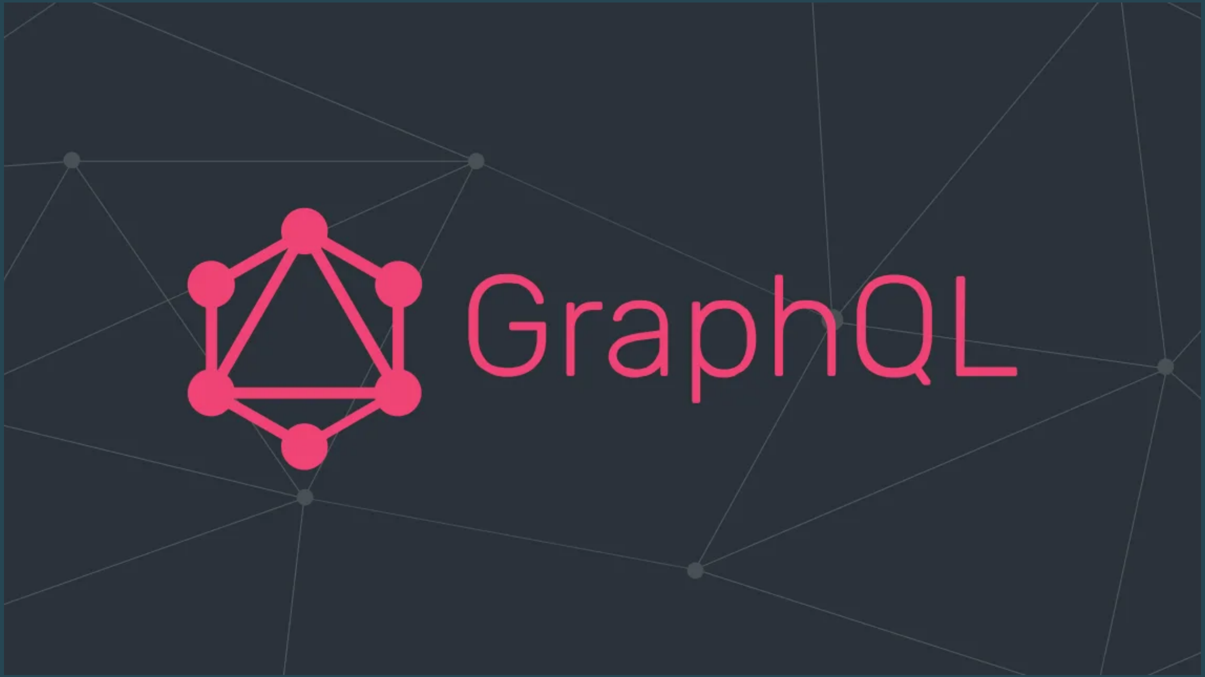

GraphQL

What is GraphQL?

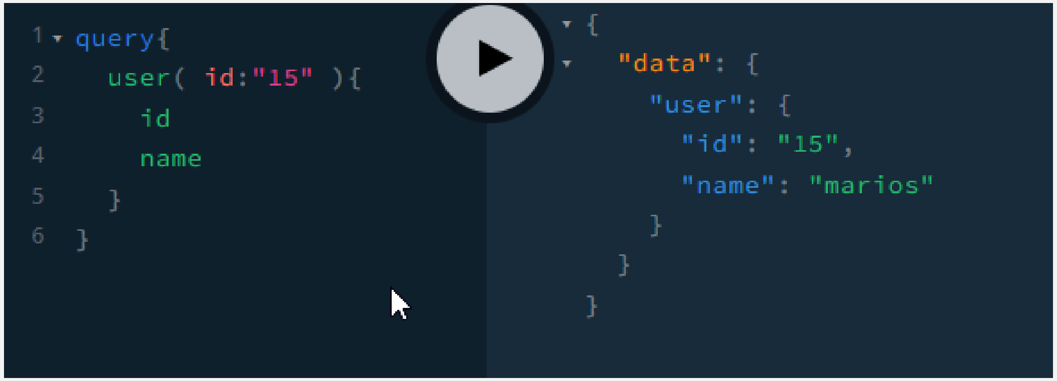
GraphQL is a query language for your API, and a server-side runtime for executing queries using a type system you define for your data. GraphQL isn't tied to any specific database or storage engine and is instead backed by your existing code and data.

