Detailed Step-by-Step Guide

Intro

Welcome to this presentation about GraphQL and how to implement a GraphQL service in Java using Spring for GraphQL and JooQ for database access.

This is not a basic introduction to GraphQL, but for those of you who do not know GraphQL much, don't worry, we will start with a basic hello world style example, and we will introduce more advanced features progressively.

The main objective is to highlight some important attention point in the implementation, more specifically related to potential performance issues, and to address them.

This talk is mainly a live coding presentation, so there will be very few slides.

Resources

References

- GraphQL
- Spring for GraphQL
- JooQ
- LangChain4J
- IntelliJ GraphQL Plugin
- Voyager

Sources

- Source code

documentation

- Slides
- Detailed step-by-step guide

Contact

- Email: didier.pirottin@gmail.com

http://bit.ly/3SEtuWb

Hello World

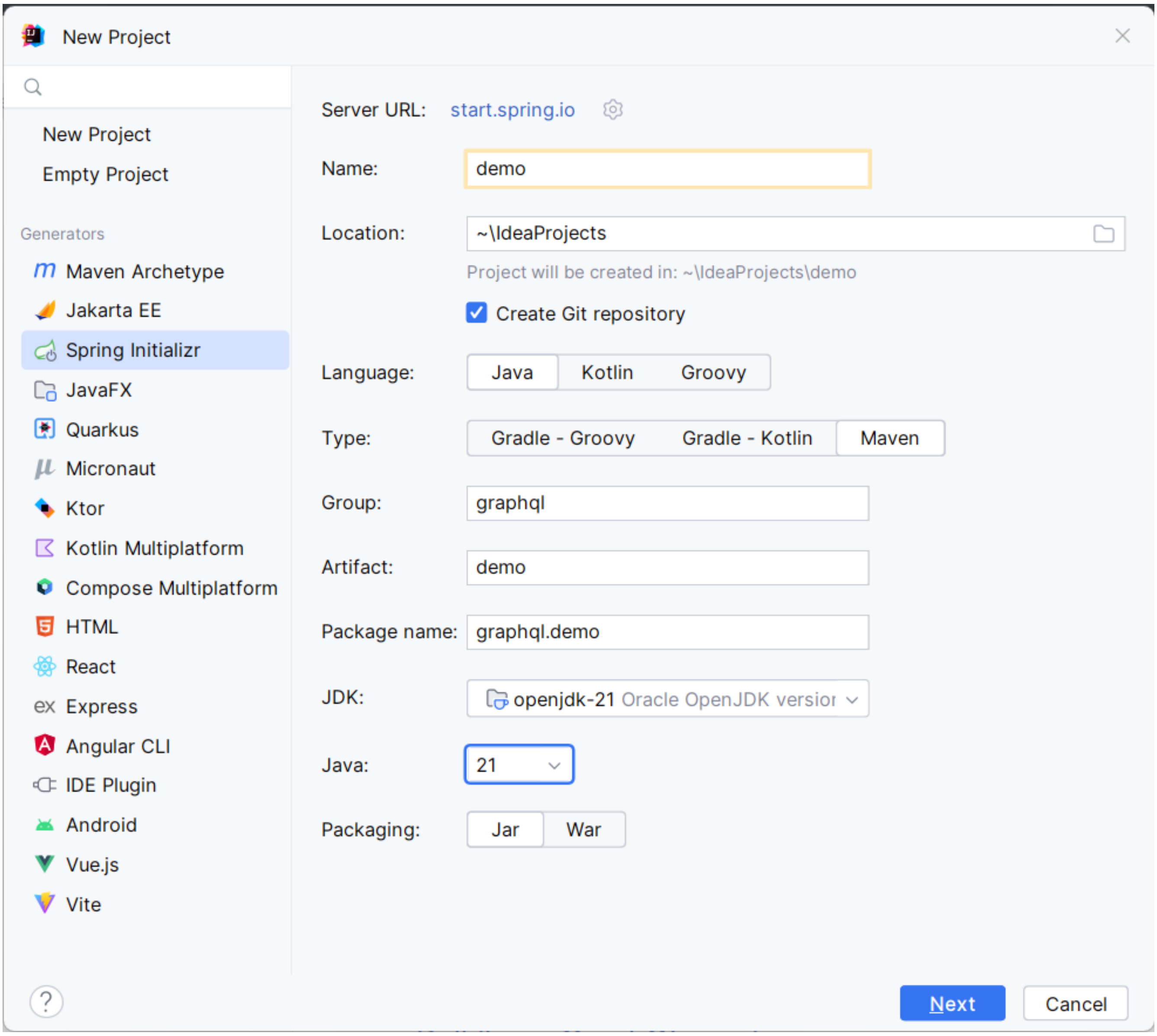
Create the workspace

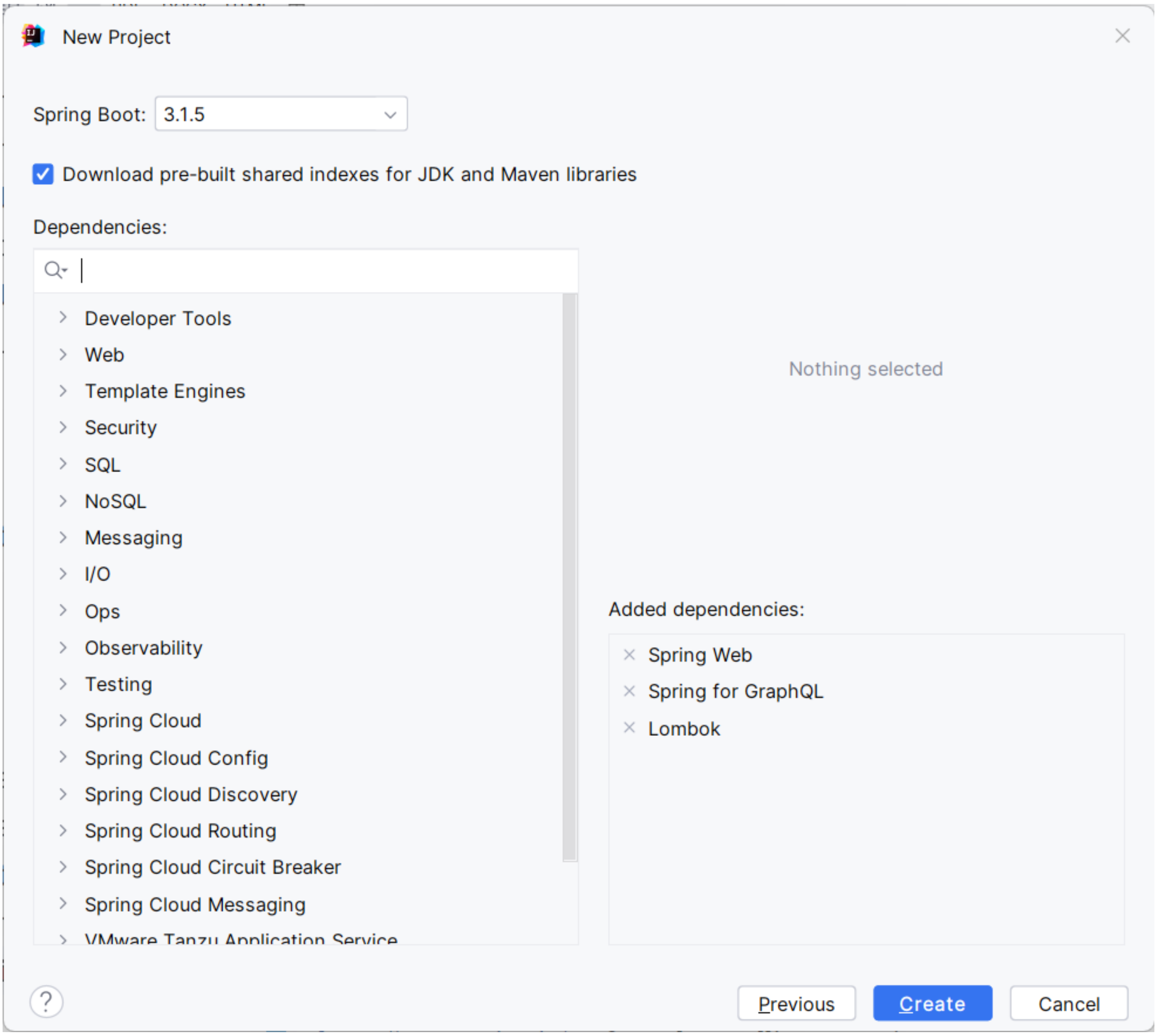
So let's start with a hello world style service.

