: { type: GraphQLString }
  })
});

// Root Mutation
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    validateAccount: {
      type: AccountType,
      args: {
        accountNumber: { type: new GraphQLNonNull(GraphQLString) }
      },
      resolve(_, args) {
        const { accountNumber } = args;

        // Validate: must be exactly 10 digits
        const isValid = /^\d{10}$/.test(accountNumber);

        if (!isValid) {
          throw new Error('Account number must be exactly 10 digits.');
```

```
        }

        return {
          accountNumber,
          message: 'Account number is valid.'
        };
      }
    }
  }
});

module.exports = new GraphQLSchema({
  mutation: Mutation
});
```

📄 File: `index.js`

javascript

```
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const schema = require('./schema');

const app = express();

app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));

app.listen(4002, () => {
  console.log('GraphQL server running at http://localhost:4002/graphql');
});
```

📘 2. Line-by-Line Explanation

`AccountType`

```javascript
const AccountType = new GraphQLObjectType({
  name: 'Account',
  fields: () => ({
    accountNumber: { type: GraphQLString },
    message: { type: GraphQLString }
  })
});
```

- Defines a simple type `Account` with fields: `accountNumber` and `message`.

`validateAccount` Mutation

```javascript
validateAccount: {
  type: AccountType,
  args: {
    accountNumber: { type: new GraphQLNonNull(GraphQLString) }
  },
```

- Takes one argument: `accountNumber` (required).
- Uses a **regular expression** to validate:

  ```javascript
  /^\d{10}$/
  ```

  - `^` start of string
  - `\d{10}` exactly 10 digits
  - `$` end of string
- If invalid, throws:

  ```javascript
  ```

```
throw new Error('Account number must be exactly 10 digits.');
```

- If valid, returns the account number and a success message.

`index.js`

- Sets up the Express server and the GraphQL endpoint at `/graphql`.

🗝️ 3. Important GraphQL + JS Keywords

Keyword	Description
`GraphQLNonNull`	Makes a field **mandatory**.
`throw new Error()`	Raises a GraphQL execution error.
`RegExp` (`/^\d{10}$/`)	Validates account number format.

🧪 4. Test Inputs and Outputs

Start server:

```bash
node index.js
```

Visit: http://localhost:4002/graphql

📤 Test 1: Valid Account Number

```graphql
```

```
mutation {
  validateAccount(accountNumber: "1234567890") {
    accountNumber
    message
  }
}
```

✅ Output:

```
json

{
  "data": {
    "validateAccount": {
      "accountNumber": "1234567890",
      "message": "Account number is valid."
    }
  }
}
```

📤 Test 2: Invalid Account Number (Too Short)

```
graphql

mutation {
  validateAccount(accountNumber: "12345") {
    accountNumber
    message
  }
}
```

❌ Output (Error):

```
json

{
  "errors": [
    {
      "message": "Account number must be exactly 10 digits.",
```

```
      "locations": [...],
      "path": ["validateAccount"]
    }
  ],
  "data": {
    "validateAccount": null
  }
}
```

📤 Test 3: Invalid Account Number (Contains Letters)

graphql

```
mutation {
  validateAccount(accountNumber: "12345abcd9") {
    accountNumber
    message
  }
}
```

❌ Output (Error):

json

```
{
  "errors": [
    {
      "message": "Account number must be exactly 10 digits.",
      "locations": [...],
      "path": ["validateAccount"]
    }
  ],
  "data": {
    "validateAccount": null
  }
}
```

✅ Use Case 7: GraphQL User Authentication (MongoDB)

🧩 Goal:

Build a secure authentication system with:

- **User Registration**
- **User Login**
- Passwords **hashed** using `bcrypt`
- JWT token generation on login

✅ 1. Project Setup

📁 Folder Structure:

```pgsql
graphql-auth-mongodb/
├── index.js
├── schema.js
├── models/
│   └── User.js
├── db.js
├── .env
├── package.json
```

📦 Step 1: Install Dependencies

```bash
npm init -y
npm install express express-graphql graphql mongoose bcryptjs jsonwebtoken dotenv
```

📄 File: `.env`

