

Joshua Dajao

3/16/25

Post-Service

Create:

The screenshot shows the GraphQL Studio interface. The 'Operation' tab on the left contains a mutation query:

```
1 mutation($title: String!, $content: String!) {  
2   createPost(title: $title, content: $content) {  
3     id  
4     title  
5     content  
6   }  
7 }
```

. The 'Variables' tab below it shows the input:

```
1 {  
2   "title": "dajahouse",  
3   "content": "qwe",  
4 }
```

. The 'Response' tab on the right shows the JSON output:

```
{  
  "data": {  
    "createPost": {  
      "id": 2,  
      "title": "dajahouse",  
      "content": "qwe"  
    }  
  }  
}
```

. The status bar at the top right indicates a 200 status code and a response time of 77.0ms.

Read

The screenshot shows the GraphQL Studio interface. The 'Operation' tab on the left contains a query:

```
1 query {  
2   posts {  
3     id  
4     content  
5     title  
6   }  
7 }
```

. The 'Variables' tab below it shows the input:

```
1 {  
2   "title": "dajahouse",  
3   "content": "qwe2",  
4 }
```

. The 'Response' tab on the right shows the JSON output:

```
{  
  "data": {  
    "posts": [  
      {  
        "id": 2,  
        "content": "qwe",  
        "title": "dajahouse"  
      }  
    ]  
  }  
}
```

. The status bar at the top right indicates a 200 status code and a response time of 12.0ms.

Update

This screenshot shows a GraphQL Studio interface with a mutation operation. The operation is a `mutation($updatePostId: Int!, $title: String!, $content: String!){ updatePost(id: $updatePostId, title: $title, content: $content) { id title content } }`. The response shows the updated post with `id: 2`, `title: "working test"`, and `content: "test working"`. The variables section shows the input values: `{ "title": "working test", "content": "test working", "updatePostId": 2 }`.

```
Operation
1  mutation($updatePostId: Int!, $title: String!, $content: String!){
2    updatePost(id: $updatePostId, title: $title, content: $content) {
3      id
4      title
5      content
6    }
7  }
```

```
Response
{
  "data": {
    "updatePost": {
      "id": 2,
      "title": "working test",
      "content": "test working"
    }
  }
}
```

```
Variables
1  {
2    "title": "working test"
3    "content": "test working",
4    "updatePostId": 2
5  }
```

This screenshot shows a GraphQL Studio interface with a query operation. The operation is a `query{ posts { content id title } }`. The response shows a list of posts, with the first post having `content: "test working"`, `id: 2`, and `title: "working test"`. The variables section is empty.

```
Operation
1  query{
2    posts {
3      content
4      id
5      title
6    }
7  }
```

```
Response
{
  "data": {
    "posts": [
      {
        "content": "test working",
        "id": 2,
        "title": "working test"
      }
    ]
  }
}
```

```
Variables
1  {
2    "title": "test successful",
3    "content": "successful test",
4    "updatePostId": 4
5  }
```

