

## Piotr's TechBlog

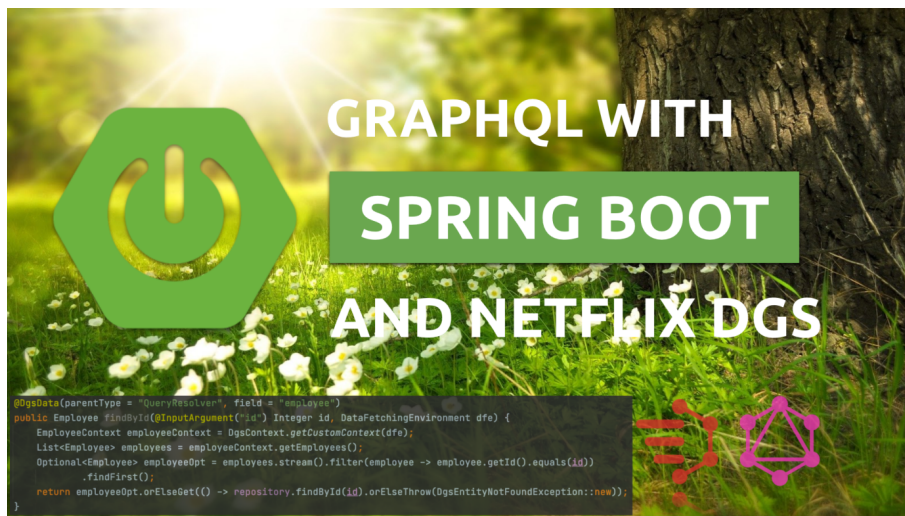
Java, Spring, Kotlin, microservices, Kubernetes, containers

# An Advanced GraphQL with Spring Boot and Netflix DGS

Spring Boot

## An Advanced GraphQL with Spring Boot and Netflix DGS

By piotr.minkowski • April 8, 2021 • 20




In this article, you will learn how to use the Netflix DGS library to simplify GraphQL development with Spring Boot. We will discuss more advanced topics related to GraphQL and databases, like filtering or relationship fetching. I published a similar article some months ago: [An Advanced Guide to GraphQL with Spring Boot](#). However, it is based on a different library called GraphQL Java Kickstart (<https://github.com/graphql-java-kickstart/graphql-spring-boot>). Since Netflix DGS has been released some months ago, you might want to take a look at it. So, that's what we will do now.

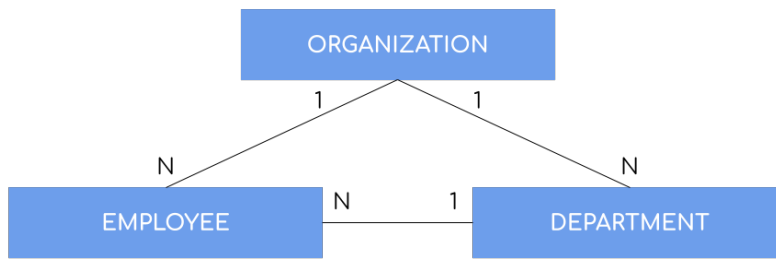
**Netflix DGS** is an annotation-based GraphQL Java library built on top of Spring Boot. Consequently, it is dedicated to Spring Boot applications. Besides the annotation-based programming model, it provides several useful features. Netflix DGS allows generating source code from GraphQL schemas. It simplifies writing unit tests and also supports websockets, file uploads, or GraphQL federation. In order to show you the differences between this library and the previously described Kickstart library, I'll use the same Spring Boot application as before. Let me just briefly describe our scenario.

## Source Code

If you would like to try it by yourself, you may always take a look at my source code. In order to do that you need to clone my GitHub [repository](#). Then you should just follow my instructions.

First, you should go to the `sample-app-netflix-dgs` directory. The example with GraphQL Java Kickstart is available inside the `sample-app-kickstart` directory.