```env
MONGO_URI=mongodb://localhost:27017/authdb
JWT_SECRET=mysecretkey
```

📄 File: `db.js`

```javascript
const mongoose = require('mongoose');
require('dotenv').config();

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI);
    console.log('MongoDB connected');
  } catch (err) {
    console.error(err.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

📄 File: `models/User.js`

javascript

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

const UserSchema = new mongoose.Schema({
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

module.exports = mongoose.model('User', UserSchema);
```

📄 File: `schema.js`

javascript

```
const {
  GraphQLObjectType,
  GraphQLString,
  GraphQLSchema,
  GraphQLNonNull
} = require('graphql');

const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('./models/User');
require('dotenv').config();

// User Type
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: {
    id: { type: GraphQLString },
    email: { type: GraphQLString },
    token: { type: GraphQLString }
  }
});
```

```
// Root Mutation
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    register: {
      type: UserType,
      args: {
        email: { type: new GraphQLNonNull(GraphQLString) },
        password: { type: new GraphQLNonNull(GraphQLString) }
      },
      async resolve(_, args) {
        const { email, password } = args;
        const existingUser = await User.findOne({ email });
        if (existingUser) throw new Error('User already exists');

        const hashedPassword = await bcrypt.hash(password, 10);
        const user = new User({ email, password: hashedPassword });
        await user.save();

        return { id: user.id, email: user.email };
      }
    },
    login: {
      type: UserType,
      args: {
        email: { type: new GraphQLNonNull(GraphQLString) },
        password: { type: new GraphQLNonNull(GraphQLString) }
      },
      async resolve(_, args) {
        const { email, password } = args;
        const user = await User.findOne({ email });
        if (!user) throw new Error('User not found');

        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) throw new Error('Invalid credentials');

        const token = jwt.sign({ id: user.id }, process.env.JWT_SECRET, {
          expiresIn: '1h'
        });

        return { id: user.id, email: user.email, token };
      }
```

```
    }
  }
});

module.exports = new GraphQLSchema({
  mutation: Mutation
});
```

📄 File: `index.js`

javascript

```
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const schema = require('./schema');
const connectDB = require('./db');
require('dotenv').config();

const app = express();
connectDB();

app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));

app.listen(4003, () => {
  console.log('GraphQL server running at http://localhost:4003/graphql');
});
```

📘 2. Line-by-Line Explanation

🔐 User Registration

javascript

```
const existingUser = await User.findOne({ email });
```

Check if the email already exists.

```javascript
const hashedPassword = await bcrypt.hash(password, 10);
```

Hash the password securely.

```javascript
const user = new User({ email, password: hashedPassword });
await user.save();
```

Save the new user to MongoDB.

🔓 User Login

```javascript
const isMatch = await bcrypt.compare(password, user.password);
```

Check if provided password matches the hash.

```javascript
const token = jwt.sign({ id: user.id }, process.env.JWT_SECRET);
```

Create a JWT for session management.

🗝️ 3. Keywords Used

Keyword	Description
`bcrypt.hash()`	Hash user passwords

Keyword	Description
`bcrypt.compare()`	Compare plaintext and hash
`jsonwebtoken`	Generate tokens for authentication
`GraphQLNonNull`	Required field
`mongoose`	MongoDB ORM for Node.js

🧪 4. Test Inputs and Outputs

Start MongoDB:

```bash
mongod
```

Run the server:

```bash
node index.js
```

Open: http://localhost:4003/graphql

✅ Register a New User

```graphql
mutation {
  register(email: "user@example.com", password: "secret123") {
    id
    email
  }
}
```

Output:

```json
{
  "data": {
    "register": {
      "id": "someId",
      "email": "user@example.com"
    }
  }
}
```

✅ Login User

```graphql
mutation {
  login(email: "user@example.com", password: "secret123") {
    id
    email
    token
  }
}
```

Output:

```json
{
  "data": {
    "login": {
      "id": "someId",
      "email": "user@example.com",
      "token": "eyJhbGciOi..."
    }
  }
}
```

✅ Use Case 8: Task Management with GraphQL, Express & MongoDB

🧩 Goal:

Build a task system where users can:

- **Add tasks**
- **Update task status**
- **View all/pending tasks**

✅ 1. Project Setup

📁 Folder Structure:

```
pgsql

graphql-task-manager/
├── index.js
├── schema.js
├── db.js
├── models/
│   └── Task.js
├── .env
├── package.json
```

📦 Step 1: Install Dependencies

```bash
npm init -y
npm install express express-graphql graphql mongoose dotenv
```

📄 File: `.env`

```env
MONGO_URI=mongodb://localhost:27017/taskdb
```

📄 File: `db.js`

```javascript
const mongoose = require('mongoose');
require('dotenv').config();

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI);
    console.log('MongoDB connected');
  } catch (err) {
    console.error(err.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

📄 File: `models/Task.js`

javascript

```javascript
const mongoose = require('mongoose');

const TaskSchema = new mongoose.Schema({
  title: { type: String, required: true },
  status: { type: String, enum: ['Pending', 'Completed'], default: 'Pending' }
});

module.exports = mongoose.model('Task', TaskSchema);
```

📄 File: `schema.js`

javascript

```javascript
const {
  GraphQLObjectType,
  GraphQLString,
  GraphQLSchema,
  GraphQLNonNull,
  GraphQLList,
  GraphQLID,
  GraphQLEnumType
} = require('graphql');

const Task = require('./models/Task');

// Task Type
const TaskType = new GraphQLObjectType({
  name: 'Task',
  fields: {
    id: { type: GraphQLID },
    title: { type: GraphQLString },
    status: { type: GraphQLString }
  }
});

// Root Query
const RootQuery = new GraphQLObjectType({
  name: 'Query',
```

```
  fields: {
    tasks: {
      type: new GraphQLList(TaskType),
      resolve() {
        return Task.find();
      }
    },
    pendingTasks: {
      type: new GraphQLList(TaskType),
      resolve() {
        return Task.find({ status: 'Pending' });
      }
    }
  }
});

// Root Mutation
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addTask: {
      type: TaskType,
      args: {
        title: { type: new GraphQLNonNull(GraphQLString) }
      },
      resolve(_, args) {
        const task = new Task({ title: args.title });
        return task.save();
      }
    },
    updateTaskStatus: {
      type: TaskType,
      args: {
        id: { type: new GraphQLNonNull(GraphQLID) },
        status: {
          type: new GraphQLEnumType({
            name: 'StatusEnum',
            values: {
              Pending: { value: 'Pending' },
              Completed: { value: 'Completed' }
            }
          })
```

```
        }
      },
      resolve(_, args) {
        return Task.findByIdAndUpdate(
          args.id,
          { status: args.status },
          { new: true }
        );
      }
    }
  }
});

module.exports = new GraphQLSchema({
  query: RootQuery,
  mutation: Mutation
});
```

📄 File: `index.js`

javascript

```
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const connectDB = require('./db');
const schema = require('./schema');
require('dotenv').config();

const app = express();
connectDB();

app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));

app.listen(4004, () => {
```

```
  console.log('Task GraphQL API running at http://localhost:4004/graphql');
});
```

📘 2. Line-by-Line Explanation

Task Model

```javascript
status: { type: String, enum: ['Pending', 'Completed'], default: 'Pending' }
```

- Task has only two valid states.
- Default is `Pending`.

GraphQL Types

- `TaskType`: GraphQL object for tasks (id, title, status).
- `GraphQLEnumType`: Enum for valid statuses in `updateTaskStatus`.

Query Resolvers

- `tasks`: Returns all tasks.
- `pendingTasks`: Returns only those with status `"Pending"`.

Mutation Resolvers

- `addTask`: Creates a task with default `"Pending"` status.
- `updateTaskStatus`: Updates status of a task by `ID`.