For this demo, we will start by creating a new GraphQL service, using Spring Initialzr. You can use it from https://start.spring.io/ or from the IntelliJ new project menu.

We will use Java 21, Spring Boot 3.1.5, Maven, just a few starters for now :

- Web
- GraphQL
- DevTools





To make it a little more interesting, we will also use the "LangChain4J" lib, so let's add it to the dependencies in the pom file.

pom.xml

```
<dependency>
    <groupId>dev.langchain4j</groupId>
    <artifactId>langchain4j</artifactId>
    <version>0.23.0</version>
</dependency>
<dependency>
    <groupId>dev.langchain4j</groupId>
    <artifactId>langchain4j-open-ai</artifactId>
    <version>0.23.0</version>
</dependency>
```

Create the GraphQL Schema

Now, let's create a GraphQL Schema file, in the `graphql` resource folder, and call it demo.

demo.graphql

```
type Query { ①
    demo: String ②
}
```

① `Query` is a special type in GraphQL, it is the entry point in the GraphQL data model.

② We simply add an entry "demo" that will return a String

Update the properties file

Update the properties file to enable the **GraphiQL** endpoint and prepare the configuration url for **langChain4J** to call openAI later on.

application.properties

```
spring.graphql.graphiql.enabled=true ①
demo.openai-proxy-url=http://langchain4j.dev/demo/openai/v1 ②
```

① Enable the GraphiQL endpoint

② This is the url that will be used by langChain4J to call openAI

Run the app

We just start the app now, go to the graphiql endpoint and see the default behaviour, which is returning a null for the created endpoint.

This is the default behaviour when no controller is defined.

Create the Controller

Now, let's creat the "DemoController" class annotated @Controller and create a method "demo", annotated by @QueryMapping such that Spring will map any call to the demo entry point to this method.

DemoController.java

```
@Controller ①
@RequiredArgsConstructor
public class DemoController {
    private final OpenAiService openAiService;

    @QueryMapping ②
    String demo() { ③
```

```
        return openAiService.demoWelcomeMessage(); ④
    }
}
```

① This is a Spring controller

② This method will be mapped to an attribute of the `Query` type from GraphQL schema. By default, the mapped attribute will be the one corresponding to the method name.

③ Make sure to give a method signature that is compatible with the corresponding definition in the GraphQL schema.

④ Delegate to the OpenAiService to get a demo message.

Here, we use **LangChain4J** library to get a text from openAI, using the `OpenAiService`

OpenAiService.java

```
public String demoWelcomeMessage() {
    return model.generate(
            """
            Write a greeting message for a live coding demo talking about Spring for
GraphQL.
            The message must be short, no more than 50 words.
            """);
}
```

Running the query again will now give the answer from ChatGpt.

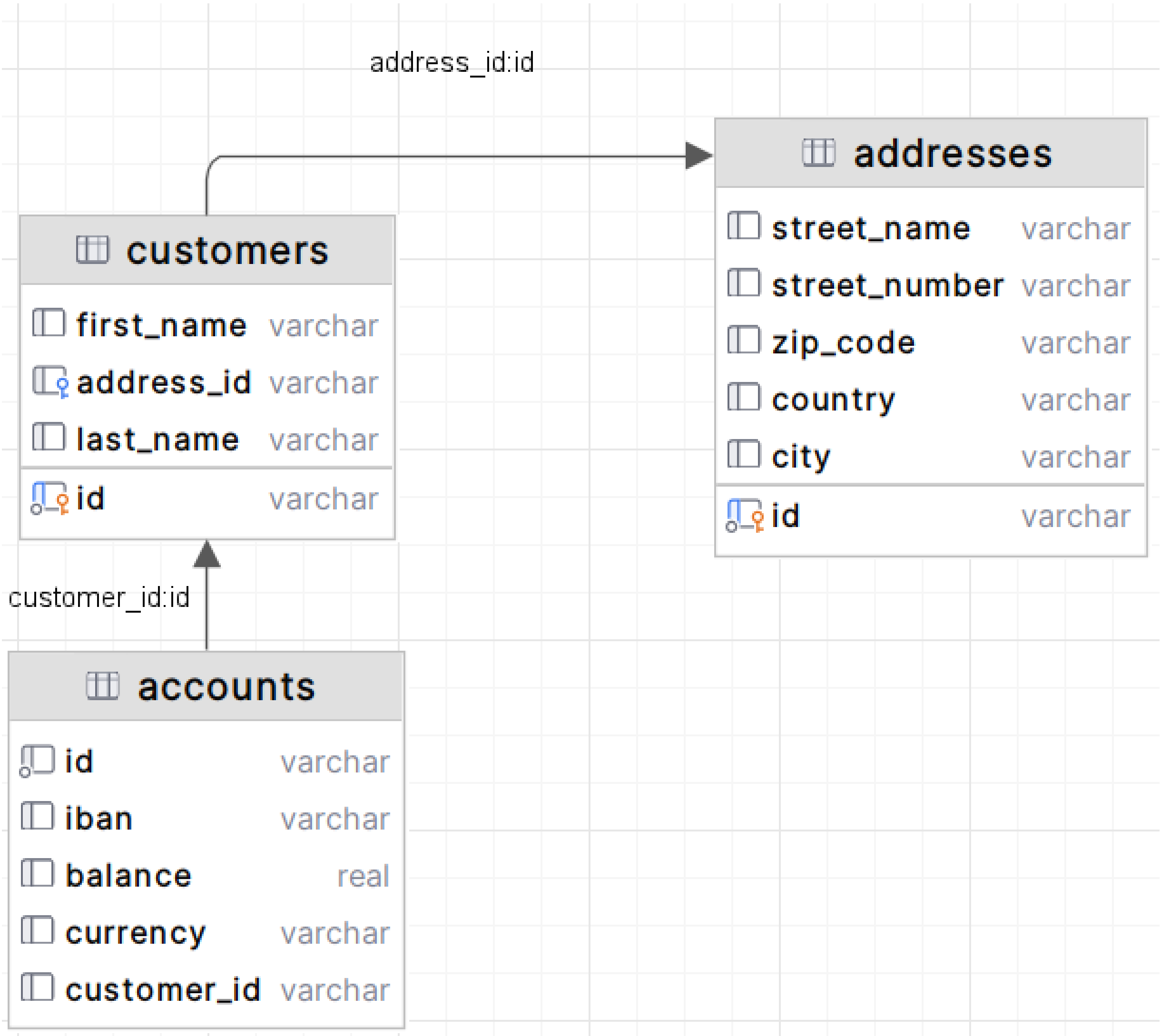
query response

```
{
  "data": {
    "demo": "\"Welcome to the Spring for GraphQL demo!Get ready to spring into action
as we dive into the world of software development with a twist of humor. Let's break
some code and have a blooming good time!\"",
  }
}
```