As I mentioned before, we use the same schema and entity model as before. I created an application that exposes API using GraphQL and connects to H2 in-memory database. We will discuss Spring Boot GraphQL JPA support. For integration with the H2 database, I'm using Spring Data JPA and Hibernate. I have implemented three entities `Employee`, `Department` and `Organization` – each of them stored in the separated table. A relationship model between them is visualized in the picture below. 



## 1. Dependencies for Spring Boot and Netflix GraphQL

Let's start with dependencies. We need to include Spring Web, Spring Data JPA, and the `com.database:h2` artifact for running an in-memory database with our application. Of course, we also have to include Netflix DGS Spring Boot Starter. Here's a list of required dependencies in Maven `pom.xml`.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
</dependency>
<dependency>
  <groupId>com.netflix.graphql.dgs</groupId>
  <artifactId>graphql-dgs-spring-boot-starter</artifactId>
  <version>${netflix-dgs.spring.version}</version>
</dependency>
```

## 2. GraphQL schemas

Before we start implementation, we need to create GraphQL schemas with objects, queries, and mutations. A schema may be defined in multiple `graphqls` files, but all of them have to be placed inside the `/src/main/resources/schemas` directory. Thanks to that, the Netflix DGS library detects and loads them automatically.

GraphQL schema for each entity is located in the separated file. Let's take a look at the `department.graphqls` file. There is the `QueryResolver` with two find methods and the `MutationResolver` with a single method for adding new departments. We also have an input object for mutation and a standard type definition for queries.

















```
type QueryResolver {
  departments: [Department]
  department(id: ID!): Department!
}

type MutationResolver {
  newDepartment(department: DepartmentInput!): Department
}

input DepartmentInput {
  name: String!
  organizationId: Int
}

type Department {
  id: ID!
  name: String!
  organization: Organization
  employees: [Employee]
}
```


Then we may take a look at the `organization.graphqls` file. It is a little bit more complicated than the previous schema. As you see I'm using the keyword `extend` on `QueryResolver` and `MutationResolver`. That's because we have several files with GraphQL schemas.

```
extend type QueryResolver {
  organizations: [Organization]
  organization(id: ID!): Organization!
}

extend type MutationResolver {
  newOrganization(organization: OrganizationInput!): Organization
}

input OrganizationInput {
  name: String!
}

type Organization {
  id: ID!
  name: String!
  employees: [Employee]
  departments: [Department]
}
```

Finally, the schema for the `Employee` entity. In contrast to the previous schemas, it has objects responsible for filtering like `EmployeeFilter`. We also need to define the  `a` object with `mutation` and `query`.

```
extend type QueryResolver {
  employees: [Employee]
  employeesWithFilter(filter: EmployeeFilter): [Employee]
  employee(id: ID!): Employee!
}

extend type MutationResolver {
  newEmployee(employee: EmployeeInput!): Employee
}

input EmployeeInput {
  firstName: String!
  lastName: String!
  position: String!
  salary: Int
  age: Int
  organizationId: Int!
  departmentId: Int!
}

type Employee {
  id: ID!
  firstName: String!
  lastName: String!
  position: String!
  salary: Int
  age: Int
  department: Department
  organization: Organization
}

input EmployeeFilter {
  salary: FilterField
  age: FilterField
  position: FilterField
}

input FilterField {
  operator: String!
  value: String!
}

schema {
  query: QueryResolver
  mutation: MutationResolver
}
```

### 3. Domain Model for GraphQL and Hibernate

We could have generated Java source code using previously defined GraphQL schemas. However, I prefer to use Lombok annotations, so I will do it manually. Here's the **Employee** entity corresponding to the **Employee** object defined in GraphQL schema.



```
@Entity
@Data
@NoArgsConstructor
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class Employee {
    @Id
    @GeneratedValue
    @EqualsAndHashCode.Include
    private Integer id;
    private String firstName;
    private String lastName;
    private String position;
    private int salary;
    private int age;
    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;
    @ManyToOne(fetch = FetchType.LAZY)
    private Organization organization;
}
```

Also, let's take a look at the **Department** entity.

