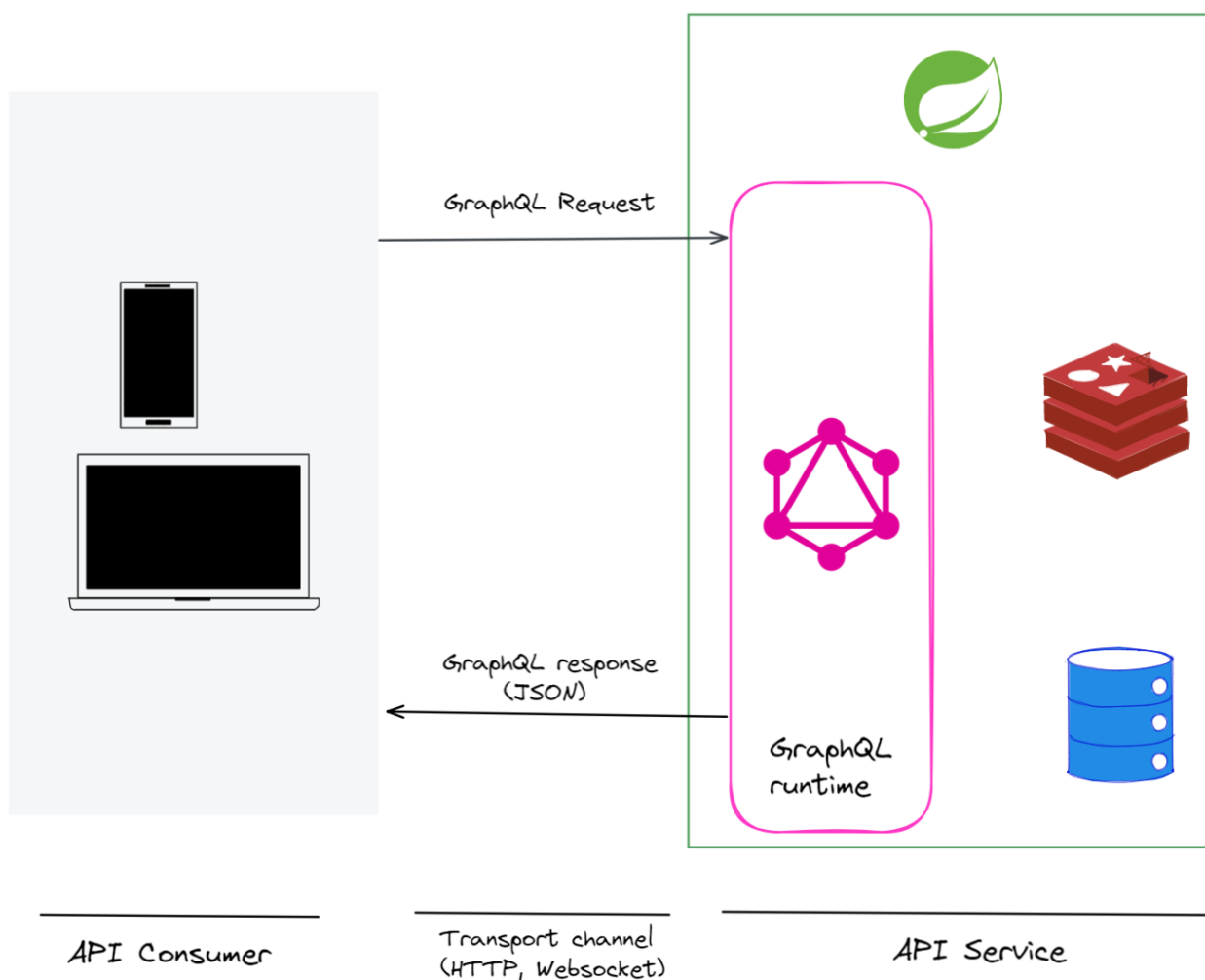




Getting started with Spring Boot GraphQL service

by **Pankaj** — July 23, 2022 in **GraphQL, Spring Boot** Reading Time: 20 mins read