For more information touch here.

Queries

What is Queries?

Let's start by looking at a very simple query and the result we get when we run it:



```
query{
  user( id:"15" ){
    id
    name
  }
}
```

```
{
  "data": {
    "user": {
      "id": "15",
      "name": "marios"
    }
  }
}
```

- This query take from the user with `id = 15` the name and id fields.
- The field `id` returns a String type, "15". The field `name` returns a String type, "marios".
- In the Start we use the keyword `query`.

Up to this point, you only learned how to fetch data in GraphQL. But the real-world applications are complex and you also need to insert, update or delete data. The question is:

How do you do it?

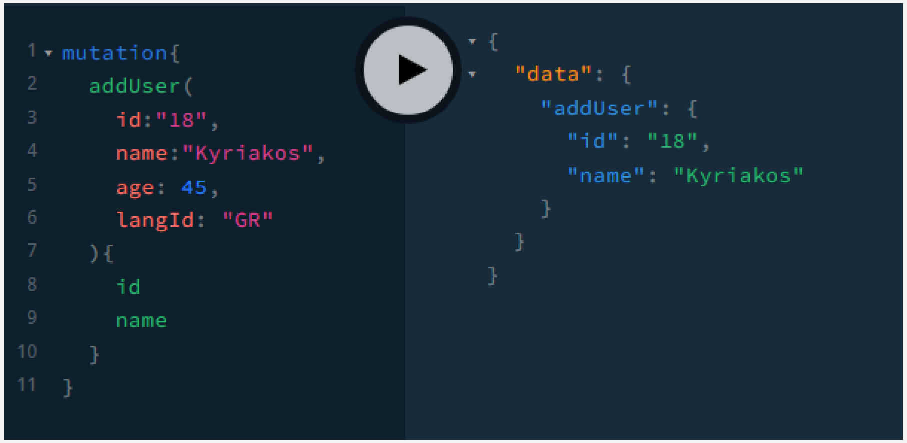
Mutation

What is Mutation?

In GraphQL, you insert, update or delete data with mutations.

A Mutation is a GraphQL Operation that allows you to insert new data or modify the existing data on the server-side.

Example



```
mutation{
  addUser(
    id:"18",
    name:"Kyriakos",
    age: 45,
    langId: "GR"
  ){
    id
    name
  }
}
```

```
{
  "data": {
    "addUser": {
      "id": "18",
      "name": "Kyriakos"
    }
  }
}
```

- In this `mutation` we introduce a `User` with fields (`id = 18`, `name = Kyriakos`, `age = 45`, `languageId = GR`).
- The response is this `User` and we take fields `id` and `name`.
- In the Start we use the keyword `mutation`.

You can started learning more for `GraphQl` from here.

You get started working which support `GraphQL` in all sorts of languages.

GraphQL with Rust Programming Language

Check here the Rust Libraries for Server and Client.

Next we will talk about Async-graphql server-side library.

`Async-graphql`

Async-graphql is a GraphQL server-side library implemented in Rust. It is fully compatible with the GraphQL specification and most of its extensions, and offers type safety and high performance.

Let's go create the `Schema`.

What is Schema?

The Schema of a GraphQL contains a required Query, an optional Mutation, and an optional Subscription. These object types are described using the structure of the Rust language. The field of the structure corresponds to the field of the GraphQL object.