```
@Entity
@Data
@NoArgsConstructor
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class Department {
    @Id
    @GeneratedValue
    @EqualsAndHashCode.Include
    private Integer id;
    private String name;
    @OneToMany(mappedBy = "department")
    private Set<Employee> employees;
    @ManyToOne(fetch = FetchType.LAZY)
    private Organization organization;
}
```

The input objects are much simpler. Just to compare, here's the **DepartmentInput** class.

```
@Data
@NoArgsConstructor
public class DepartmentInput {
    private String name;
    private Integer organizationId;
}
```

## 4. Using Netflix DGS with Spring Boot

Netflix DGS provides annotation-based support for Spring Boot. Let's analyze the most interesting features using the example implementation of a query resolver. The **EmployeeFetcher** is responsible for defining queries related to the **Employee** object. We should annotate such a class with **@DgsComponent** (1). We may create our custom context definition to pass data between different methods or even different query resolvers (2). Then, we have to annotate every query method with **@DgsData** (3). The fields **parentType** and **fields** should match the names declared in GraphQL schemas. We defined three queries in the **employee.graphqls** file, so we have three methods inside **EmployeeFetcher**. After fetching all employees, we may save them in our custom context object (4), and then reuse them in other methods or resolvers (5).

The last query method **findWithFilter** performs advanced filtering based on the dynamic list of fields passed in the input (6). To pass an input parameter we should annotate the method argument with **@InputArgument**.



```

@dgsComponent // (1)
public class EmployeeFetcher {

    private EmployeeRepository repository;
    private EmployeeContextBuilder contextBuilder; // (2)

    public EmployeeFetcher(EmployeeRepository repository,
        EmployeeContextBuilder contextBuilder) {
        this.repository = repository;
        this.contextBuilder = contextBuilder;
    }

    @DgsData(parentType = "QueryResolver", field = "employees") // (3)
    public List<Employee> findAll() {
        List<Employee> employees = (List<Employee>) repository.findAll();
        contextBuilder.withEmployees(employees).build(); // (4)
        return employees;
    }

    @DgsData(parentType = "QueryResolver", field = "employee")
    public Employee findById(@InputArgument("id") Integer id,
        DataFetchingEnvironment dfe) {
        EmployeeContext employeeContext = DgsContext.getCustomContext(dfe); // (5)
        List<Employee> employees = employeeContext.getEmployees();
        Optional<Employee> employeeOpt = employees.stream()
            .filter(employee -> employee.getId().equals(id)).findFirst();
        return employeeOpt.orElseGet(() ->
            repository.findById(id)
                .orElseThrow(DgsEntityNotFoundException::new));
    }

    @DgsData(parentType = "QueryResolver", field = "employeesWithFilter")
    public Iterable<Employee> findWithFilter(@InputArgument("filter") EmployeeFilter filter) { // (6)
        Specification<Employee> spec = null;
        if (filter.getSalary() != null)
            spec = bySalary(filter.getSalary());
        if (filter.getAge() != null)
            spec = (spec == null ? byAge(filter.getAge()) : spec.and(byAge(filter.getAge())));
        if (filter.getPosition() != null)
            spec = (spec == null ? byPosition(filter.getPosition()) :
                spec.and(byPosition(filter.getPosition())));
        if (spec != null)
            return repository.findAll(spec);
        else
            return repository.findAll();
    }

    private Specification<Employee> bySalary(FilterField filterField) {
        return (root, query, builder) ->
            filterField.generateCriteria(builder, root.get("salary"));
    }

    private Specification<Employee> byAge(FilterField filterField) {
        return (root, query, builder) ->
            filterField.generateCriteria(builder, root.get("age"));
    }