0 0 0



Spring Boot GraphQL service

1
SHARES1.2k
VIEWS

f Share on Facebook

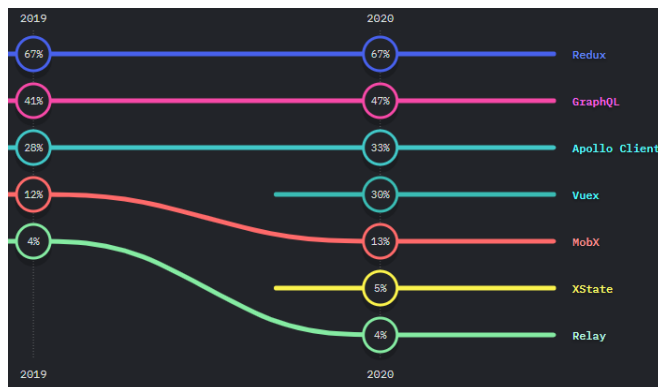
t Share on Twitter

in Share on LinkedIn



Facebook developed GraphQL in 2012 to solve its problem of mobile communication. Since it was open-sourced in 2015, it has been adopted by companies like Twitter, Shopify, Lyft, Airbnb, Github, Yelp, and many more.

There is no doubt that GraphQL adoption is on the rise. The State of JavaScript 2020 report [mentions](#) that only 6% of developers surveyed had used it in 2016, however, 47% of developers used it in 2020.



Spring for GraphQL, which recently released version 1.0, provides a higher level abstraction for building the **Spring Boot GraphQL** service.

If you want a quick overview of GraphQL and how to implement the Spring Boot GraphQL service then you have come to the right place.

Let's get started!

In this article

[What is GraphQL?](#)

[Features of GraphQL](#)

[Strongly typed](#)

[Hierarchal data](#)

[Client-specific response](#)

[Introspection](#)

[Why GraphQL?](#)

[GraphQL document](#)

[GraphQL operation](#)

[Schemas and types](#)

[Object type and fields](#)

[GraphiQL Editor](#)

[Spring Boot GraphQL support](#)

[Server transport](#)

[GraphQL service implementation](#)

[GraphQL dependency](#)

[RuntimeWiring](#)

[TypeRuntimeWiring](#)

[Data Fetchers](#)

[RuntimeWiringConfigurer](#)

[Running and Testing](#)

[Summary](#)

What is GraphQL?

GraphQL is a query language and server runtime.

1. **Query language:** GraphQL is a query language for API. In GraphQL, you design API based on its type system. A GraphQL API is expressed as a statically typed schema.
2. **Server runtime:** On the service side, a GraphQL service provides a **runtime layer that describes the structure of data exposed by API**, and this runtime layer is responsible for **parsing GraphQL requests and calling the appropriate data fetcher (also called resolver) for each field**.

In short, GraphQL is a **query language for API** and a **server-side runtime** for executing queries. Also, it's not tied to any specific programming language, database, or storage engine.

A GraphQL query to return book by id:

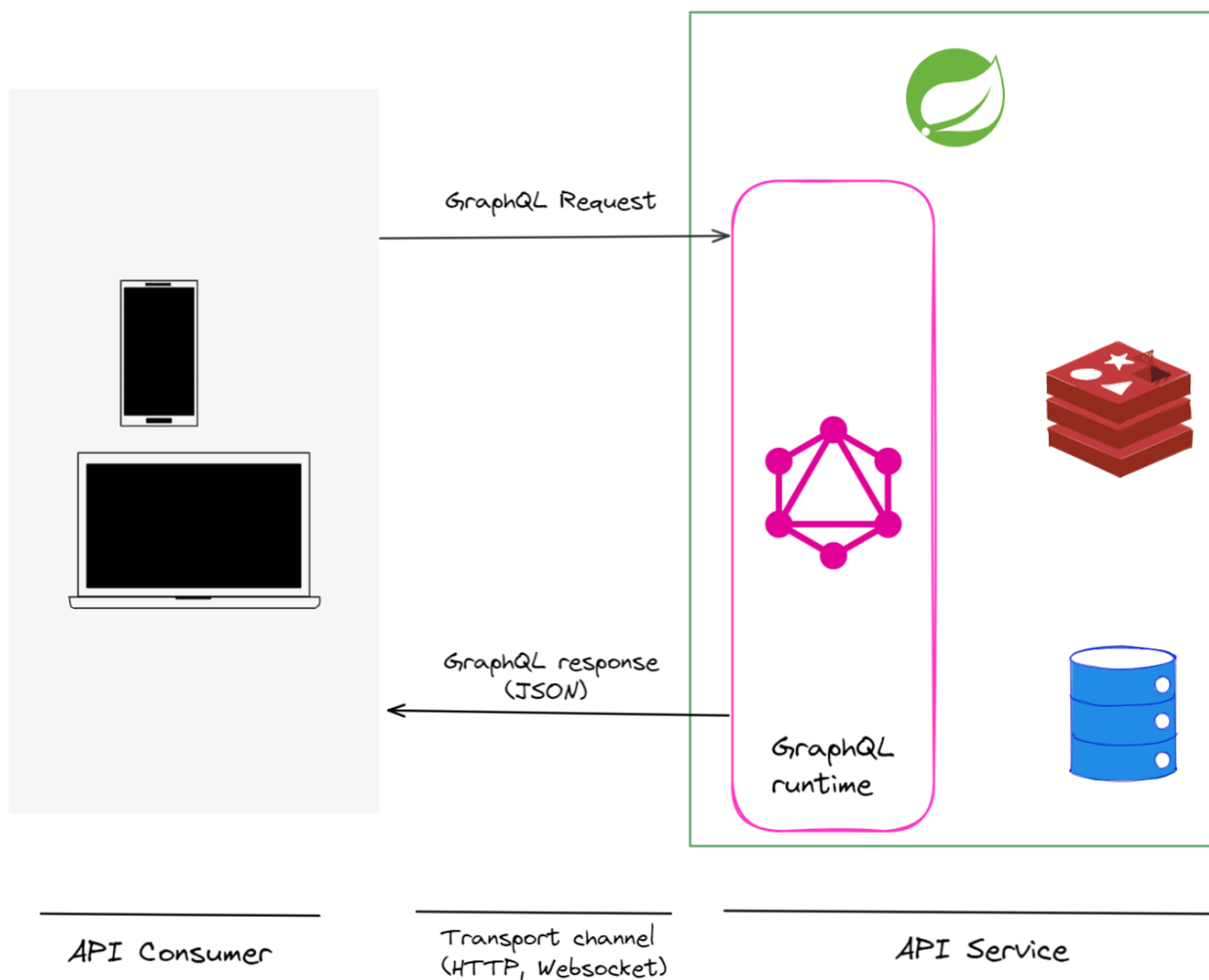
```
{
  bookById(id: 4) {
    name
    author
  }
}
```

Response

```
{
  "data": {
    "bookById": {
      "name": "Atomic Habits",
      "author": "James Clear"
    }
  }
}
```

Spring Boot GraphQL Service





Features of GraphQL

Some important features of GraphQL are:

Strongly typed

GraphQL APIs are **strongly typed** (expressed as schema). Because of its strongly typed nature, GraphQL APIs are more predictable and discoverable. Moreover, the type system also brings other benefits like introspection.

Hierarchal data

As the Graph in GraphQL suggests, GraphQL provides first-class support for hierarchal data. In GraphQL, you can create a request as a graph of related fields. A GraphQL response is shaped like the request which is a very natural way for a client to describe its data requirement.

Client-specific response

REST APIs are often criticized for over-fetching. Generally, a REST API returns all data under a resource even though you need fraction of the data. This is not a problem in GraphQL as **a GraphQL client can choose to request data on the field level**



granularity.

Introspection

Because of the statically typed nature of GraphQL, any **client can 'introspect' the server** and ask for the schema. One popular GraphQL tool that relies on this concept is **GraphiQL**, a feature-rich browser-based editor to explore and test GraphQL requests.

You can explore some popular GraphiQL interfaces

- Star Wars GraphQL API: az.dev/swapi-graphql
- GitHub GraphQL API: az.dev/github-api

Why GraphQL?

[REST](#) is the most popular way of building APIs, but it has many shortcomings. Let's understand the problem with REST APIs that GraphQL promises to solve.

