OCI | WE ARE SOFTWARE ENGINEERS.

GORM + GraphQL

James Kleeh

AGENDA

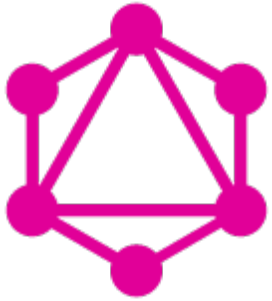
1. GraphQL
    1. Features / Benefits
    2. Challenges
    3. Limitations
2. GORM GraphQL
    1. Features / Benefits
    2. Customizations

The Problem

REST is too ***rigid***!

The same data types and format is returned for every execution. Some use cases of the API may only need a fraction of the properties. The extra data that is sent is wasting time and resources both on the server and on the network.

The Solution



GraphQL

An API middleware specification that allows clients to control the response of the server for each individual request. Has no dependencies on any communication or persistence layer.

OCI | WE ARE SOFTWARE ENGINEERS

What is GraphQL?

It has nothing to do with a graphing database!

- Specification
- Schema Definition
- Introspective
- Application Layer Middleware
- Strongly Typed

OCI | WE ARE SOFTWARE ENGINEERS

Key Advantages

Efficiency, Efficiency, Efficiency

- Multiple operations simultaneously
- Only the data requested is returned
- Self documenting
- Client features to control response based on variables and data types
- One “endpoint” can support many use cases

OCI | WE ARE SOFTWARE ENGINEERS

Key Disadvantages

- Truly Dynamic Data Not Easy
- Recursive Data Structures
- Security Not As Expressive
- Time Intensive To Setup
- Another Point Of Failure
- Not As Simple For The Client

DEMO

GitHub API v4 is implemented with GraphQL!

https://developer.github.com/v4/

OCI | WE ARE SOFTWARE ENGINEERS

The Challenge

- Generating a schema for all persistent entities that should be exposed
- Parsing the requested properties to execute the most efficient queries
- Implement communication with the API over HTTP

Creating A Schema

Due to the nature of GraphQL schemas, it is often required to create multiple “types” that represent the same underlying resource.

```
class Book {

    String title

    Date dateCreated
    Date lastUpdated

    static hasMany = [pages: Page]

    static constraints = {
    }
}
```

```
class Page {

    Integer number
    String text

    static belongsTo = [book: Book]

    static constraints = {
    }
}
```

Creating A Schema (CREATE)