    private Specification<Employee> byPosition(FilterField filterField) {
        return (root, query, builder) ->
            filterField.generateCriteria(builder, root.get("position"));
    }
}

```

Then, we may switch to the `DepartmentFetcher` class. It shows the example of relationship fetching. We use `DataFetchingEnvironment` to detect if the input query contains a relationship field (1). In our case, it may be `employees` or `organization`. If any of those fields is defined we add the relation to the JOIN statement (2). We implement the same

approach for both `findById` (3) and `findAll` methods. However, the `findById` method also uses data stored in the custom context represented by the `EmployeeContext` bean (4). If the method `findAll` in `EmployeeFetcher` has already been invoked, we can fetch employees assigned to the particular department from the context instead of including the relation to the JOIN statement (5).



```
@DgsComponent
public class DepartmentFetcher {

    private DepartmentRepository repository;

    DepartmentFetcher(DepartmentRepository repository) {
        this.repository = repository;
    }

    @DgsData(parentType = "QueryResolver", field = "departments")
    public Iterable<Department> findAll(DataFetchingEnvironment environment) {
        DataFetchingFieldSelectionSet s = environment.getSelectionSet(); // (1)
        List<Specification<Department>> specifications = new ArrayList<>();
        if (s.contains("employees") && !s.contains("organization")) // (2)
            return repository.findAll(fetchEmployees());
        else if (!s.contains("employees") && s.contains("organization"))
            return repository.findAll(fetchOrganization());
        else if (s.contains("employees") && s.contains("organization"))
            return repository.findAll(fetchEmployees().and(fetchOrganization()));
        else
            return repository.findAll();
    }

    @DgsData(parentType = "QueryResolver", field = "department")
    public Department findById(@InputArgument("id") Integer id,
        DataFetchingEnvironment environment) { // (3)
        Specification<Department> spec = byId(id);
        DataFetchingFieldSelectionSet selectionSet = environment.getSelectionSet();
        EmployeeContext employeeContext = DgsContext.getCustomContext(environment); // (4)
        Set<Employee> employees = null;
        if (selectionSet.contains("employees")) {
            if (employeeContext.getEmployees().size() == 0) // (5)
                spec = spec.and(fetchEmployees());
            else
                employees = employeeContext.getEmployees().stream()
                    .filter(emp -> emp.getDepartment().getId().equals(id))
                    .collect(Collectors.toSet());
        }
        if (selectionSet.contains("organization"))
            spec = spec.and(fetchOrganization());
        Department department = repository
            .findOne(spec).orElseThrow(DgsEntityNotFoundException::new);
        if (employees != null)
            department.setEmployees(employees);
        return department;
    }

    private Specification<Department> fetchOrganization() {
        return (root, query, builder) -> {
            Fetch<Department, Organization> f = root.fetch("organization", JoinType.LEFT);
            Join<Department, Organization> join = (Join<Department, Organization>) f;
            return join.getOn();
        };
    }

    private Specification<Department> fetchEmployees() {
        return (root, query, builder) -> {
            Fetch<Department, Employee> f = root.fetch("employees", JoinType.LEFT);
            Join<Department, Employee> join = (Join<Department, Employee>) f;
            return join.getOn();
        };
    }

    private Specification<Department> byId(Integer id) {
        return (root, query, builder) -> builder.equal(root.get("id"), id);
    }
}
```



```
}

```

In comparison to the data fetchers implementation of mutation handlers is rather simple. We just need to define a single method for adding new entities. Here's the implementation of `DepartmentMutation`.