- **Standard:** [GraphQL specification](#) is maintained by the GraphQL community. Thus, it provides a comprehensive standard for developing APIs in a maintainable way. Even though REST API has been around for many years, the industry-wide standard is non-existent.
- **API Documentation:** In REST API, we maintain API documentation in OpenAPI spec. One of the common problems with REST API is that API documentation and implementation drift in due course. Unlike REST API, in GraphQL, you **don't need to maintain APIs documentation separately**.
- **Over fetching:** Over fetching is a big problem in REST API, particularly for mobile applications. Firstly, pure REST APIs are built around resources. Secondly, it doesn't have any concept of partial response. For example – a mobile e-commerce app, that displays order history, needs to show the name of the product, purchase date, and price but REST API `/orders` API may return many more fields including payment details, discounts, shipment details, etc, making it inefficient for mobile use.
- **Under fetching:** As REST APIs are built around resources and are usually fine-grained, a client application may have to call multiple APIs to construct a view. This is typically not a problem in GraphQL because of its hierarchal nature.

This doesn't mean GraphQL is perfect in every sense, it has its own shortcomings. For example:

- As REST API is around for many years, it has a very **well-defined security measure but the same is not true for GraphQL**.
- Compared to REST APIs, **implementing client-side caching is very hard in GraphQL**.
- Things like **rate limiting are harder to implement in GraphQL**. This can very easily lead to a denial of service attack, as it's very much possible to bring down the whole service by requesting deeply nested data.

As with everything in life, choosing between REST and GraphQL boils down to making trade-offs.

GraphQL document

A GraphQL **request structure is called a document**. A GraphQL document may contain **operations, variables, directives, and fragments** (a reusable type definition).

GraphQL operation

A GraphQL service can support the following operations

1. **Queries:** queries represent **READ** operations.
2. **Mutation:** mutation involves **WRITE** then **READ** operation.



5. **Subscription:** subscription is used for continuous **READ** (for example, over websocket).

Schemas and types

Every GraphQL API defines a set of **types** that describe the data you can query on that API. A GraphQL request is validated and executed against that schema.

Object type and fields

The most basic components of a GraphQL schema are **object types**, which represent an object you can fetch from a service, and what fields it has. In the GraphQL schema language, you can represent a **Book** **object type** as:

```
type Book {  
  id : ID  
  name : String  
  author: String  
  price: Float  
  ratings: [Rating]  
}
```

In the above example,

- **Type:** **Book** is GraphQL **object type**.
- **Fields:** id, name, author, price, and ratings are fields of **Book** type.
- **Scalar types:** String, Float, and Int are some of the built-in scalar types.
- **ID:** the ID scalar type represents a unique identifier, often used to refetch an object. Additionally, the ID type is serialized in the same way as a String.

Example of a GraphQL book-catalog API

```
type Query {  
  books: [Book]  
  bookById(id : ID) : Book  
}  
  
type Book {  
  id : ID  
  name : String  
  author: String  
  price: Float  
  ratings: [Rating]  
}  
  
type Rating {  
  id: ID  
  rating: Int  
  comment: String
```



```
}
```

GraphQL Editor

One of the reasons GraphQL is very popular is **GraphiQL (pronounced “graphical”)**, an **open-source web application (written with React.js and GraphQL) that runs in a browser**. The best thing about GraphiQL is that it provides **intelligent type-ahead and auto-completion features**, which is possible because of GraphQL's statically typed schema.



Star Wars GraphiQL editor (az.dev/swapi-graphql)

Spring Boot GraphQL support

Spring Boot GraphQL provides support for Spring applications built on [GraphQL Java](#). It supports the handling of GraphQL requests over HTTP, WebSocket, and RSocket.

Spring for GraphQL is the successor of the [GraphQL Java Spring](#) project from the GraphQL Java team. It aims to be the foundation for all Spring, GraphQL applications.

Server transport


GraphQL specification itself doesn't talk anything about transport layer protocol. [GraphQL over HTTP](#) specification extends GraphQL specification to **cover the topic of serving GraphQL services over HTTP**.

As per GraphQL over HTTP specification, requests must use **HTTP POST with request details included as JSON in the request body**. Once the JSON body has been successfully decoded, the HTTP response status is always **200 (OK)**, and any *errors from GraphQL request execution appear in the "errors" section of the GraphQL response*.

The default and preferred choice of the media type is `"application/graphql+json"`, but `"application/json"` is also supported.

GraphQL service implementation

In Spring Boot, you can implement the GraphQL service using **low-level GraphQL Java API or a higher-level abstraction as `Controller` annotation (similar to REST API)**. However, if you are just starting out then, I think it's better to start with the low-level GraphQL Java implementation as it covers some important concepts. In this article, we'll use GraphQL Java APIs rather than `Controller` annotation.