A more realistic sample with a Database

Now let’s give a more realistic example, with some data stored in a customer database.

The database is a Postgres DB, with the following tables :



- A **customer** table, containing customers with their first name and laste name
- An **address** table, containing addresses, an address could be shared by multiple customers
- And an **account** table, each customer could have multiple accounts.

API First

GraphQL promotes the **API first** approach through its DSL for designing the API. So, we'll start the implementation by specifying the API we would like to expose in the graphql schema.

demo.graphqls

```
type Query {
    customers: [Customer]
}

type Customer {
    id: ID!
    firstName: String!
    lastName: String!
    address: Address
    accounts: [Account]!
```

```
}

type Account {
    id: ID!
    iban: String!
    balance: Float!
    currency: String!
}

type Address {
    streetNumber: String!
    streetName: String!
    zipCode: String!
    city: String!
    country: String!
}
```

Note the following syntax in GraphQL :

- `[<typeName>]` means this is a list of `<typeName>`
- `<typeName>!` means that this value will never be `null`

Visualize the GraphQL data model using Voyager

One of the advantages of the well-defined GraphQL specification is the possibility to create useful tools around it. Voyager is one these tools, let's just create a simple html file and see how the schema we just created wan be visualised to explore the GraphQL API.

voyager.html

```
<!doctype html>
<html>
<head>
    <link
        rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/graphql-voyager/dist/voyager.css"
    /> ①
    <script
        src="https://cdn.jsdelivr.net/npm/graphql-voyager/dist/voyager.standalone.js">
    </script> ①
</head>

<body>
<div id="voyager">Loading...</div>
<script type="module">
    const {voyagerIntrospectionQuery: query} = GraphQLVoyager;
    const response = await fetch(
        'http://localhost:8082/graphql', ②
        {
```

```
            method: 'post',
            headers: {
                Accept: 'application/json',
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({query}),
            credentials: 'omit',
        },
    );
    const introspection = await response.json();

    // Render <Voyager /> into the body.
    GraphQLVoyager.renderVoyager(document.getElementById('voyager'), {
        introspection,
    });
</script>
</body>
</html>
```

① Include CSS and Javascript from CDN

② Configure the GraphQL endpoint

Here is a screenshot of the Voyager UI on our basic example :

Datasource and JooQ configuration

Before implementing the API, we need to add a few dependencies to access the Postgres DB. Here, we will use JooQ as the database library. As we will see later on, this library is a perfect match for GraphQL.

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jooq</artifactId> ①
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.6.0</version> ②
</dependency>
```

① JooQ Spring Starter

② PostgreSql driver

JooQ code generation

As JooQ is based on **code generation**, from the dabase schema, we also add a **JooQ maven plugin** to perform the required code generation.

pom.xml

```
<plugin>
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <version>3.18.6</version>

    <executions>
        <execution>
            <id>jooq-codegen</id>
            <phase>generate-sources</phase> ①
            <goals>
                <goal>generate</goal>
            </goals>
        </execution>
    </executions>

    <configuration>
        <!-- Configure the database connection here -->
        <jdbc> ②
            <driver>org.postgresql.Driver</driver>
            <url>jdbc:postgresql://localhost:5432/graphql-demo</url>
            <user>demo</user>
            <password>demo</password>
        </jdbc>

        <generator>
            <!-- The default code generator. -->
            <name>org.jooq.codegen.JavaGenerator</name>

            <database> ③
                <!-- The database type. -->
                <name>org.jooq.meta.postgres.PostgresDatabase</name>
                <!-- The database schema to be generated -->
                <inputSchema>public</inputSchema>
                <!-- All elements that are generated from your schema -->
                <includes>.*</includes>
            </database>