```
@DgsComponent
public class DepartmentMutation {

    private DepartmentRepository departmentRepository;
    private OrganizationRepository organizationRepository;

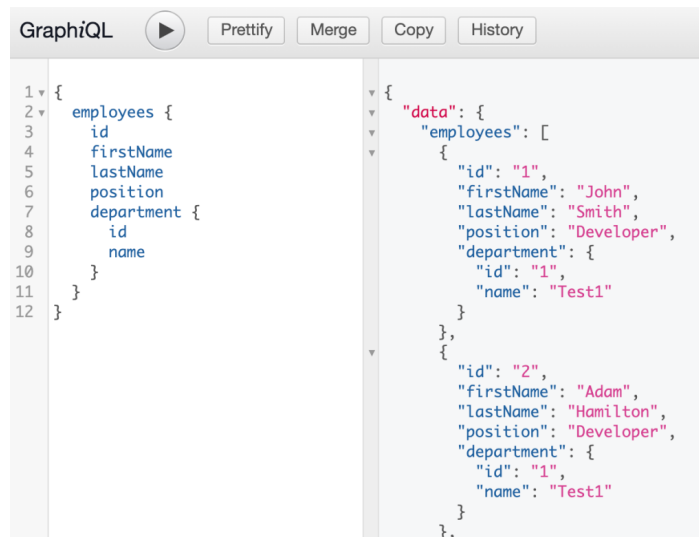
    DepartmentMutation(DepartmentRepository departmentRepository,
        OrganizationRepository organizationRepository) {
        this.departmentRepository = departmentRepository;
        this.organizationRepository = organizationRepository;
    }

    @DgsData(parentType = "MutationResolver", field = "newDepartment")
    public Department newDepartment(DepartmentInput input) {
        Organization organization = organizationRepository
            .findById(input.getOrganizationId())
            .orElseThrow();
        return departmentRepository
            .save(new Department(null, input.getName(), null, organization));
    }
}
```

## 5. Running Spring Boot application and testing Netflix GraphQL support

The last step in our exercise is to run and test the Spring Boot application. It inserts some test data to the H2 database on startup. So, let's just use the GraphQL tool to run test queries. It is automatically included in the application by the Netflix DGS library. We may display it by invoking the URL <http://localhost:8080/graphql>.

In the first step, we run the GraphQL query responsible for fetching all employees with departments. The method that handles the query also builds a custom context and stores there all existing employees.



Then, we may run a query responsible for finding a single department by its `id`. We will fetch both relations one-to-many with `Employee` and many-to-one with `Organization`.



The screenshot shows the GraphiQL IDE interface. On the left, a GraphQL query is entered:
 

```

1 {
2   department(id: 1) {
3     id
4     name
5     employees {
6       id
7       firstName
8       lastName
9     }
10    organization {
11      id
12    }
13  }
14 }
15
  
```

 On the right, the JSON response is displayed:
 

```

{
  "data": {
    "department": {
      "id": "1",
      "name": "Test1",
      "employees": [
        {
          "id": "1",
          "firstName": "John",
          "lastName": "Smith"
        },
        {
          "id": "2",
          "firstName": "Adam",
          "lastName": "Hamilton"
        },
        {
          "id": "3",
          "firstName": "Tracy",
          "lastName": "Smith"
        }
      ],
      "organization": {
        "id": "1"
      }
    }
  }
}
  
```

 The interface includes buttons for 'Prettify', 'Merge', 'Copy', and 'History' at the top.

While the **Organization** entity is fetched using the JOIN statement, **Employee** is taken from the context. Here's the SQL query generated for our current scenario.

```

Hibernate:
select
    department0_.id as id1_0_0_,
    organizati1_.id as id1_2_1_,
    department0_.name as name2_0_0_,
    department0_.organization_id as organiza3_0_0_,
    organizati1_.name as name2_2_1_
from
    department department0_
left outer join
    organization organizati1_
        on department0_.organization_id=organizati1_.id
where
    department0_.id=1
  
```

Finally, we can test our filtering feature. Let's filter employees using **salary** and **age** criteria.

The screenshot shows the GraphiQL IDE interface. On the left, a GraphQL query is entered:
 

```

1 {
2   employeesWithFilter(filter: {
3     age: {
4       operator: "gt"
5       value: "30"
6     }
7     salary: {
8       operator: "lt"
9       value: "15000"
10    }
11  }) {
12    id
13    firstName
14    lastName
15    age
16    salary
17  }
18 }
  
```

 On the right, the JSON response is displayed:
 

```

{
  "data": {
    "employeesWithFilter": [
      {
        "id": "2",
        "firstName": "Adam",
        "lastName": "Hamilton",
        "age": 35,
        "salary": 12000
      }
    ]
  }
}
  
```

 The interface includes buttons for 'Prettify', 'Merge', 'Copy', and 'History' at the top.

Let's take a look at the SQL query for the recently called method.