🗝️ 3. Important Keywords

Keyword	Description
`GraphQLList`	Returns an array of GraphQL types
`GraphQLEnumType`	Restricts values (e.g., Pending/Completed)
`GraphQLNonNull`	Makes field required
`findByIdAndUpdate`	Mongoose method to update by `_id`

🧪 4. Test Inputs & Outputs

Start MongoDB:

```bash
mongod
```

Run server:

```bash
node index.js
```

Open GraphiQL: http://localhost:4004/graphql

✅ Add Task

```graphql
mutation {
  addTask(title: "Write GraphQL Code") {
    id
    title
    status
```

```
  }
}
```

Output:

```json
{
  "data": {
    "addTask": {
      "id": "abc123",
      "title": "Write GraphQL Code",
      "status": "Pending"
    }
  }
}
```

🔄 Update Task Status

```graphql
mutation {
  updateTaskStatus(id: "abc123", status: Completed) {
    id
    title
    status
  }
}
```

Output:

```json
{
  "data": {
    "updateTaskStatus": {
      "id": "abc123",
      "title": "Write GraphQL Code",
      "status": "Completed"
    }
```

```
  }
}
```

📥 Get All Tasks

graphql

```graphql
query {
  tasks {
    id
    title
    status
  }
}
```

📥 Get Only Pending Tasks

graphql

```graphql
query {
  pendingTasks {
    id
    title
    status
  }
}
```


Codeboosters Tech


team_codeboosters


www.codeboosters.in

✅ Use Case 9: Order Management API with GraphQL, Express & MongoDB

🧩 Goal:

Implement an API where users can:

- **Place new orders**
- **Update order status**
- **View order status**

✅ 1. Project Setup

📁 Folder Structure:

```pgsql
graphql-order-system/
├── index.js
├── schema.js
├── db.js
├── models/
│   └── Order.js
├── .env
├── package.json
```

📦 Step 1: Install Dependencies

```bash
npm init -y
npm install express express-graphql graphql mongoose dotenv
```

File: `.env`

```env
MONGO_URI=mongodb://localhost:27017/orderdb
```

File: `db.js`

```javascript
const mongoose = require('mongoose');
require('dotenv').config();

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI);
    console.log('MongoDB connected');
  } catch (err) {
    console.error(err.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

File: `models/Order.js`

```javascript
const mongoose = require('mongoose');

const OrderSchema = new mongoose.Schema({
  product: { type: String, required: true },
  quantity: { type: Number, required: true },
  status: {
    type: String,
```

```
    enum: ['Placed', 'Processing', 'Shipped', 'Delivered'],
    default: 'Placed'
  }
});

module.exports = mongoose.model('Order', OrderSchema);
```

📄 File: `schema.js`

javascript

```
const {
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLSchema,
  GraphQLNonNull,
  GraphQLList,
  GraphQLID,
  GraphQLEnumType
} = require('graphql');

const Order = require('./models/Order');

// Order Type
const OrderType = new GraphQLObjectType({
  name: 'Order',
  fields: {
    id: { type: GraphQLID },
    product: { type: GraphQLString },
    quantity: { type: GraphQLInt },
    status: { type: GraphQLString }
  }
});

// Root Query
const RootQuery = new GraphQLObjectType({
  name: 'Query',
  fields: {
```

```
    orders: {
      type: new GraphQLList(OrderType),
      resolve() {
        return Order.find();
      }
    },
    order: {
      type: OrderType,
      args: { id: { type: GraphQLID } },
      resolve(_, args) {
        return Order.findById(args.id);
      }
    }
  }
});

// Mutations
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    placeOrder: {
      type: OrderType,
      args: {
        product: { type: new GraphQLNonNull(GraphQLString) },
        quantity: { type: new GraphQLNonNull(GraphQLInt) }
      },
      resolve(_, args) {
        const order = new Order({
          product: args.product,
          quantity: args.quantity
        });
        return order.save();
      }
    },
    updateOrderStatus: {
      type: OrderType,
      args: {
        id: { type: new GraphQLNonNull(GraphQLID) },
        status: {
          type: new GraphQLEnumType({
            name: 'OrderStatusEnum',
            values: {
```

```
              Placed: { value: 'Placed' },
              Processing: { value: 'Processing' },
              Shipped: { value: 'Shipped' },
              Delivered: { value: 'Delivered' }
            }
          })
        }
      },
      resolve(_, args) {
        return Order.findByIdAndUpdate(
          args.id,
          { status: args.status },
          { new: true }
        );
      }
    }
  }
});

module.exports = new GraphQLSchema({
  query: RootQuery,
  mutation: Mutation
});
```

📄 File: `index.js`

javascript