            <target> ④
                <!-- The destination package of your generated classes (within the
destination directory) -->
                <packageName>graphql.demo.jooq.generated</packageName>
               <!-- The destination directory of your generated classes. Using Maven
directory layout here -->
                <directory>target/generated-sources</directory>
            </target>
        </generator>
    </configuration>
</plugin>
```

① The plugin will be executed during the **code-generation** phase.

② As the code generator will use the database schema data, it requires a configuration to access the database

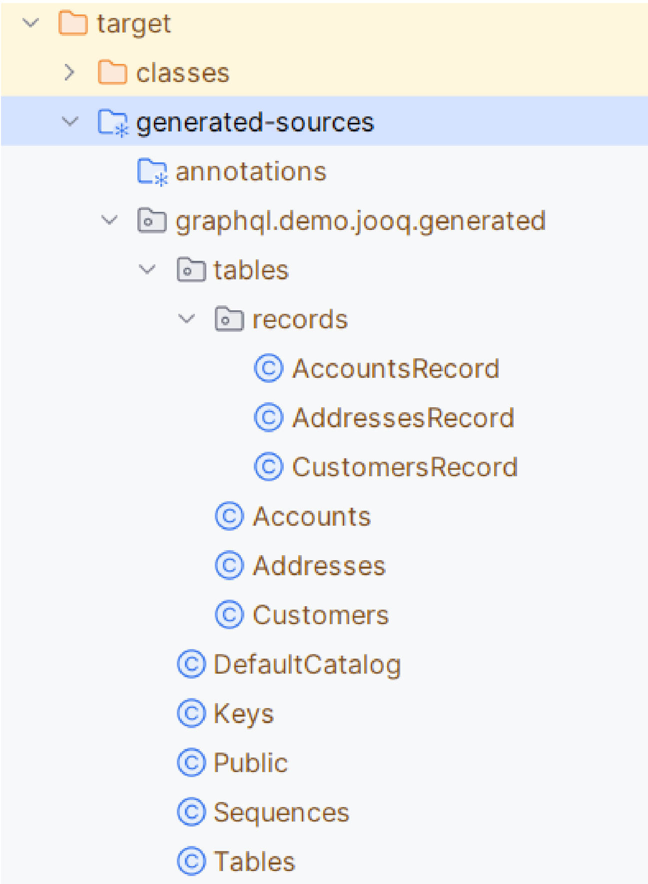
③ This section configure what has to be generated

④ And this section configure where the code will be generated

See Code Generation JooQ documentation for more details and options.

We can now generate the code, compiling our project using maven.

Make sure the generated code is considered as source code by your IDE.



Let’s explore the generated code, it contains

- A class per table, containing information about the table and its columns
- A **"record" Class** (it’s not Java records) that can be used to get data returned by SQL queries.

The exposed model classes

JooQ generates classes that map the Database model.

On the other hand, we need to return data that maps the GraphQL data model and most of the time, *there are differences between these two models*.

A good practice is hence to **decouple** these two models by creating "**GraphQL model classes**" that are more aligned to the GraphQL data model. This is similar to the DTOs (Data Transfer Object) used in REST services.

So, let’s create a CustomerModel as a simple Java record.

CustomerModel.java

```
public record CustomerModel(
        String id,
        String firstName,
        String lastName,
        String addressId) { }
```

> Note here that we also define the "addressId" in this model. It is not *exposed* in the GraphQL data model, but it will be used to get the *exposed address* as we will see later on.

Similarly, we can already create the other model classes of our GraphQL schema : `AddressModel` and `AccountModel`.

CustomerController

To implement the API, we will create a `CustomerController` class.

CustomerController.java

```
@Controller
public class CustomerController {
    private final DSLContext dslContext; ①

    public CustomerController(DSLContext dslContext) {
        this.dslContext = dslContext;
    }

    @QueryMapping ②
    List<CustomerModel> customers() { ③
        return dslContext.select(CUSTOMERS.asterisk()) ④
                .from(CUSTOMERS)
                .fetch()
                .stream()
                .map(CustomerModelMapper::mapCustomerRecordToModel) ⑤
                .toList();
    }
}
```

① The dslContext is the JooQ object that will be used to create SQL queries

② The `@QueryMapping` Spring annotation indicates that this method will be mapped to the corresponding GraphQL attribute on the `Query` type

③ The method signature must be aligned with the corresponding GraphQL definition

④ Here, we use the `dslContext` to create the query on the Customer table and fetch the results

⑤ Then we map the returned `CustomerRecord` to the `CustomerModel` and return the resulting list.

To map the `CustomerRecord` to a `CustomerModel`, we define a separate helper class `CustomerModelMapper`.

CustomerModelMapper.java

```
public static CustomerModel mapCustomerRecordToModel(Record record) {
    if (record == null) {
        return null;
    }
    CustomersRecord customerRecord = record.into(CUSTOMERS); ①
        return new CustomerModel(
                customerRecord.getId(),
                customerRecord.getFirstName(),
                customerRecord.getLastName(),
                customerRecord.getAddressId()
        );
}
```

① We use the `into` method to map the generic record to a `CustomerRecord`, this is the specific record classes generated by JooQ for each table.

② We map the JooQ `CustomerRecord` to our model `CustomerModel`.

> In a similar way, we can already implement `AddressModelMapper` and `AccountModelMapper`

We can now test our service with a simple query :

query

```
query {
  customers {
    firstName
    lastName
  }
}
```

Fetching the customer's address

Now, if we try to get customer address data, such as in this query :

query

```
query {
  customers {
    firstName
    lastName
    address {
      streetNumber
      streetName
```

```
        zipCode
        city
        country
      }
    }
  }
```

All the returned addresses are `null` by default.

So let's add an `@SchemaMapping` to the CustomerController :

CustomerController.java

```
@SchemaMapping(typeName = "Customer") ①
AddressModel address(CustomerModel customer) { ②
    if (customer.addressId() == null) { ③
        return null;
    }
    return dslContext.selectFrom(ADDRESSES)
            .where(ADDRESSES.ID.eq(customer.addressId())) ④
            .fetchOne()
            .map(AddressModelMapper::mapAddressRecordToModel); ⑤
}
```

① The `@SchemaMapping` annotation maps the attributes of a given GraphQL type, passed as argument in the annotation.

② The name of the mapped attribute is the method name, by default. Also note here that the current `CustomerModel` is injected in the method called by Spring.

③ If the customer does not have an `addressId`, we simply return null, as it has no known address

④ We perform a SQL query on the `Addresses` table whith a where clause based on the `addressId` of the given customer.

⑤ Finaly, we map the `AddressRecord` to an `AddressModel` using the previously define model mapper.

We can now run our query again, and it returns the address data, for the customers having an address in the database.

Fetching the customer's accounts

Similarly, if we test a query returning the customer's accounts data, such as this one

query

```
query {
  customers {
    firstName
    lastName
    accounts {
      iban
```

```
      balance
      currency
    }
  }
}
```

We get an error, because the GraphQL model defines the `accounts` attribute as being "non null" and the current implementation returns `null`.

To fix this, we define a new @SchemaMapping for the `accounts` attribute.

CustomerController.java

```java
@SchemaMapping(typeName = "Customer")
List<AccountModel> accounts(CustomerModel customer) {
    return dslContext.selectFrom(ACCOUNTS)
            .where(ACCOUNTS.CUSTOMER_ID.eq(customer.id()))
            .fetch()
            .stream()
            .map(AccountModelMapper::mapAccountRecordToModel)
            .toList();
}
```

Here, the code will never return null, as it will return an empty list if no account is found.

Now, if we execute the query again, we get the list of accounts for each customer.

Adding filtering (using JooQ)

Now let's add some filtering to filter the customer's by their names:

demo.graphqls