```
Hibernate:
select
    employee0_.id as id1_1_,
    employee0_.age as age2_1_,
    employee0_.department_id as departme7_1_,
    employee0_.first_name as first_na3_1_,
    employee0_.last_name as last_nam4_1_,
    employee0_.organization_id as organiza8_1_,
    employee0_.position as position5_1_,
    employee0_.salary as salary6_1_
from
    employee employee0_
where
    employee0_.age>30
    and employee0_.salary<15000
```

## Final Thoughts

Netflix DGS seems to be an interesting alternative to other libraries that provide support for GraphQL with Spring Boot. It has been open-sourced some weeks ago, but it is rather a stable solution. I guess that before releasing it publicly, the Netflix team has tested it in the battle. I like its annotation-based programming style and several other features. This article will help you in starting with Netflix DGS.

[graphql](#)[graphql](#)[H2](#)[Hibernate](#)[JPA](#)[netflix dgs](#)[Spring Boot](#)[Spring Data](#)

Written by:

**piotr.minkowski**

[View All Posts](#) →

## 20 COMMENTS



**RSWRC** April 8, 2021 12:38 pm

public List findAll(DataFetchingEnvironment dfe) - here the dfe is not used inside. Is it required to be passed there?

Loading...

[REPLY](#)



**piotr.minkowski** April 9, 2021 9:32 am

Of course no, thank you for paying attention. I used that before and forgotten to remove it. Now it's ok.

Loading...

[REPLY](#)



**code22-bit** April 22, 2021 2:32 pm

@piotr.minkowski great work ! it's possible to implement this for mongo database?

Loading...

[REPLY](#)





**piotr.minkowski** May 23, 2021 7:19 am

Thanks. Of course it is possible with Spring Data Mongo support

Loading...

[REPLY](#)



**Shahed** May 29, 2021 6:42 am

Hi, thank you for a well-written blog. I've read your other pieces on graphql-java-kickstart, but for someone starting with microservices with Spring Boot, how do you propose we add a config server and service discovery? Is there any difference (from a regular webservice based microservice) in a graphql based microservice?

Loading...

[REPLY](#)



**piotr.minkowski** June 1, 2021 8:23 am

Hello,

I have already described a microservices-based approach with GraphQL in one of my previous articles: <https://piotrminkowski.com/2018/08/16/graphql-the-future-of-microservices/>. Hope it helps.

Loading...

[REPLY](#)



**srinivasallu** July 8, 2021 9:21 am

Hi,

How to handle the custom exceptions in dgs framework?

Loading...

[REPLY](#)



**piotr.minkowski** January 18, 2023 11:19 pm

Hi, Here's the guide: <https://netflix.github.io/dgs/error-handling/>

Loading...

[REPLY](#)



**Ravi** July 13, 2021 12:47 pm

Nice blog. I tried creating a graphql java client with dgs and I am getting error: There is no scalar implementation for the named 'Long' scalar type. Let me know if you can help, thanks.

Loading...

[REPLY](#)



**piotr.minkowski** January 18, 2023 11:18 pm

I added some tests with client. Now everything should work fine

Loading...

[REPLY](#)



**Arulmurugan** August 21, 2021 6:58 pm

Hi, Thanks for your article. When i query employees with department, for every employee there is department query executed, is there any optimal way to reduce db round calls?