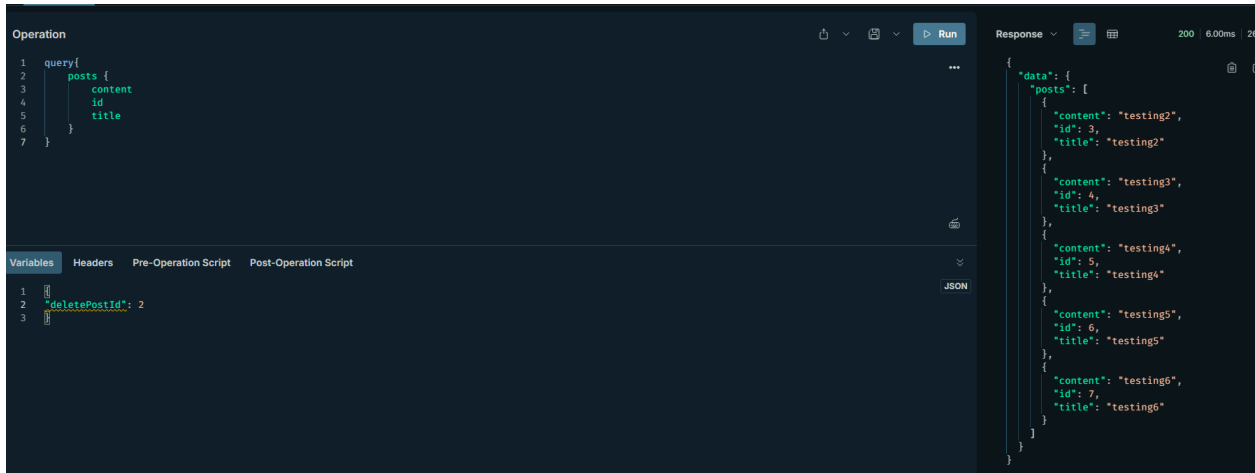
Delete

This screenshot shows a GraphQL Studio interface with a mutation operation. The operation is a `mutation($deletePostId: Int!) { deletePost(id: $deletePostId) { id } }`. The response shows the deleted post with `id: 2`. The variables section shows the input value: `{ "deletePostId": 2 }`.

```
Operation
1  mutation($deletePostId: Int!) {
2    deletePost(id: $deletePostId) {
3      id
4    }
5  }
```

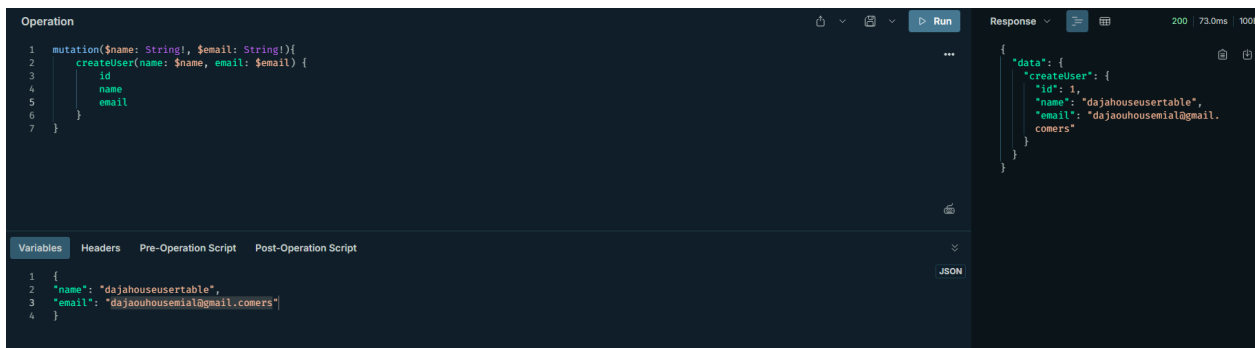
```
Response
{
  "data": {
    "deletePost": {
      "id": 2
    }
  }
}
```

```
Variables
1  {
2    "deletePostId": 2
3  }
```