```
type Query {
  customers(filter: CustomerPredicate): [Customer] ①
}

input CustomerPredicate { ②
    firstName: StringPredicate
    lastName: StringPredicate
}

input StringPredicate { ③
    is: String
    isNot: String
    contains: String
    startsWith: String
    endsWith: String
```

```
    isOneOf: [String]
}
```

① Any attribute in a GraphQL schema can declare input parameters. We use it to pass an optional `filter` parameter

② The `CustomerPredicate` is an **input** type in GraphQL, it declares the different filter input we will accept for filtering our customers

③ We use a generic predicate on Strings

Let's add Java classes to map these new types in our model package :

CustomerPredicate.java

```
public record CustomerPredicate(
        StringPredicate firstName,
        StringPredicate lastName) {
}
```

StringPredicate.java

```
public record StringPredicate(
        String is,
        String isNot,
        List<String> isOneOf,
        String contains,
        String startsWith,
        String endsWith) {
        }
```

Now, we can modify our `CustomerController` to use the filter.

CustomerController.java

```
@QueryMapping
List<CustomerModel> customers(@Argument CustomerPredicate filter) { ①
    @NotNull SelectJoinStep<Record> query = dslContext.select(CUSTOMERS.asterisk())
            .from(CUSTOMERS); ②
    if (filter != null) {
        filter.applyOn(query); ③
    }
    return query
            .fetch()
            .stream()
            .map(CustomerModelMapper::mapCustomerRecordToModel)
            .collect(toList());
}
```

① The GraphQL parameter is passed to the method, using the @Argument annotation

② We split the creation of the query, to be able to add the filter later on

③ If a filter is provided, we apply the filter on the query, using the `applyOn` method.

Then we use JooQ to apply the filter on the query.

CustomerPredicate.java

```
public SelectJoinStep<Record> applyOn(SelectJoinStep<Record> query) {
    ifNonNull(firstName, firstName -> query.where(firstName.conditions(CUSTOMERS
.FIRST_NAME))); ①
    ifNonNull(lastName, lastName -> query.where(lastName.conditions(CUSTOMERS
.LAST_NAME)));
    return query; ①
}
```

① If the filter parameter is provided, we apply the corresponding condition on the query, using JooQ `where` method.

The actual conditions being provided by our generic `StringPredicate` class.

StringPredicate.java

```
public List<Condition> conditions(TableField field) {
    return Stream.of(
                    mapIfNonNull(is, is -> field.eq(is)),
                    mapIfNonNull(isNot, isNot -> field.ne(isNot)),
                    mapIfNonNull(startsWith, startsWith -> field.startsWith(
startsWith)),
                    mapIfNonNull(endsWith, endsWith -> field.endsWith(endsWith)),
                    mapIfNonNull(contains, contains -> field.contains(contains)),
                    mapIfNonNull(isOneOf, isOneOf -> field.in(isOneOf)))
            .filter(Objects::nonNull)
            .toList();
}
```

Now, we can try our filter using different queries such as :

query

```
query {
  customers(filter: {
    firstName: {isOneOf: ["Dane", "Maura"]}
  }) {
    firstName
    lastName
  }
}
```

Compose data coming from another backend

Now, let's add another attribute to illustrate how easy it is to combine multiple backends and expose their data as a single data model through GraphQL.

We add a `greeting` argument to our customer.

demo.graphqls

```
type Customer {
   greeting: String!
}
```

And we implement it using a `CustomerGreetingController` and our OpenAIService :

CustomerGreetingController.java

```
    @SchemaMapping(typeName = "Customer")
    String greeting(CustomerModel customer, DataFetchingEnvironment env) {
        return openAiService.greeting(customer.firstName());
    }
```

Finally, let's implement the greeting service, calling open AI using langChain4J library.

OpenAiService.java

```
public static final PromptTemplate GREETING_PROMPT_TEMPLATE = PromptTemplate.from(
"write a greeting message for {{it}}. The message must be short, no more than 10
words");

public String greeting(String name) {
    return model.generate(GREETING_PROMPT_TEMPLATE.apply(name).text());
}
```

We can now mix data coming from different backend in a single GraphQL query such this one :

query

