6. Resolver - post.ts

#graphql #resolver #mutation #query #mikroorm #backend

How to implement the Post resolver

- First we set up 5. GraphQL
- Then we implement the resolver:

Resolver(), Query(), Mutation(), Arg(), Ctx(), Int, are imported from type-graphql package;

Query() - only retrieves data from the DB, does not make any changes

Mutation() - makes changes to the DB

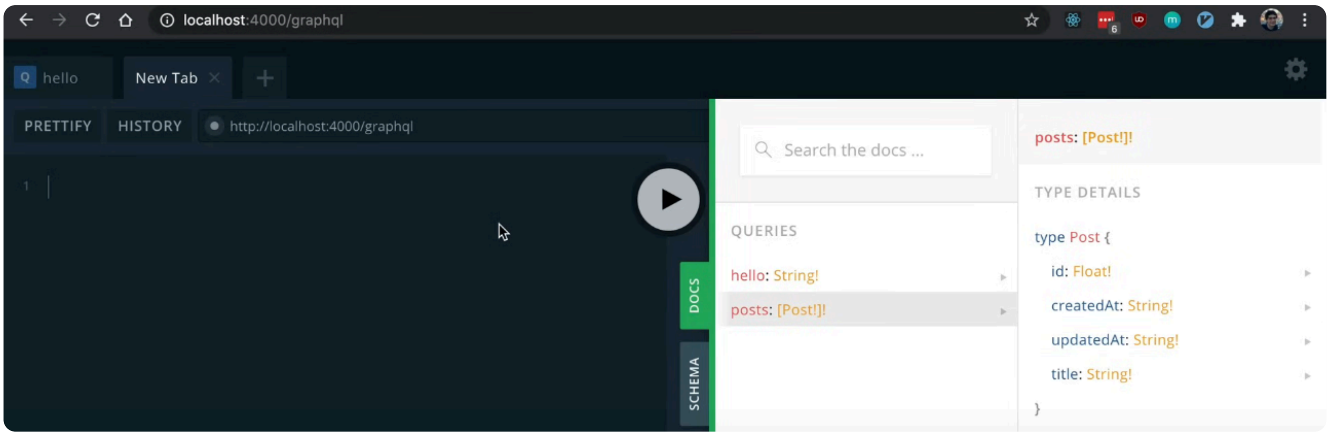
Arg() - is defined if the operation will have parameters

Ctx() - is the context provided by the Apollo server, accessible by all resolvers - defined here: Adding Context to the Apollo Server

/resolvers/post.ts

```
import { Post } from "../entities/Post";
import { MyContext } from "src/types";
import { Arg, Ctx, Int, Mutation, Query, Resolver } from "type-graphql";
// import { sleep } from "../utils/sleep";

@Resolver()
export class PostResolver {
  @Query(() => [Post]) // [Post] is how we define arrays in return type for the
resolver
  async posts(@Ctx() { em }: MyContext): Promise<Post[]> {
    //await sleep(3000); // simulate delay to test csr vs ssr load times
    return em.find(Post, {});
  }

  @Query(() => Post, { nullable: true })
  post(
    @Arg("id", () => Int) id: number, // "id" is the name to use in GraphQL
schema, id is the field name and type in DB
    @Ctx() { em }: MyContext
  ): Promise<Post | null> {
    return em.findOne(Post, { id });
  }
  @Mutation(() => Post)
  async createPost(
    @Arg("title", () => String) title: string,
    @Ctx() { em }: MyContext
  ): Promise<Post> {
    const post = em.create(Post, { title });

    await em.persistAndFlush(post);
    return post;
  }

  @Mutation(() => Post, { nullable: true })
  async updatePost(
    @Arg("id") id: number, // here we ommitted type declaration in @Arg - type
inference works for Int and String
    @Arg("title", () => String, { nullable: true }) title: string, // here we
explicitly set type since we want to make it nullable
    @Ctx() { em }: MyContext
  ): Promise<Post | null> {
    const post = await em.findOne(Post, { id });

    if (!post) {
      return null;
    }

    if (typeof title !== "undefined") {
      post.title = title;
      await em.persistAndFlush(post);
    }

    return post;
  }

  @Mutation(() => Boolean)
  async deletePost(
    @Arg("id") id: number,
    @Ctx() { em }: MyContext
  ): Promise<boolean> {
    const post = await em.findOne(Post, { id });
    if (!post) {
      return false;
    }

    await em.nativeDelete(Post, { id });
    return true;
  }
}
```

- and now we have the localhost:4000 /graphql endpoint as follows:

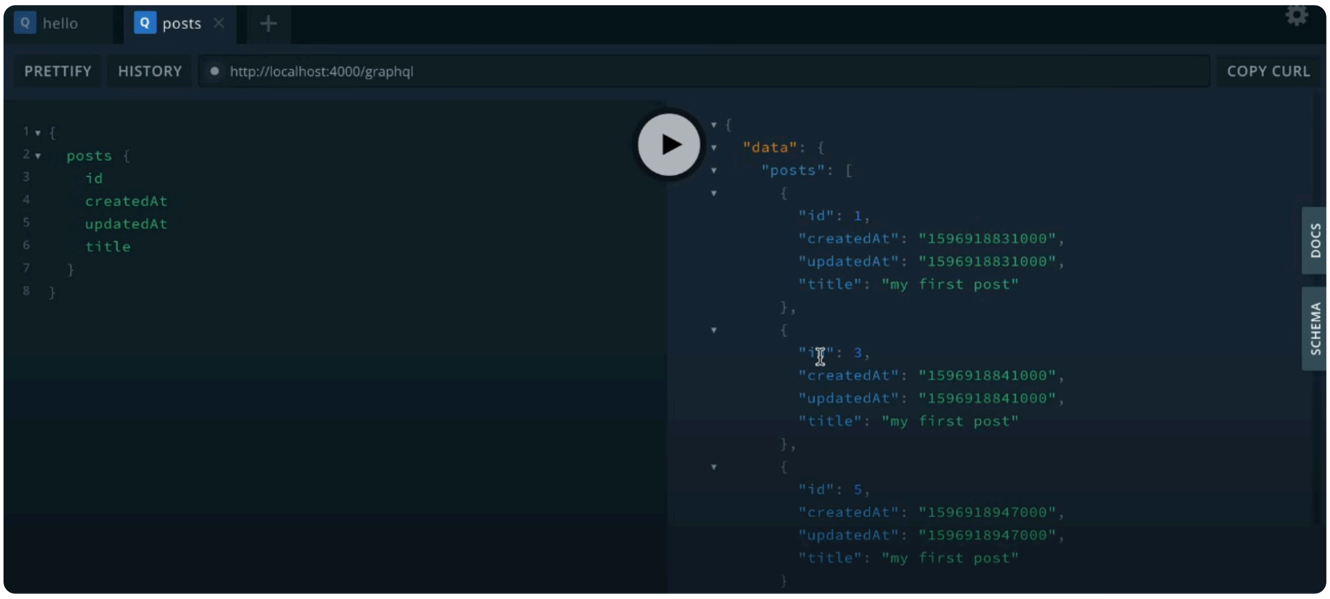


hello: String!
posts: [Post!]!

posts: [Post!]!

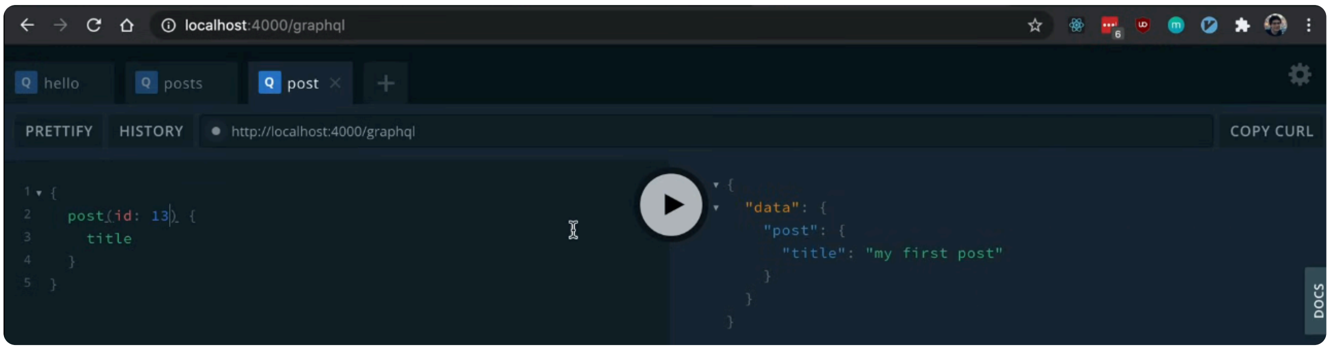
```
type Post {
  id: Float!
  createdAt: String!
  updatedAt: String!
  title: String!
}
```

Running the GraphQL queries



```
{
  posts {
    id
    createdAt
    updatedAt
    title
  }
}
```

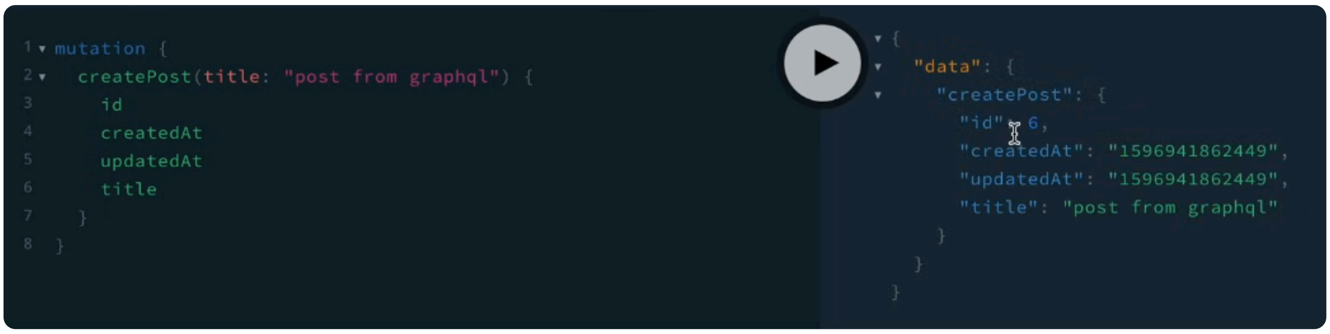
```
{
  "data": {
    "posts": [
      {
        "id": 1,
        "createdAt": "1596918831000",
        "updatedAt": "1596918831000",
        "title": "my first post"
      },
      {
        "id": 3,
        "createdAt": "1596918841000",
        "updatedAt": "1596918841000",
        "title": "my first post"
      },
      {
        "id": 5,
        "createdAt": "1596918947000",
        "updatedAt": "1596918947000",
        "title": "my first post"
      }
```



```
{
  post(id: 13) {
    title
  }
}
```

```
{
  "data": {
    "post": {
      "title": "my first post"
    }
  }
}
```

- for mutations the syntax is as follows:



```
mutation {
  createPost(title: "post from graphql") {
    id
    createdAt
    updatedAt
    title
  }
}
```

```
{
  "data": {
    "createPost": {
      "id": 6,
      "createdAt": "1596941862449",
      "updatedAt": "1596941862449",
      "title": "post from graphql"
    }
  }
}
```