And the question is:

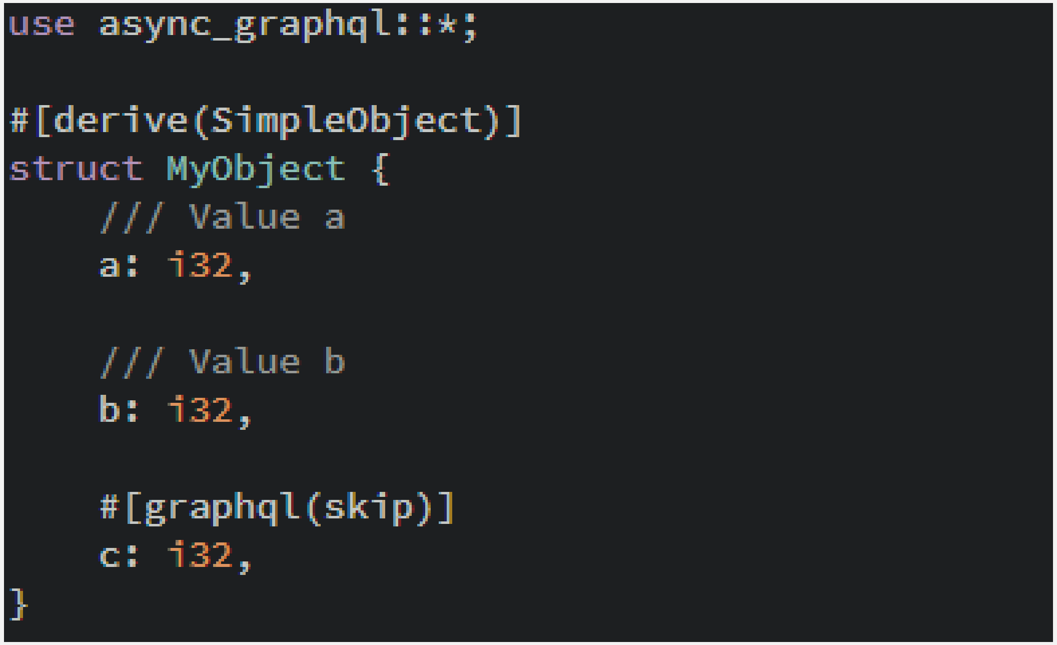
What is `Object` or `SimpleObject`?

Simple Object

What is Simple Object?

`SimpleObject` directly maps all the fields of a struct to `GraphQL` object. If you don't require automatic mapping of fields, see Object.

The example below defines an object MyObject which includes the fields a and b. c will be not mapped to GraphQL as it is labelled as `#[graphql(skip)]`.



```rust
use async_graphql::*;

#[derive(SimpleObject)]
struct MyObject {
    /// Value a
    a: i32,

    /// Value b
    b: i32,

    #[graphql(skip)]
    c: i32,
}
```