```
query {
  customers {
    firstName
    lastName
    greeting
    accounts {
      iban
      balance
      currency
    }
  }
```

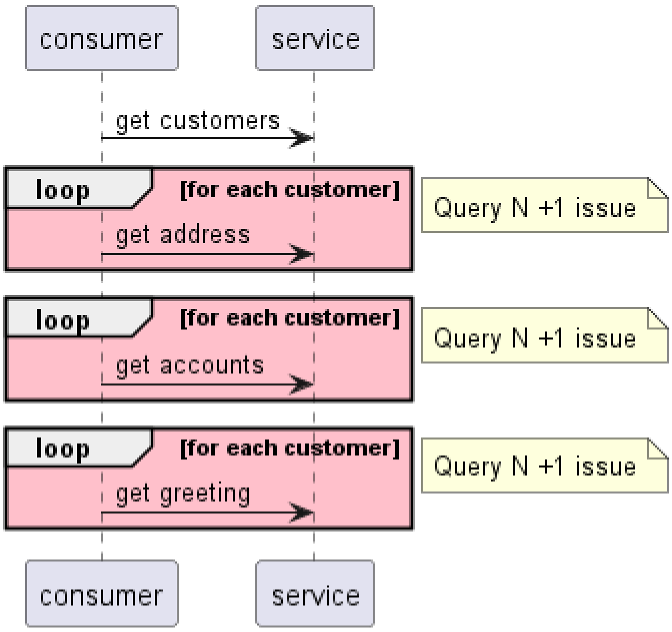
```
}
```

Performance analysis

A typical performance issues when using REST services are :

- **over fetching** : the available service might return some data that the consumer doesn't need
- **under fetching** : the available service might not return all the required data, implying that the consumer need to call other services to get the required data.
- the **query N + 1 issue** : is actually a consequence of under-fetching leading to the explosion of the number of service calls.

For example, in our demo, if a consumer wants to get the customers with their address, accounts and greeting message, the sequence of calls will often be something like :

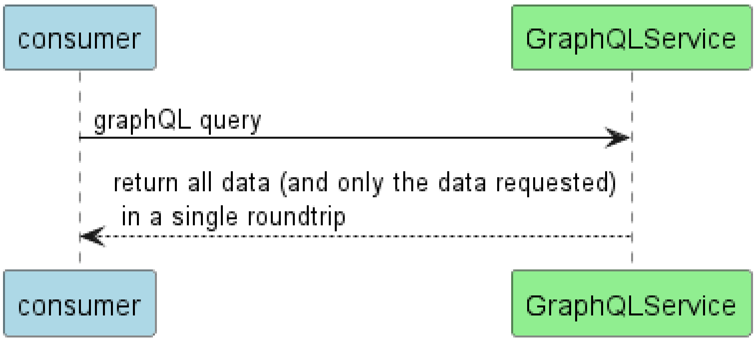


This could lead to :

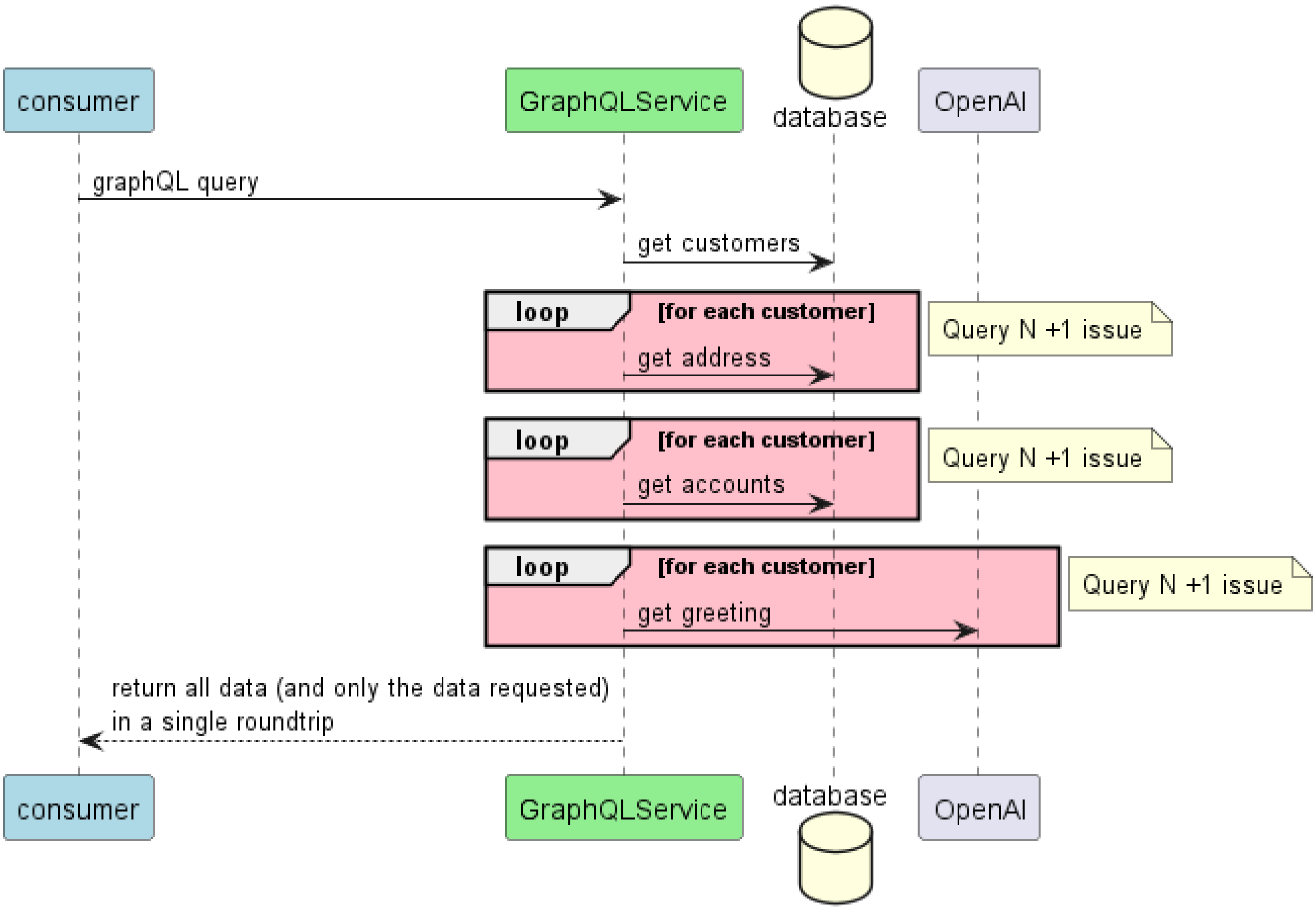
- 1 Query to get "N" customers
- N Queries to get each customer's address
- N Queries to get each customer's accounts
- N Queries to get each customer's greeting message

GraphQL allow to get all the required data in a single roundtrip, which solves these issues, as the consumer get

- only the required data,
- all the required data,
- in a single roundtrip



However, the query N + 1 issue did not really disappear, as it is still present in the backend access to the DB or other backends used to get the data.



So, let's see how we can address this and improve the GraphQL performance.

Performance optimisations using JooQ

JooQ is a perfect match for GraphQL, as it allows to build SQL queries dynamically, based on the GraphQL query. This will be very useful to optimise our service.

Optimising the address query

Let's start by optimising the address query. Instead of querying the address table for each customer, we can use an SQL join to get all the addresses in a single query.

CustomerController.java

```
@QueryMapping
List<CustomerModel> customers(@Argument CustomerPredicate filter,
DataFetchingFieldSelectionSet selectionSet) { ①
    SelectSelectStep<Record> select = dslContext.select(CUSTOMERS.asterisk());
    if (selectionSet.contains("address")) { ②
        select.select(ADDRESSES.asterisk());
    }
    SelectJoinStep<Record> query = select.from(CUSTOMERS);
    if (selectionSet.contains("address")) { ③
        query.leftJoin(ADDRESSES).on(ADDRESSES.ID.eq(CUSTOMERS.ADDRESS_ID));
    }
    if (filter != null) {
        filter.applyOn(query);
    }
    return query
            .fetch()
            .stream()
            .map(CustomerModelMapper::mapCustomerRecordToModel)
            .toList();
}
```

① We can inject the `DataFetchingFieldSelectionSet` in the method, to get the list of fields requested in the GraphQL query.

② If the `address` field is requested, we add the address fields to the select clause of the query.

③ If the `address` field is requested, we add a join to the address table.

We can now test our query again, and we see the executed query is using a join if the `address` field are requested (see the JooQ Logs to see the details of the executed SQL queries).

We still need to modify the Model and the mapping to take into account the new address fields.

CustomerModel.java

```
public record CustomerModel(
        String id,
        String firstName,
        String lastName,
        AddressModel address) { ①
}
```

① The `address` attribute is now an `AddressModel` so it will be mapped to the address fields. We do not need anymore to keep the `addressId` as it was only used to create the SQL query to get the address later on.

CustomerModelMapper.java

```
    public static CustomerModel mapCustomerRecordToModel(Record record) {
        if (record == null) {
```

