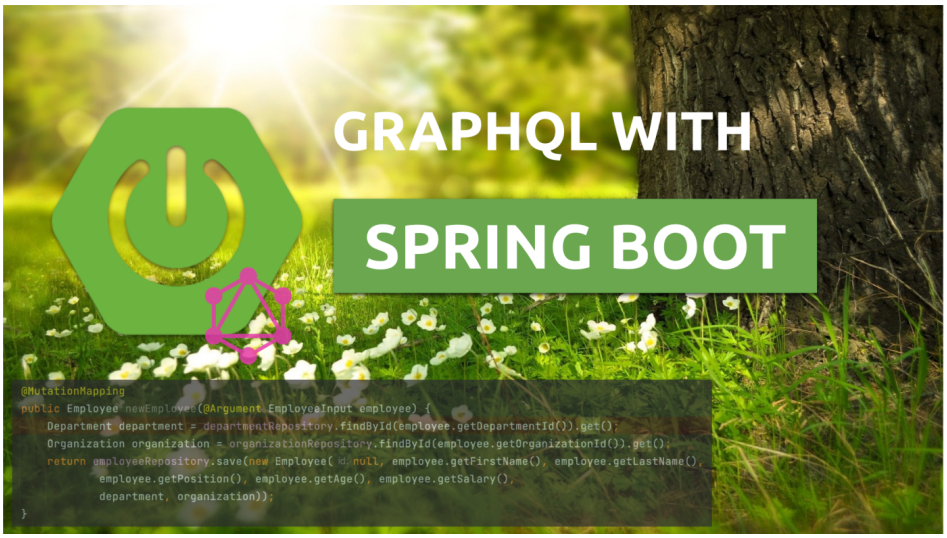
An Advanced GraphQL with Spring Boot

Spring Boot

An Advanced GraphQL with Spring Boot

By piotr.minkowski • January 18, 2023 • 1



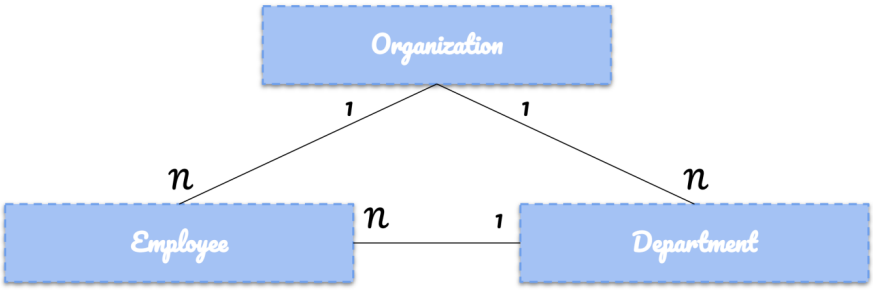
In this article, you will learn how to use Spring for GraphQL in your Spring Boot app. Spring for GraphQL is a relatively new project. The `1.0` version was released a few months ago. Before that release, we had to include third-party libraries to simplify GraphQL implementation in the Spring Boot app. I have already described two alternative solutions in my previous articles. In the following **article**, you will learn about the GraphQL Java Kickstart project. In the other **article**, you can see how to create some more advanced GraphQL queries with the Netflix DGS library.

We will use a very similar schema and entity model as in those two articles about Spring Boot and GraphQL.

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.

Firstly, you should go to the `sample-app-spring-graphql` directory. Our sample Spring Boot exposes API over GraphQL and connects to the in-memory H2 database. It uses Spring Data JPA as a layer to interact with the database. There are three entities `Employee`, `Department` and `Organization`. Each of them is stored in a separate table. Here's a relationship model.



Getting started with Spring for GraphQL

In addition to the standard Spring Boot modules we need to include the following two dependencies:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-graphql</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.graphql</groupId>
  <artifactId>spring-graphql-test</artifactId>
  <scope>test</scope>
</dependency>
```

The `spring-graph-test` provides additional capabilities for building unit tests. The starter comes with required libraries and auto-configuration. However, it does not enable the GraphiQL interface. In order to enable it, we should set the following property in the `application.yml` file:

```
spring:
  graphql:
    graphiql:
      enabled: true
```

By default, Spring for GraphQL tries to load schema files from the `src/main/resources/graphql` directory. It looks there for the files with the `.graphqls` or `.gqls` extensions. Let's GraphQL schema for the `Department` entity. The `Department` type references the two other types: `Organization` and `Employee` (the list of employees). There are two queries for searching all departments and a department by id, and a single mutation for adding a new department.

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

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

input DepartmentInput {
   name: String!
   organizationId: Int
}

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

The `Organization` type schema is pretty similar. From the more advanced stuff, we need to handle joins to the `Employee` and `Department` types.

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

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

input OrganizationInput {
   name: String!
}

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

And the last schema – for the `Employee` type. Unlike the previous schemas, it defines the type responsible for handling filtering. The EmployeeFilter is able to filter by salary, position, or age. There is also the query method for handling filtering – `employeesWithFilter`.

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

extend type Mutation {
  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!
}
```

Create Entities

Do not hold that against me, but I'm using Lombok in entity implementation. Here's the `Employee` entity corresponding to the `Employee` type defined in GraphQL schema.

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class Employee {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @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;
}
```

Here we have the `Department` entity.

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

Finally, we can take a look at the `Organization` entity.

```
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class Organization {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @EqualsAndHashCode.Include
  private Integer id;
  private String name;
  @OneToMany(mappedBy = "organization")
  private Set<Department> departments;
  @OneToMany(mappedBy = "organization")
  private Set<Employee> employees;
}
```

Using GraphQL for Spring with Spring Boot

Spring for GraphQL provides an annotation-based programming model using the well-known `@Controller` pattern. It is also possible to adapt the Querydsl library and use it together with Spring Data JPA. You can then use it in your Spring Data repositories annotated with `@GraphQLRepository`. In this article, I will use the standard JPA Criteria API for generating more advanced queries with filters and joins.

Let's start with our first controller. In comparison to both previous articles about Netflix DGS and GraphQL Java Kickstart, we will keep queries and mutations in the same class. We need to annotate query methods with the `@QueryMapping`, and mutation methods with `@MutationMapping`. The last query method `employeesWithFilter` performs advanced filtering based on the dynamic list of fields passed in the input `EmployeeFilter` type. To pass an input parameter we should annotate the method argument with `@Argument`.

```
@Controller
public class EmployeeController {

   DepartmentRepository departmentRepository;
   EmployeeRepository employeeRepository;
   OrganizationRepository organizationRepository;

   EmployeeController(DepartmentRepository departmentRepository,
                      EmployeeRepository employeeRepository,
                      OrganizationRepository organizationRepository) {
      this.departmentRepository = departmentRepository;
      this.employeeRepository = employeeRepository;
      this.organizationRepository = organizationRepository;
   }

   @QueryMapping
   public Iterable<Employee> employees() {
       return employeeRepository.findAll();
   }

   @QueryMapping
   public Employee employee(@Argument Integer id) {
       return employeeRepository.findById(id).orElseThrow();
   }

   @MutationMapping
   public Employee newEmployee(@Argument EmployeeInput employee) {
      Department department = departmentRepository
         .findById(employee.getDepartmentId()).get();
      Organization organization = organizationRepository
         .findById(employee.getOrganizationId()).get();
      return employeeRepository.save(new Employee(null, employee.getFirstName(), employee.getLastName(),
                employee.getPosition(), employee.getAge(), employee.getSalary(),
                department, organization));
   }

   @QueryMapping
   public Iterable<Employee> employeesWithFilter(
         @Argument EmployeeFilter filter) {
      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 employeeRepository.findAll(spec);
      else
         return employeeRepository.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"));
   }
}
```

Here's our JPA repository implementation. In order to use JPA Criteria API we need it needs to extend the `JpaSpecificationExecutor` interface. The same rule applies to both others `DepartmentRepository` and `OrganizationRepository`.

```
public interface EmployeeRepository extends
   CrudRepository<Employee, Integer>, JpaSpecificationExecutor<Employee> {
}
```

Now, let's switch to another controller. Here's the implementation of `DepartmentController`. It shows the example of relationship fetching. We

use `DataFetchingEnvironment` to detect if the input query contains a relationship field. In our case, it may be `employees` or `organization`. If any of those fields is defined we add the particular relation to the JOIN statement. The same approach applies to both `department` and `deparments` methods

```
@Controller
public class DepartmentController {

   DepartmentRepository departmentRepository;
   OrganizationRepository organizationRepository;

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

   @MutationMapping
   public Department newDepartment(@Argument DepartmentInput department) {
      Organization organization = organizationRepository
         .findById(department.getOrganizationId()).get();
      return departmentRepository.save(new Department(null, department.getName(), null, organization));
   }