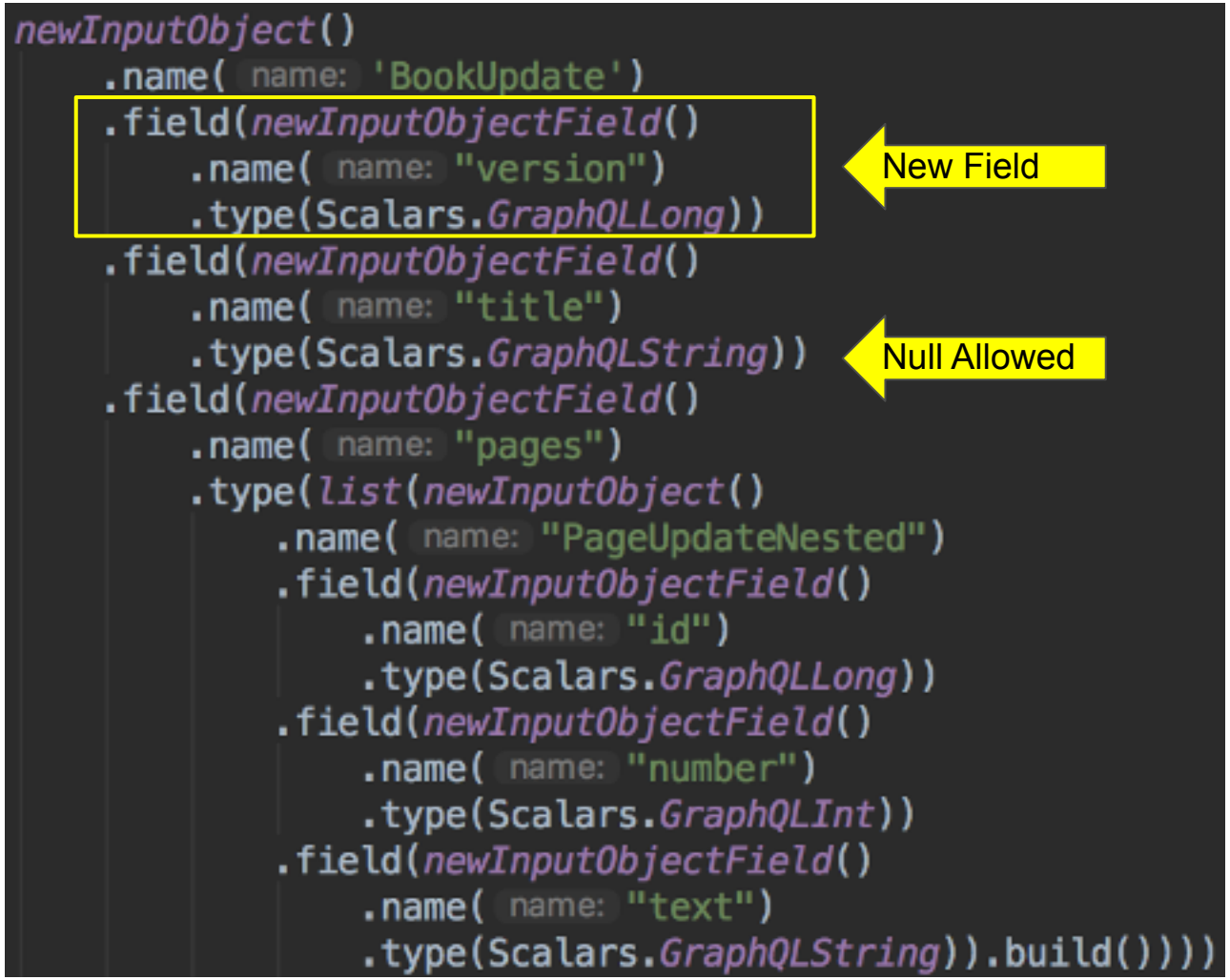
In order to create Book objects, we need to define an argument type for users to pass in the data required to successfully persist the Book.

```
newInputObject()
    .name( name: 'BookCreate')
    .field(newInputObjectField()
        .name( name: "title")
        .type(nonNull(Scalars.GraphQLString)))
    .field(newInputObjectField()
        .name( name: "pages")
        .type(list(newInputObject()
            .name( name: "PageCreateNested")
            .field(newInputObjectField()
                .name( name: "id")
                .type(Scalars.GraphQLLong))
            .field(newInputObjectField()
                .name( name: "number")
                .type(Scalars.GraphQLInt))
            .field(newInputObjectField()
                .name( name: "text")
                .type(Scalars.GraphQLString)).build())))
```

Creating A Schema (UPDATE)

In order to update a Book object, we need to define an argument type for users to pass in the data required to successfully update the Book.

The version property was added to support optimistic locking. The title field is no longer null to allow users to send partial data.



Implementing HTTP Communication

http://graphql.org/learn/serving-over-http/

While there is nothing in the official specification that determines how to communicate with a GraphQL server over HTTP, the website has guidelines on:

- Content Types
- Request Methods
- Parameter Names
- Endpoint URI
- Response Format

GORM + GraphQL Has You Covered

GORM-GraphQL is a library that integrates the GraphQL technology with GORM to automatically create a schema and operations for persistent entities (domain classes). A Grails plugin also exists to implement the HTTP communication as well as several other features that are specific to Grails. https://github.com/grails/gorm-graphql

OCI | WE ARE SOFTWARE ENGINEERS

Main Features

- Schema Generation
- Efficient Query Building
- Grails Plugin
  - HTTP Communication
  - GraphiQL Browser
  - Spring Beans
  - Data Binding
  - Testing Support

Getting Started

To expose a domain class as a set of GraphQL operations, simply add a static property.

By default, six operations will be created for each domain class that has been designated for GraphQL processing.

Create, Update, Delete, List, Show, Count

```
class Book {

    String title

    Date dateCreated
    Date lastUpdated

    static hasMany = [pages: Page]

    static constraints = {
    }

    static graphql = true
}
```

DEMO

Example Application

https://github.com/grails-samples/graphql-demo

Customizations

The default operations are great, but…

How do I customize the existing operations?

How do I add my own operations?

How do I add custom properties?

How do I deprecate properties?

How do I add a description for my properties?

How do I supply my own queries?

How do I change how validation errors are returned?

How do I change the response for a delete?

How do I …

Definitions

Data Fetcher: The glue between the persistence layer and the GraphQL server. Responsible for retrieving all data.

Interceptor: Intercepts data fetchers for custom and provided operations and provides the ability to cancel (return null). Schema interceptors can also be created to manipulate the GraphQL schema manually.

Data Binders: Responsible for using the GraphQL arguments and applying the data to a persistent entity instance. If the plugin is being used, the default Grails data binding is configured for all domains.