```java
            return null;
        }

        AddressModel addressModel = AddressModelMapper.mapAddressRecordToModel(record
.into(Tables.ADDRESSES));

        return new CustomerModel(
                customerRecord.getId(),
                customerRecord.getFirstName(),
                customerRecord.getLastName(),
                addressModel,
                accounts
        );
    }
```

① We map the address fields to an `AddressModel` using the `AddressModelMapper`.

② We pass the `AddressModel` to the `CustomerModel` constructor.

We can now test our query again, to see if the address data is returned.

But we get an error, because when there is no address, the address fields returned by the query are `null`, and we defined them as non-null in the GraphQL schema.

Let's fix this.

AddressModelMapper.java

```java
    public static AddressModel mapAddressRecordToModel(Record record) {
        if (record == null) {
            return null;
        }
        AddressesRecord addressRecord = record.into(ADDRESSES);
        if (addressRecord.getId() == null) { ①
            return null;
        }
        return new AddressModel(
                addressRecord.getStreetNumber(),
                addressRecord.getStreetName(),
                addressRecord.getZipCode(),
                addressRecord.getCity(),
                addressRecord.getCountry()
        );
    }
```

① We check if the address id is null, and if it is the case, we return null.

We can now test our query again, and we see the address data is returned, and we execute only one SQL query.

Optimising the accounts query

Now, let's optimise the accounts query. But here, it is a little different as the relation is one-to-many, so we cannot use a join like for the address.

JooQ has a solution for this, using the `multiset` feature.

CustomerController.java

```
@QueryMapping
List<CustomerModel> customers(@Argument CustomerPredicate filter,
DataFetchingFieldSelectionSet selectionSet) {
    SelectSelectStep<Record> select = dslContext.select(CUSTOMERS.asterisk());
    if (selectionSet.contains("address")) {
        select = select.select(ADDRESSES.asterisk());
    }
    if (selectionSet.contains("accounts")) { ①
        select = select.select(
                DSL.multiset( ②
                        dslContext
                                .selectFrom(ACCOUNTS)
                                .where(ACCOUNTS.CUSTOMER_ID.eq(CUSTOMERS.ID))) ③
                        .as("Accounts_Multiset")); ④
    }
    ...
}
```

① If the `accounts` field is requested, we add a *multiset* field to the select clause of the query.

② The `multiset` feature of JooQ allows to create a sub-query that will return a list of accounts for each customer and store it in the result as a single (json) field

③ We create the sub-query, using the `dslContext`.

④ We give a name to the multiset field, that will be used to map the result to a list of `AccountModel`.

If we run our query and look at the logs, we see that JooQ format the list of accounts as a JSON array.

log

```
+----+----------+----------+-----------
+--------------------------------------------------+
|id  |first_name|address_id|last_name  |Accounts_Multiset
|
+----+----------+----------+-----------
+--------------------------------------------------+
|0   |Season    |0         |Kling      |[(0, MR4397602666149964536457958, -39681.0,
KID...|
|1   |Maura     |0         |Marks      |[(3, KW52VCGU0Cp7MFCZ0Di10PUGgTfUCp, 421570.0,
...|
|2   |Perry     |{null}    |Cruickshank|[(4, FR2497686504425I144umb2YJ34, 442239.0,
MRU...|
```

```
|3   |Dane      |{null}    |Jaskolski  |[(6, SV42IOXV33200864928287361387, 624348.0,
PY...|
|4   |Charisse  |2         |Braun      |[(7, IQ39INEQ957796527837825, 753317.0, SLL,
4)]  |
+----+----------+----------+-----------
+----------------------------------------------------+
```

We can now adapt the `CustomerModel` to include the list of accounts.

CustomerModel.java

```java
public record CustomerModel(
        String id,
        String firstName,
        String lastName,
        AddressModel address,
        List<AccountModel> accounts) ①
```

① We add the `accounts` attribute to the `CustomerModel`.

And we can now adapt the mapping to take into account the new accounts fields.

CustomerModelMapper.java

```java
    public static CustomerModel mapCustomerRecordToModel(Record record) {
        if (record == null) {
            return null;
        }

        AddressModel addressModel = AddressModelMapper.mapAddressRecordToModel(record
.into(Tables.ADDRESSES));

        List<AccountModel> accounts = Collections.emptyList();
        if (record.field("Accounts_Multiset") != null) {
            accounts = AccountModelMapper.mapAccountRecordToModel((Result<Record>)
record.get("Accounts_Multiset")); ①
        }

        return new CustomerModel(
                record.get(CUSTOMERS.ID),
                record.get(CUSTOMERS.FIRST_NAME),
                record.get(CUSTOMERS.LAST_NAME),
                addressModel,
                accounts ②
        );
    }
```

① If there is a sub-query result, we get it by its field name, cast it to a `Result<Record>` and pass it to the `AccountModelMapper` to get the list of accounts.

② We pass the accounts to the CustomerModel constructor.

AccountModelMapper.java

```
public static List<AccountModel> mapAccountRecordToModel(Result<Record>
accountsMultiset) {
    return accountsMultiset.stream() ①
            .map(AccountModelMapper::mapAccountRecordToModel) ②
            .toList();
}
```

① We stream the result of the sub-query

② And we map each AccountRecord to an AccountModel.

We can now test our query again, and we see the accounts data is returned, and we still execute only one SQL query !

Some refactoring

To get cleaner code, we can now refactor the code a little bit.

Let’s come back to our CustomerController, and extract the query creation in a separate methods to make the code more readable.

filename.java

```
    @QueryMapping
    List<CustomerModel> customers(@Argument CustomerPredicate filter,
DataFetchingFieldSelectionSet selectionSet) {
        SelectSelectStep<Record> select = selectStep(selectionSet); ①
        SelectJoinStep<Record> query = joinStep(selectionSet, select); ②
        query = whereStep(filter, query); ③
        return executeAndMap(query); ④
    }
```

① Select the fields to be returned in the query, depending on the selection set

② Create the join step, depending on the selection set

③ Apply the filter, if any

④ Fetch the results and map them to the CustomerModel

Optimising the greeting query using Dataloaders

Now, let’s optimise the greeting query.

As we have seen, the greeting query is calling the OpenAI service for each customer, which is not optimal. The idea here would be to ***batch*** the calls to the OpenAI service, to get the greeting messages for all the customers in a single call.

log

```
Call openAI with name: Season
Call openAI with name: Maura
Call openAI with name: Dane ①
Call openAI with name: Charisse
Call openAI with name: Rocco
Call openAI with name: Leon
Call openAI with name: Hugh
Call openAI with name: Frances
Call openAI with name: Lawana
Call openAI with name: Dane ①
```

① Also note here that there can be multiple calls with the same input, which is usually also a potential performance issue.

For this, we will use the Dataloader mechanism, that is a common pattern in GraphQL.

So let's change the @SchemaMapping for the greeting attribute :

CustomerGreetingController.java