```
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const schema = require('./schema');
const connectDB = require('./db');
require('dotenv').config();

const app = express();
connectDB();

app.use('/graphql', graphqlHTTP({
  schema,
```

```
  graphiql: true
}));

app.listen(4005, () => {
  console.log('Order API running at http://localhost:4005/graphql');
});
```

📘 2. Line-by-Line Explanation

Order Schema

```javascript
status: {
  type: String,
  enum: ['Placed', 'Processing', 'Shipped', 'Delivered'],
  default: 'Placed'
}
```

- Order status is restricted to valid values.
- Defaults to `Placed` on creation.

GraphQL Mutations

- `placeOrder`: creates a new order
- `updateOrderStatus`: updates an order by `id` using a strict enum type

GraphQL Queries

- `orders`: list all orders
- `order(id)`: fetch an order by ID

🗝️ 3. Important Keywords

Keyword	Description
`GraphQLID`	For Mongo `_id`
`GraphQLEnumType`	Enforces enum validation
`findByIdAndUpdate`	MongoDB update
`GraphQLList`	Returns multiple results

🧪 4. Test Inputs & Outputs

Start MongoDB:

```bash
mongod
```

Run the server:

```bash
node index.js
```

Go to: http://localhost:4005/graphql

✅ Place New Order

```graphql
mutation {
  placeOrder(product: "Laptop", quantity: 1) {
    id
    product
    quantity
    status
```

```
  }
}
```

Output:

```json
{
  "data": {
    "placeOrder": {
      "id": "abc123",
      "product": "Laptop",
      "quantity": 1,
      "status": "Placed"
    }
  }
}
```

🔄 Update Order Status

```graphql
mutation {
  updateOrderStatus(id: "abc123", status: Shipped) {
    id
    status
  }
}
```

Output:

```json
{
  "data": {
    "updateOrderStatus": {
      "id": "abc123",
      "status": "Shipped"
    }
```

```
  }
}
```

📥 Get All Orders

graphql

```graphql
query {
  orders {
    id
    product
    quantity
    status
  }
}
```

📥 Get Order by ID

graphql

```graphql
query {
  order(id: "abc123") {
    product
    quantity
    status
  }
}
```

✅ Use Case 10: Real-time Chat App using GraphQL Subscriptions

🧩 Goal:

Implement a **real-time chat system** using:

- **GraphQL Subscriptions** to handle real-time updates.
- **Express** and **Apollo Server** for server setup.
- **WebSockets** to enable real-time communication between users.

✅ 1. Project Setup

📁 Folder Structure:

```pgsql
graphql-chat-app/
├── index.js
├── schema.js
├── db.js
├── models/
│   └── Message.js
├── .env
├── package.json
```

📦 Step 1: Install Dependencies

```bash
npm init -y
npm install express apollo-server-express graphql mongoose dotenv
```

File: `.env`

```env
MONGO_URI=mongodb://localhost:27017/chatdb
```

File: `db.js`

```javascript
const mongoose = require('mongoose');
require('dotenv').config();

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI);
    console.log('MongoDB connected');
  } catch (err) {
    console.error(err.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

File: `models/Message.js`

```javascript
const mongoose = require('mongoose');

const MessageSchema = new mongoose.Schema({
  content: { type: String, required: true },
  sender: { type: String, required: true },
  timestamp: { type: Date, default: Date.now }
});
```

```
module.exports = mongoose.model('Message', MessageSchema);
```

📄 File: `schema.js`

javascript

```
const { gql } = require('apollo-server-express');

// GraphQL Type Definitions
const typeDefs = gql`
  type Message {
    id: ID!
    content: String!
    sender: String!
    timestamp: String!
  }

  type Query {
    messages: [Message]
  }

  type Mutation {
    sendMessage(content: String!, sender: String!): Message
  }

  type Subscription {
    messageSent: Message
  }
`;

module.exports = typeDefs;
```

📄 File: `resolvers.js`

javascript

```
const Message = require('./models/Message');

// GraphQL Resolvers
const resolvers = {
  Query: {
    messages: async () => {
      return await Message.find();
    },
  },

  Mutation: {
    sendMessage: async (_, { content, sender }) => {
      const message = new Message({ content, sender });
      await message.save();
      return message;
    },
  },

  Subscription: {
    messageSent: {
      subscribe: (_, __, { pubsub }) => {
        return pubsub.asyncIterator("MESSAGE_SENT");
      },
    },
  },
};

module.exports = resolvers;
```