   @QueryMapping
   public Iterable<Department> departments(DataFetchingEnvironment environment) {
      DataFetchingFieldSelectionSet s = environment.getSelectionSet();
      List<Specification<Department>> specifications = new ArrayList<>();
      if (s.contains("employees") && !s.contains("organization"))
         return departmentRepository.findAll(fetchEmployees());
      else if (!s.contains("employees") && s.contains("organization"))
         return departmentRepository.findAll(fetchOrganization());
      else if (s.contains("employees") && s.contains("organization"))
         return departmentRepository.findAll(fetchEmployees().and(fetchOrganization()));
      else
         return departmentRepository.findAll();
   }

   @QueryMapping
   public Department department(@Argument Integer id, DataFetchingEnvironment environment) {
      Specification<Department> spec = byId(id);
      DataFetchingFieldSelectionSet selectionSet = environment
         .getSelectionSet();
      if (selectionSet.contains("employees"))
         spec = spec.and(fetchEmployees());
      if (selectionSet.contains("organization"))
         spec = spec.and(fetchOrganization());
      return departmentRepository.findOne(spec).orElseThrow(NoSuchElementException::new);
   }

    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);
   }
}
```

Here's the `OrganizationController` implementation.

```
@Controller
public class OrganizationController {

   OrganizationRepository repository;

   OrganizationController(OrganizationRepository repository) {
      this.repository = repository;
   }

   @MutationMapping
   public Organization newOrganization(@Argument OrganizationInput organization) {
      return repository.save(new Organization(null, organization.getName(), null, null));
   }

   @QueryMapping
   public Iterable<Organization> organizations() {
      return repository.findAll();
   }

   @QueryMapping
   public Organization organization(@Argument Integer id, DataFetchingEnvironment environment) {
      Specification<Organization> spec = byId(id);
      DataFetchingFieldSelectionSet selectionSet = environment
         .getSelectionSet();
      if (selectionSet.contains("employees"))
         spec = spec.and(fetchEmployees());
      if (selectionSet.contains("departments"))
         spec = spec.and(fetchDepartments());
      return repository.findOne(spec).orElseThrow();
   }

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

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

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

Create Unit Tests

Once we created the whole logic it's time to test it. In the next section, I'll show you how to use GraphiQL IDE for that. Here, we are going to focus on unit tests. The simplest way to start with Spring for GraphQL tests is through the `GraphQLTester` bean. We can use in mocked web environment. You can also build tests for HTTP layer with another bean – `HttpGraphQlTester`. However, it requires us to provide an instance of `WebTestClient`.

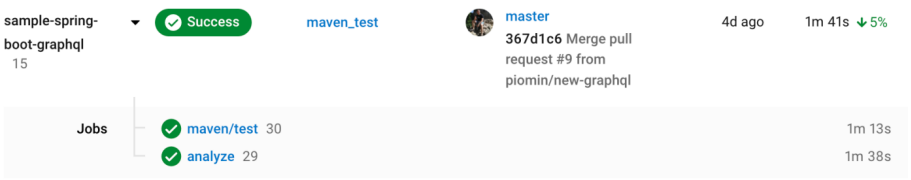
Here are the test for the `Employee` `@Controller`. Each time we are building an inline query using the GraphQL notation. We need to annotate the whole test class with `@AutoConfigureGraphQlTester`. Then we can use the DSL API provided by the `GraphQLTester` to get and verify data from backend. Besides two simple tests we also verifies if `EmployeeFilter` works fine in the `findWithFilter` method.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureGraphQlTester
public class EmployeeControllerTests {

   @Autowired
   private GraphQlTester tester;

   @Test
   void addEmployee() {
      String query = "mutation { newEmployee(employee: { firstName: \"John\" lastName: \"Wick\" position: \"developer\" salary: 10000
      Employee employee = tester.document(query)
              .execute()
              .path("data.newEmployee")
              .entity(Employee.class)
              .get();
      Assertions.assertNotNull(employee);
      Assertions.assertNotNull(employee.getId());
   }

   @Test
   void findAll() {
      String query = "{ employees { id firstName lastName salary } }";
      List<Employee> employees = tester.document(query)
             .execute()
             .path("data.employees[*]")
             .entityList(Employee.class)
             .get();
      Assertions.assertTrue(employees.size() > 0);
      Assertions.assertNotNull(employees.get(0).getId());
      Assertions.assertNotNull(employees.get(0).getFirstName());
   }