```
query {  
  employees {
```



```

id,
firstName,
department {
  id,
  name
}
}
}

```

```

Hibernate:
select
employee0_.id as id1_1_,
employee0_.age as age2_1_,
employee0_.department_id as departme7_1_,
employee0_.first_name as first_na3_1_,
employee0_.last_name as last_nam4_1_,

```

## Search

Search

```

employee0_.salary as salary6_1_
from
employee employee0_

```

## Follow Blog via Email

```

Hibernate:
select

```

Enter your email address to follow this blog and receive notifications of new posts by email.

```

department0_ organization_id as organiza3_0_0_

```

Enter your email address

```

department department0_

```

Follow

```

where
department0_.id=?

```

```

Hibernate:
select

```

```

department0_.id as id1_0_0_,
department0_.name as name2_0_0_,
department0_.organization_id as organiza3_0_0_
from

```

## Categories

Cloud (28)

```

department department0_

```

```

where

```

Containers (59)

```

department0_.id=?

```

```

Hibernate:

```

Continuous Integration (25)

```

department0_.id as id1_0_0_,

```

Data Grids (47)

```

department0_.name as name2_0_0_,

```

```

department0_.organization_id as organiza3_0_0_

```

Kotlin (9)

```

from

```

```

department department0_

```

Kubernetes (70)

```

department0_.id=?

```

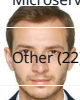
Message Brokers (23)

Loading...

Micronaut (12)

[REPLY](#)

Microservices (60)



**piotr.minkowski** August 31, 2021 3:34 pm

Other (22)

Hello,

I didn't see this problem during the article write-up. Of course, we just need to tweak Hibernate query used for that, as in the other join examples.

Performance (19)

Loading...

Quarkus (7)

[REPLY](#)

Security (20)

Spring Boot (70)

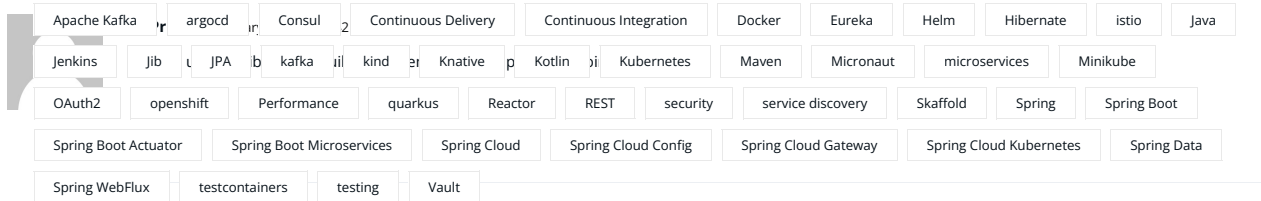
**Dikshyant Acharya** December 21, 2021 12:54 pm

Hello, thanks for this explanation. But can you clarify little bit when we want to send our data not from graphql but from other frontend(from angular), we have

cross origin problem. In rest api, we simply put(@CrossOrigin("url")) but it doesnt seem to work while using dgsframework. Thanks for your time and have a good day. 😊

Loading...

## Tags

[REPLY](#)


**piotr.minkowski** January 13, 2022 12:40 pm

Yes. See <https://netflix.github.io/dgs/advanced/java-client/>

Loading...

[REPLY](#)

## Contact info

If you would like to contact me in order you have any questions, thoughts or ideas (e.g. suggestions for future articles) contact me via email.

📍 Warsaw, Poland

✉ [piotr.minkowski@gmail.com](mailto:piotr.minkowski@gmail.com)

## Top Posts & Pages

- Best Practices for Java Apps on Kubernetes
- Guide to building Spring Boot library
- Kafka Streams with Spring Cloud Stream
- Kafka Transactions with Spring Boot
- Create and Release Your Own Helm Chart

## Social Media

Proudly powered by WordPress | Theme: HoneyWaves by SpiceThemes



actually, I figured it out, thanks!

Loading...

[REPLY](#)

## Leave a Reply