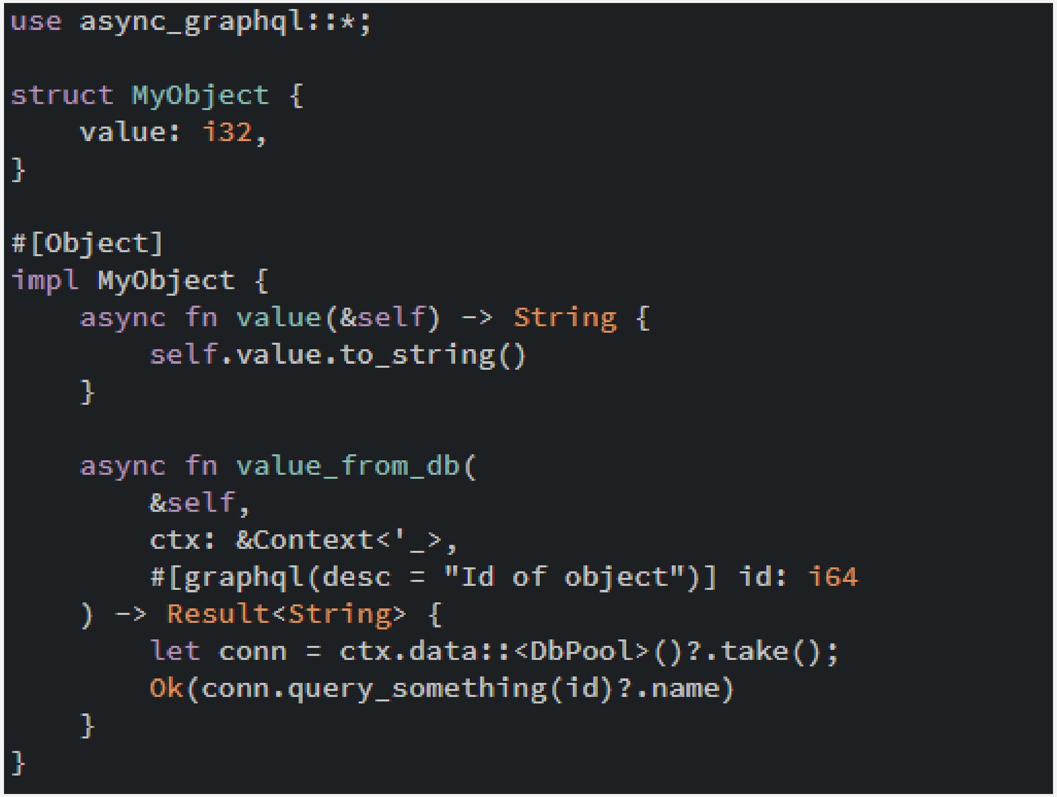
Object

What is Object?

- Different from `SimpleObject`, `Object` must have a resolver defined for each field in its `impl`.
    - A resolver function has to be `asynchronous`. The first argument has to be `&self`, the second is an optional Context and it is followed by field arguments.
- When creating your Schema, you can use `SchemaBuilder::data` to configure the `global data`, and `Context::data` to configure Context data.
- The following `value_from_db` function shows how to retrieve a `database` connection from `Context`.

Check the `next` slide for the `Example`.

Example

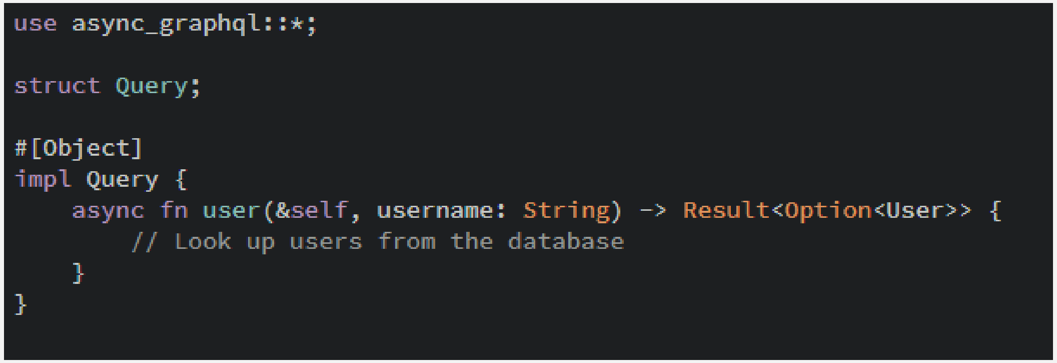


```rust
use async_graphql::*;

struct MyObject {
    value: i32,
}

#[Object]
impl MyObject {
    async fn value(&self) -> String {
        self.value.to_string()
    }

    async fn value_from_db(
        &self,
        ctx: &Context<'_>,
        #[graphql(desc = "Id of object")] id: i64
    ) -> Result<String> {
        let conn = ctx.data::<DbPool>()?.take();
        Ok(conn.query_something(id)?.name)
    }
}
```

Query root Object

What is Query root Object?

The query root object is a GraphQL object with a definition similar to other objects. Resolver functions for all fields of the query object are executed concurrently.