```
@SchemaMapping(typeName = "Customer")
CompletableFuture<String> greeting(CustomerModel customer, DataFetchingEnvironment
env) { ①
    DataLoader<String, String> dataLoader = env.getDataLoader(GREETING_DATALOADER); ②
    return dataLoader.load(customer.firstName()); ③
}
```

① The mapping returns a `CompletableFuture<String>`. When this mapping will be called, it doesn't have to answer immediately. Instead it will delegate to an asynchronous dataloader.

② We get the dataloader from the `DataFetchingEnvironment` using its name.

③ And we call the dataloader to get a completable future of the greeting for the current customer.

Now, let's create the dataloader.

CustomerGreetingDataloader.java

```
@Service
@RequiredArgsConstructor
public class CustomerGreetingDataloader {
    public static final String GREETING_DATALOADER = "GREETING_DATALOADER";

    private final OpenAiService openAiService;
    private final BatchLoaderRegistry batchLoaderRegistry;

    @PostConstruct
    void initDataLoader() { ①
        batchLoaderRegistry.forTypePair(String.class, String.class)
                .withName(GREETING_DATALOADER) ②
```

```
                .withOptions(newOptions()
                        .setMaxBatchSize(5) ③
                )
                .registerBatchLoader((List<String> names, BatchLoaderEnvironment env)
                        -> generateGreetings(names)); ④
    }

    private Flux<String> generateGreetings(List<String> names) { ⑤
        CompletableFuture<List<String>> futureList =
                supplyAsync(() -> openAiService.greetings(names), ⑥
                        newVirtualThreadPerTaskExecutor()); ⑦
        return mapToFlux(futureList); ⑧
    }
```

① At startup, we initialize the dataloader, by registering it in the `batchLoaderRegistry`.

② We give it a name

③ And we set the maximum batch size, to limit the number data that will be processed by the dataloader in a single call.

④ We provide a function that will be called by the dataloader when it is triggered.

⑤ First note that the function returns a `Flux<String>` and not a list of Strings (greetings). The GraphQL dataloaders, in addition to process data by "batch", also process data asynchronously, and the Spring layer uses the reactive `Flux` and `Mono` for this.

⑥ We use a `CompletableFuture` to call the OpenAI service asynchronously, giving a list of names as input.

⑦ Also note here that we use a `VirtualThreadPerTaskExecutor` to benefit from Java 21 Virtual threads.

⑧ Finally, we convert the `CompletableFuture` to a `Flux`.

We can now test our query again, and we see the greeting data is returned, and we execute only two calls to the OpenAI service.

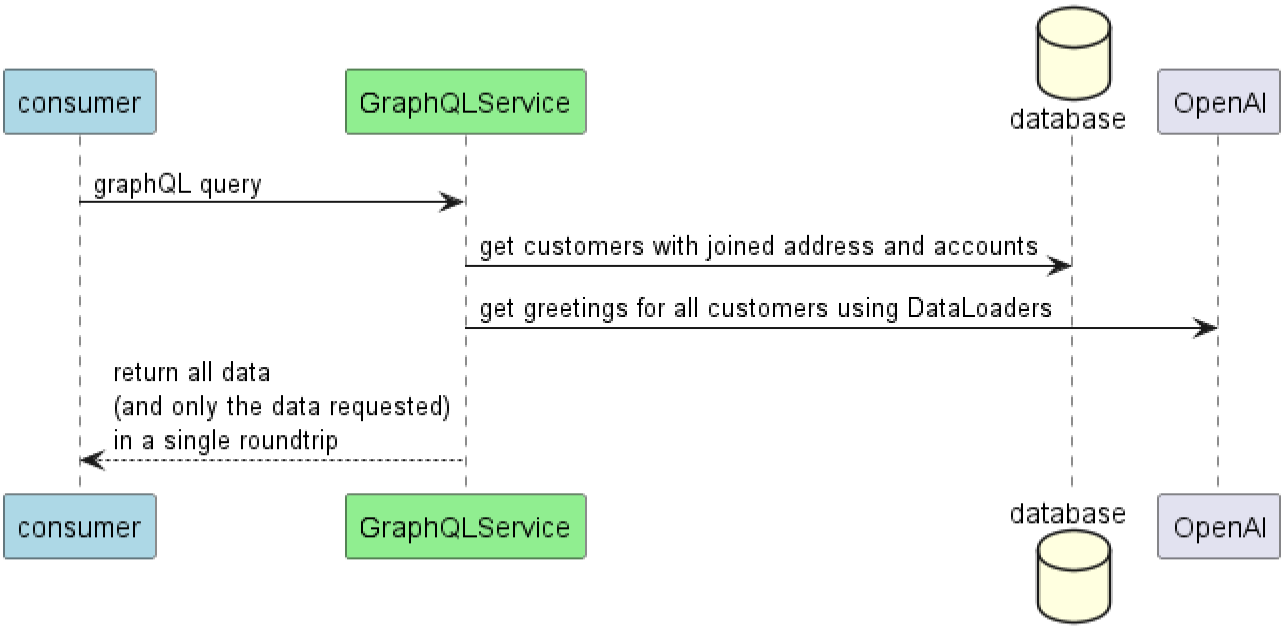
log

```
Call openAI with names: [Season, Maura, Dane, Charisse, Rocco] on thread
VirtualThread[#66]/runnable@ForkJoinPool-1-worker-1 ①
Call openAI with names: [Leon, Hugh, Frances, Lawana] on thread
VirtualThread[#72]/runnable@ForkJoinPool-1-worker-2 ①
```

① We have a batch limit of 5, and we have 10 customers in DB. So the first call contains 5 names.

② Note that the second call contains only 4 names. This is because the dataloader, by default, will *cache* the values, it does not call the API twice with the same input ("Dane" in our case).

After all these optimisations we successfully optimised the performance of our GraphQL service:



Conclusions

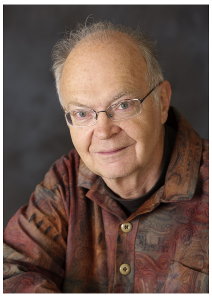
GraphQL and Spring for GraphqQL offer a very **flexible** approach to implement data access services.

Optimising such a service **performance** can be done using the **Dataloader** mechanism, allowing to address the Query N + 1 issue :

- It "**Batches**" backend calls
- It "**Caches**" backend calls for same input
- It executes **Asynchronously** allowing parallel calls and the use Java 21 Virtual Threads.

JooQ Dynamic queries are a perfect match for GraphQL implementation! It allows to

- **Join** tables when needed
- Select a "tree of date" using the **Multiset** feature
- Select **only the required fields**



> We *should* forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**.
>
> Yet we *should not pass up our opportunities in that critical 3%*.
>
> — D. Knuth

As suggested by D. Knuth, we should not always try to optimise the code from the start as it complexifies it. However, there is a good opportunity of performance improvement applying the strategies explained in this demo.