Scalars: Handles type conversion in GraphQL. Converts properties of standard or custom Java types to a simple format (Number, Float, String, etc)

Customizations

There are several areas where customizations can occur.

Persistent Entity Mapping

- Define Metadata (Description, etc)
- Add New Properties
- Create New Operations
- Override Data Fetchers For Properties
- Exclude Properties

Bean Post Processor (Plugin Only)

- Add Custom Scalar Types
- Register Data Fetchers For Domains
- Register Data Binders For Domains
- Register Interceptors

Bean Overrides (Plugin Only)

- Delete Response Handler
- Validation Errors Response Handler
- Many Others…

Customizations - Getting Started

Replace:

```
static graphql = true
```

With:

```
static graphql = GraphQLMapping.build {


}
```

Customizations - Return Types

When declaring a custom property, operation, argument, or field, the “type” can be declared in 3 different ways:

- Class (Author): When the return type is a single instance
- List<Class> ([Author]): When the return type is a collection of instances
- PaginatedType: When the return type is a collection, but the result should be a paginated result (defined by the GraphQLPaginationResponseHandler)
- String (“CustomType”): When the return type is custom and will be defined

add(name, type) {} // Adding a custom property

query(name, type) {} // Defining a new query operation

mutation(name, type) {} // Defining a new mutation operation

Customizations - Custom Types

Custom types are a collection of fields.

Custom types require a data fetcher to be provided in order to return the desired data to GraphQL.

The defined fields follow the same rules for return types and can be a custom type themselves.

```
class Author {

    LocalDate birthDay

    Map getAge() {
        LocalDate now = LocalDate.now()

        [days: ChronoUnit.DAYS.between(birthDay, now),
         months: ChronoUnit.MONTHS.between(birthDay, now),
         years: ChronoUnit.YEARS.between(birthDay, now)]
    }

    static graphql = GraphQLMapping.build {
        add( 'age',  typeName: 'Age') {
            input  input: false
            type {
                field('days', Long)
                field('months', Long)
                field('years', Long)
            }
        }
    }
}
```

Customizations - Metadata

Metadata including a description and deprecation information can be provided for persistent entities and properties, custom properties, arguments, fields, etc.

The metadata can be supplied in one of two ways; The mapping DSL and the @GraphQL annotation. In some cases, it is only possible to use the annotation because the given class is not a persistent entity. Examples include embedded POGOs and enums.

Examples

Adding a description to a property (Mapping DSL):

```
property( name: "name", description: "The name of the Author")
```

And with the annotation:

```
@GraphQL("The name of the author")
String name
```

Customizations - Custom Operations

Custom operations allow you to provide “endpoints” for users of the API. They fall into one of two categories, query and mutation. It is necessary to supply a data fetcher for the new operation.

```
query( name: 'usersByRole', [User]) {
    argument('role', Long)
    dataFetcher(new UsersByRoleDataFetcher())
}
```

This example could be part of the typical User -> UserRole <- Role domain relationship. A new query operation “usersByRole” is being created that returns a list of User instances. It has one argument, “role”, which is a Long. The data fetcher is responsible for retrieving the list of users.

Customizations - Custom Operations (contd.)

```groovy
@CompileStatic
class UsersByRoleDataFetcher extends EntityDataFetcher<List<User>> {

    UsersByRoleDataFetcher() {
        super(User.gormPersistentEntity)
    }

    @GrailsCompileStatic
    @Override
    protected List executeQuery(DataFetchingEnvironment environment, Map queryArgs) {
        Role role = Role.load((Serializable) environment.getArgument( name: 'role'))
        def users = UserRole.where { role == role }.property( property: 'user')
        User.where {
            id in users
        }.list(queryArgs)
    }
}
```

Provided By Library

Type being queried

Join Properties

Data Fetchers - Context

Data Fetchers will typically not participate in the application context and therefore will not be able to inject any dependent beans. In addition, data fetchers are not aware of the current HTTP request since GraphQL is not specific to HTTP. To be able to pass necessary information to your data fetchers, you can manipulate the “context”.

The context is available in the data fetching environment and can be any type (`DataFetchingEnvironment#getContext()`). The plugin exposes a bean that is used to populate the context. To customize it, override the bean with your own.

By default the context is a Map that contains a key “locale” that stores the locale of the current request.