   @Test
   void findById() {
      String query = "{ employee(id: 1) { id firstName lastName salary } }";
      Employee employee = tester.document(query)
             .execute()
             .path("data.employee")
             .entity(Employee.class)
             .get();
      Assertions.assertNotNull(employee);
      Assertions.assertNotNull(employee.getId());
      Assertions.assertNotNull(employee.getFirstName());
   }

   @Test
   void findWithFilter() {
      String query = "{ employeesWithFilter(filter: { salary: { operator: \"gt\" value: \"12000\" } }) { id firstName lastName salary
      List<Employee> employees = tester.document(query)
             .execute()
             .path("data.employeesWithFilter[*]")
             .entityList(Employee.class)
             .get();
      Assertions.assertTrue(employees.size() > 0);
      Assertions.assertNotNull(employees.get(0).getId());
      Assertions.assertNotNull(employees.get(0).getFirstName());
   }
}
```

Test tests for the `Deparment` type are very similar. Additionally, we are testing join statements in the `findById` test method by declaring the `organization` field in the query.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureGraphQlTester
public class DepartmentControllerTests {

   @Autowired
   private GraphQlTester tester;

   @Test
   void addDepartment() {
      String query = "mutation { newDepartment(department: { name: \"Test10\" organizationId: 1}) { id } }";
      Department department = tester.document(query)
             .execute()
             .path("data.newDepartment")
             .entity(Department.class)
             .get();
      Assertions.assertNotNull(department);
      Assertions.assertNotNull(department.getId());
   }

   @Test
   void findAll() {
      String query = "{ departments { id name } }";
      List<Department> departments = tester.document(query)
             .execute()
             .path("data.departments[*]")
             .entityList(Department.class)
             .get();
      Assertions.assertTrue(departments.size() > 0);
      Assertions.assertNotNull(departments.get(0).getId());
      Assertions.assertNotNull(departments.get(0).getName());
   }

   @Test
   void findById() {
      String query = "{ department(id: 1) { id name organization { id } } }";
      Department department = tester.document(query)
             .execute()
             .path("data.department")
             .entity(Department.class)
             .get();
      Assertions.assertNotNull(department);
      Assertions.assertNotNull(department.getId());
      Assertions.assertNotNull(department.getOrganization());
      Assertions.assertNotNull(department.getOrganization().getId());
   }

}
```

Each time you are cloning my repository you can be sure that the examples work fine thanks to automated tests. You can always verify the status of the repository build in my CircleCI pipeline.



Testing with GraphiQL

We can easily start the application with the following Maven command:

```
$ mvn clean spring-boot:run
```

Once you do it, you can access the GraphiQL tool under the address http://localhost:8080/graphiql. The app inserts some demo data to the H2 database on startup. GraphiQL provides content assist for building GraphQL queries. Here's the sample query tested there.

```
employeesWithFilter(filter: {
  salary: {
    operator: "gt"
    value: "12000"
  }
}) {
  id
  firstName
  lastName
  position
}
```

```
{
  "data": {
    "employeesWithFilter": [
      {
        "id": "3",
        "firstName": "Tracy",
        "lastName": "Smith",
        "position": "Architect"
      },
      {
        "id": "4",
        "firstName": "Lucy",
        "lastName": "Kim",
        "position": "Developer"
      },
      {
        "id": "5",
        "firstName": "Peter",
        "lastName": "Wright",
        "position": "Director"
      },
      {
        "id": "6",
        "firstName": "Alan",
        "lastName": "Murray",
        "position": "Developer"
      }
    ]
  }
}
```

Variables Headers

Final Thoughts

Spring for GraphQL is very interesting project and I will be following its development closely. Besides `@Controller` support, I tried to use the querydsl integration with Spring Data JPA repositories. However, I've got some problems with it, and therefore I avoided to place that topic in the article. Currently, Spring for GraphQL is the third solid Java framework with high-level GraphQL support for Spring Boot. My choice is still Netflix DGS, but Spring for GraphQL is rather during the active development. So we can probably except some new and useful features soon.

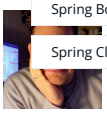
graphiql graphql H2 JPA Spring Boot Spring Data spring-graphql

Written by:

piotr.minkowski

View All Posts →

1 C… 1



…y endpoint can have multiples ROLES and as such have different REQUEST/RESPONSE datasets.

Where do you place those and what happens if there is a failure in the schema stitching chain?

Also how does one cache? GraphQL is known for being unable to cache.

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

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