📄 File: `index.js`

javascript

```
const express = require('express');
const { ApolloServer, PubSub } = require('apollo-server-express');
const mongoose = require('mongoose');
const http = require('http');
const connectDB = require('./db');
const typeDefs = require('./schema');
```

```
const resolvers = require('./resolvers');
require('dotenv').config();

const pubsub = new PubSub();
const app = express();
const server = http.createServer(app);

// Apollo Server setup with GraphQL Subscriptions
const apolloServer = new ApolloServer({
  typeDefs,
  resolvers,
  context: ({ req, res }) => ({
    pubsub
  })
});

connectDB();

apolloServer.applyMiddleware({ app });

app.listen(4006, () => {
  console.log('Server running at http://localhost:4006/graphql');
});
```

📘 2. Line-by-Line Explanation

GraphQL Types and Schema

graphql

```
type Message {
  id: ID!
  content: String!
  sender: String!
  timestamp: String!
}
```

- Define the `Message` type which will represent each chat message.

```graphql
type Subscription {
  messageSent: Message
}
```

- **Subscription** allows real-time updates whenever a message is sent.

Resolvers

Query Resolvers

```javascript
messages: async () => {
  return await Message.find();
}
```

- This query fetches all messages from the database.

Mutation Resolvers

```javascript
sendMessage: async (_, { content, sender }) => {
  const message = new Message({ content, sender });
  await message.save();
  pubsub.publish("MESSAGE_SENT", { messageSent: message });  // Trigger subscription
  return message;
}
```

- `sendMessage` handles new messages. After saving the message to MongoDB, it publishes to the `messageSent` subscription.

Subscription Resolvers

```javascript
messageSent: {
  subscribe: (_, __, { pubsub }) => {
```

```
    return pubsub.asyncIterator("MESSAGE_SENT");
  }
}
```

- Subscriptions listen for the `"MESSAGE_SENT"` event and send the new message to all subscribers.

🗝️ 3. Important Keywords

Keyword	Description
`ApolloServer`	Express middleware for setting up GraphQL server
`PubSub`	Used for publishing and subscribing to events in GraphQL
`asyncIterator`	Enables subscription in real-time with WebSockets
`graphql-subscriptions`	Enables GraphQL real-time updates

🧪 4. Test Inputs & Outputs

Start MongoDB:

```bash
mongod
```

Run the server:

```bash
node index.js
```

Open GraphiQL:

- Go to: http://localhost:4006/graphql

✅ Send a Message (Mutation)

```graphql
mutation {
  sendMessage(content: "Hello, how are you?", sender: "User1") {
    id
    content
    sender
    timestamp
  }
}
```

Output:

```json
{
  "data": {
    "sendMessage": {
      "id": "abc123",
      "content": "Hello, how are you?",
      "sender": "User1",
      "timestamp": "2025-04-29T12:00:00Z"
    }
  }
}
```

📥 Get All Messages (Query)

```graphql
query {
  messages {
    id
    content
    sender
    timestamp
```

```
  }
}
```

Output:

```json
{
  "data": {
    "messages": [
      {
        "id": "abc123",
        "content": "Hello, how are you?",
        "sender": "User1",
        "timestamp": "2025-04-29T12:00:00Z"
      },
      // other messages...
    ]
  }
}
```

🔔 Real-time Message Subscription (Subscription)

```graphql
subscription {
  messageSent {
    id
    content
    sender
    timestamp
  }
}
```

- When a new message is sent, it will automatically trigger a real-time update.

Output (After sending a new message):

```json
```

```
{
  "data": {
    "messageSent": {
      "id": "def456",
      "content": "How's everything?",
      "sender": "User2",
      "timestamp": "2025-04-29T12:05:00Z"
    }
  }
}
```

Notes:

- WebSocket is required for the subscription to work. The Apollo server handles this automatically.
- You can have multiple subscribers connected and when any new message is sent, all of them will be notified instantly.