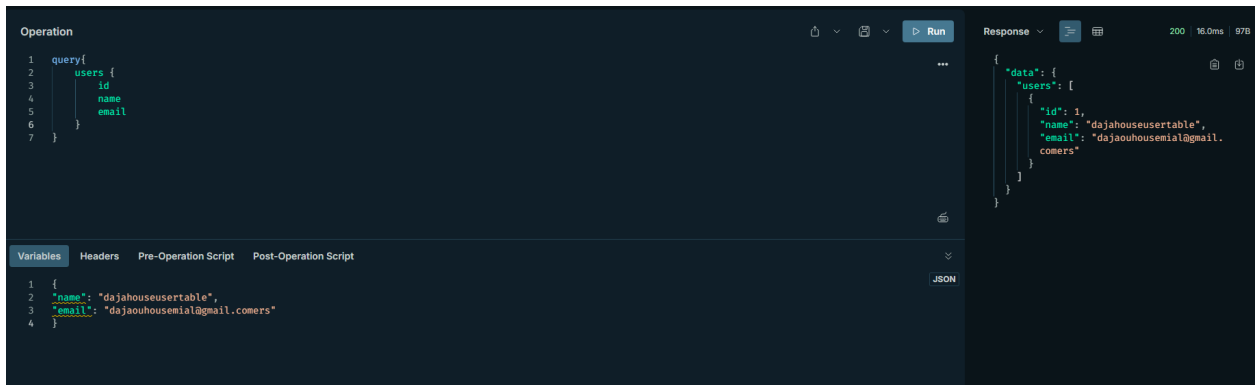


Users-table

Create



Read



Update

The screenshot shows the GraphQL Studio interface for an `updateUser` mutation. The operation is defined as follows:

```
1 mutation($updateUserId: Int!, $name: String!, $email: String!) {  
2   updateUser(id: $updateUserId, name: $name, email: $email) {  
3     id  
4     name  
5     email  
6   }  
7 }
```

The response is a JSON object:

```
{  
  "data": {  
    "updateUser": {  
      "id": 1,  
      "name": "Daqweudatey Updated",  
      "email": "daqwe@gmail.com"  
    }  
  }  
}
```

The Variables tab shows the input variables:

```
1 {  
2   "name": "Daqweudatey Updated",  
3   "email": "daqwe@gmail.com",  
4   "updateUserId": 1  
5 }
```

The screenshot shows the GraphQL Studio interface for a `query` to fetch all users. The operation is defined as follows:

```
1 query {  
2   users {  
3     id  
4     name  
5     email  
6   }  
7 }
```

The response is a JSON object:

```
{  
  "data": {  
    "users": [  
      {  
        "id": 1,  
        "name": "Daqweudatey Updated",  
        "email": "daqwe@gmail.com"  
      },  
      {  
        "id": 2,  
        "name": "dajahouseutable",  
        "email": "dajajouhousemia@gmail.comers"  
      },  
      {  
        "id": 3,  
        "name": "dajahouseutable",  
        "email": "dajajouh2@gmail.comers"  
      }  
    ]  
  }  
}
```

The Variables tab shows the input variables:

```
1 {  
2   "name": "dajahouseutable",  
3   "email": "dajajouhousemia@gmail.comers",  
4   "updateUserId": 2  
5 }
```

Delete

The screenshot shows the GraphQL Studio interface for a `deleteUser` mutation. The operation is defined as follows:

```
1 mutation($deleteUserId: Int!) {  
2   deleteUser(id: $deleteUserId) {  
3     id  
4   }  
5 }
```

The response is a JSON object:

```
{  
  "data": {  
    "deleteUser": {  
      "id": 2  
    }  
  }  
}
```

The Variables tab shows the input variables:

```
1 {  
2   "deleteUserId": 2  
3 }
```

What Do Database Migrations Do and Why Are They Useful?

From what I understand, database migrations work like a "snapshot" that captures the state of a database at a certain point in time. Each migration builds on the last one whenever there are changes or additions. They're useful because they keep track of the database's history, making it easy to go back to an earlier version if needed. It's kind of like how I can see MySQL and XAMPP working together—when I update something in MySQL, I can monitor those changes through XAMPP, which helps me manage and test my database more easily.

How Does GraphQL Differ from REST for CRUD Operations?

Based on discussions I've had with my friends while working on this output, GraphQL uses a single endpoint where clients can ask for just the data they need. This helps avoid problems like over-fetching or under-fetching. REST, on the other hand, usually has multiple endpoints with fixed responses, which can sometimes be inefficient. Another thing we talked about was how GraphQL's type system helps keep things clear between the client and server. Furthermore, in diving deeper into the iceberg of IT, in terms of making a modern web app, graphql definitely comes first compared to xampp because it's just a more flexible api, and just gives you what you need nothing more nothing less.