Data Fetchers - Context (contd.)

```
class MyGraphQLContextBuilder extends DefaultGraphQLContextBuilder {

    @Autowired
    SpringSecurityService springSecurityService

    @Override
    Map buildContext(GrailsWebRequest request) {
        Map context = super.buildContext(request)
        context.springSecurityService = springSecurityService
        context
    }
}
```

It is not necessary to extend from the default class, however the object returned should either be a map with a "locale" key, or the returned object should implement **`LocaleAwareContext`** in order to function correctly with the default data fetchers.

Customizations - Interceptors

Since all requests go to a single endpoint (typically /graphql), security can not be defined at the URL level. In GraphQL, the current best practice is to simply return null for data that a user doesn’t have access to.

There are two flavors of interceptors that can be registered with gorm-graphql. The first is an interceptor that gets executed before the data fetcher that will return the appropriate data. The other is designed to allow users to hook into the schema creation process at the lowest level, the graphql-java API.

For provided operations, the interceptor has access to the data fetching environment and the data fetcher type. For custom operations, the interceptor has access to the data fetching environment and the name of the operation.

Customizations - Interceptors (contd.)

```
class AuthorUpdateInterceptor extends BaseGraphQLFetcherInterceptor {

    @Override
    boolean onMutation(DataFetchingEnvironment environment, GraphQLDataFetcherType type) {
        if (type == GraphQLDataFetcherType.UPDATE) {
            Author author = Author.get((Long) environment.getArgument( name: "id"))
            Map context = (Map) environment.context
            SpringSecurityService springSecurityService = (SpringSecurityService) context.springSecurityService
            springSecurityService.currentUser == author.user
        } else {
            true
        }
    }
}
```

The above example verifies that the author being updated is owned by the current user. The author is retrieved based on the “id” argument and compared with the current user found through the SpringSecurityService that was added to the context in a previous slide.

Customizations - Interceptors (contd.)

To register the interceptor with GraphQL in a Grails application, define a bean post processor and access the interceptor manager.

```
class MyGraphQLCustomizer extends GraphQLPostProcessor {

    @Override
    void doWith(GraphQLInterceptorManager interceptorManager) {
        interceptorManager.registerInterceptor(Author, new AuthorUpdateInterceptor())
    }
}
```

Then register the class as a bean in resources.groovy.

```
beans = {
    myGraphQLCustomizer(MyGraphQLCustomizer)
}
```

Customizations - Data Binding

Providing your own data binding implementation for one or all entities is simple. Create a class that extends **`GraphQLDataBinder`** and register it with the data binder manager.

```
@CompileStatic
class UserDataBinder extends GrailsGraphQLDataBinder {

    @Override
    void bind(Object object, Map data) {
        Integer first = (Integer)data.remove( key: 'firstNumber')
        Integer second = (Integer)data.remove( key: 'secondNumber')
        if (first != null && second != null) {
            data.put('addedNumbers', first + second)
        }
        super.bind(object, data)
    }
}
```

Customizations - Data Binding (contd.)

To register the data binder with GraphQL in a Grails application, define a bean post processor and access the data binder manager.

```
class MyGraphQLCustomizer extends GraphQLPostProcessor {

    @Override
    void doWith(GraphQLDataBinderManager dataBinderManager) {
        dataBinderManager.registerDataBinder(User, new UserDataBinder())
    }
}
```

Then register the class as a bean in resources.groovy.

```
beans = {
    myGraphQLCustomizer(MyGraphQLCustomizer)
}
```

Testing

The plugin provides a functional testing trait

Easy to create a unit test for your schema that is specific to your datastore

Functional Testing

```groovy
@Integration
class PetIntegrationSpec extends Specification implements GraphQLSpec {

    void "test calling graphql"() {
        when:
        def resp = graphQL.graphql( requestBody: """
            mutation {
                petCreate(pet: {
                    name: "Butch",
                    type: DOG
                }) {
                    id
                }
            }
        """)

        def result = resp.json

        then:
        result.data.petCreate.id == 1
    }
}
```

Schema Testing

```
class SchemaSpec extends Specification implements GraphQLSchemaSpec {

    @Shared @AutoCleanup HibernateDatastore hibernateDatastore
    @Shared GraphQLSchema schema
    @Shared GraphQLObjectType queryType
    @Shared GraphQLObjectType mutationType

    void setupSpec() {
        hibernateDatastore = new HibernateDatastore(
                DatastoreUtils.createPropertyResolver(
                        Collections.singletonMap(Settings.SETTING_DB_CREATE, 'create-drop')),
                GeneralPackage.getPackage(), HibernatePackage.getPackage())
        schema = new Schema(hibernateDatastore.mappingContext).generate()
        queryType = schema.queryType
        mutationType = schema.mutationType
    }
```

Q&A

OBJECT COMPUTING