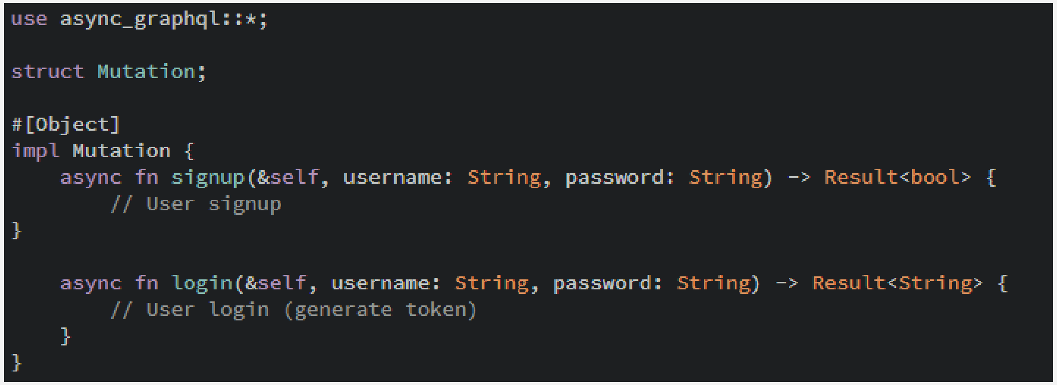
```rust
use async_graphql::*;

struct Query;

#[Object]
impl Query {
    async fn user(&self, username: String) -> Result<Option<User>> {
        // Look up users from the database
    }
}
```

Mutation root Object

What is Mutation root Object?

The mutation root object is also a GraphQL object, but it executes sequentially. One mutation following from another will only be executed only after the first mutation is completed.

The following mutation root object provides an example of user registration and login:



```
use async_graphql::*;

struct Mutation;

#[Object]
impl Mutation {
    async fn signup(&self, username: String, password: String) -> Result<bool> {
        // User signup
}

    async fn login(&self, username: String, password: String) -> Result<String> {
        // User login (generate token)
    }
}
```

Now, we have create our Objects(Query, Mutation, Objects and SimpleObjects).

Let's go create the schema in our program(main.rs).

We will use the Axum framework.

In Cargo.toml

```
[dependecies]
```

```
async-graphql = "4.0.12"
async-graphql-axum = "4.0.11"
```

In main.rs

We can connect the schema with our base or with any Object.

`With our database:`

Here, we have create a `Base`(for example `Mongo-Base`).

```
async fn main(){
  let schema = Schema::build(Query, Mutation, EmptySubscription)
                .data(my_database)
                .finish();
}
```

`With our Object:`

Here, we have create the `Users` Object(remember from previous slides).

```
async fn main(){
  let schema = Schema::build(Query, Mutation, EmptySubscription)
                .data(Users::new())
                .finish();
}
```

Good step.

In `axum` we use the `routers`.
Let's go to see how create the routers with `Graphql`.

We will connect the data with `User` Object.

```
async fn main(){

  //Schema
  let schema = Schema::build(Query, Mutation, EmptySubscription)
                 .data(my_database)
                 .finish();
  //Router
  let app = Router::new()
      .route("/api/graphql", get(graphql_playground).post(graphql_handler
      .layer(Extension(schema));

  //Server up
  axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
      .serve(app.into_make_service())
      .await
      .unwrap();
}

async fn graphql_handler(
  schema: Extension< Schema<Query, EmptyMutation, EmptySubscription>>,
  Json(req): Json<Request>
) -> Json<Response> {
  schema.execute(req.).await.into()
}

async fn graphql_playground() -> impl IntoResponse {
  Html(playground_source(GraphQLPlaygroundConfig::new("/api/graphql")))
}
```

You have a lot of information to get you started in `GraphQL`.

If you want to check out my Project with GraphQL, tap here.

Tips

If you want learning GraphQL with Rust:

- Read for GraphQL.
- Read the Async-graphql Book.
- Check the Documentation for `Async-graphql`.
- Clone the Repository with `Asynch-graphql` examples.
- Check the Cargo-Package.

*Don't lose time **start** write code.*

-- Dimitris Rammos