The working code example of this article is listed on [GitHub](#) . To run the example, clone the repository, and import **graphql-spring** as a project in your favorite IDE as a `Gradle` project.

In this code example, we'll implement the following GraphQL API.

```
type Query {  
  books: [Book]  
  bookById(id : ID) : Book  
}  
  
type Book {  
  id : ID  
  name : String  
  author: String  
  price: Float  
  ratings: [Rating]  
}  
  
type Rating {  
  id: ID  
  rating: Int  
  comment: String  
  user: String  
}
```

GraphQL dependency

To implement GraphQL in Spring Boot, the main dependency you need to define is `spring-boot-starter-graphql`. For the transport layer, I have added dependency on `spring-boot-starter-webflux`. Additionally, you can use Spring MVC, Instead of WebFlux, as Spring provides support for both.




```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-graphql'
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'io.projectreactor:reactor-test'
    testImplementation 'org.springframework.graphql:spring-graphql-test'
}
```

RuntimeWiring

GraphQL Java `RuntimeWiring.Builder` is used to register `DataFetcher` s, type resolvers, custom scalar types, and more. You can declare `RuntimeWiringConfigurer` beans in your Spring config to get access to the `RuntimeWiring.Builder` .

You first create *type* wiring for Query type by registering `Query` type with `RuntimeWiring.Builder` as

```
return new RuntimeWiringConfigurer() {
    @Override
    public void configure(RuntimeWiring.Builder builder) {

        builder.type(
            "Query",
            new UnaryOperator<TypeRuntimeWiring.Builder>() {
                @Override
                public TypeRuntimeWiring.Builder apply(TypeRuntimeWiring.Builder builder) {
                    .....
                }
            });
    }
};
```

TypeRuntimeWiring

As the type `Query` has two fields `books` and `bookById` , the next step is to define `DataFetcher` for fields `books` and `bookById` and register data fetchers with `TypeRuntimeWiring.Builder` as

```
return new RuntimeWiringConfigurer() {
    @Override
    public void configure(RuntimeWiring.Builder builder) {
        builder.type(
            "Query",
            new UnaryOperator<TypeRuntimeWiring.Builder>() {
                @Override
                public TypeRuntimeWiring.Builder apply(TypeRuntimeWiring.Builder builder) {
                    return builder
                        .dataFetcher(
```



```

        new DataFetcher<>() {
            @Override
            public Collection<Book> get(DataFetchingEnvironment environment)
                throws Exception {
                return bookCatalogService.getBooks();
            }
        })
        .dataFetcher(
            "bookById",
            new DataFetcher<>() {
                @Override
                public Book get(DataFetchingEnvironment environment) throws Exception {
                    return bookCatalogService.bookById(
                        Integer.parseInt(environment.getArgument("id")));
                }
            });
    }
    });
}
};

```

In the above code,

1. Data Fetcher for **field** `books` fetches all books by calling the service method `bookCatalogService.getBooks()` .
2. Data Fetcher for **field** `bookById` fetches a book by calling the service method `bookCatalogService.bookById` . You can get arguments from the `DataFetchingEnvironment` as `environment.getArgument("id")` .

Similarly, for **type** `Book` and the **field** `ratings` , you can define `DataFetcher` as:

```

builder.type(
    "Book",
    new UnaryOperator<TypeRuntimeWiring.Builder>() {
        @Override
        public TypeRuntimeWiring.Builder apply(TypeRuntimeWiring.Builder builder) {
            return builder.dataFetcher(
                "ratings",
                new DataFetcher<>() {
                    @Override
                    public List<Rating> get(DataFetchingEnvironment environment)
                        throws Exception {
                        return bookCatalogService.ratings(environment.getSource());
                    }
                });
        }
    });
}
};

```

Data Fetchers

Probably the most important concept for a GraphQL Java server is a `DataFetcher` . While GraphQL Java is executing a query, it calls the appropriate `DataFetcher` for each field it encounters in the query.





Every field from the schema has a **DataFetcher** associated with it. If you don't specify any **DataFetcher** for a specific field, then the default **PropertyDataFetcher** is used.

In above example `Query.books`, `Query.bookById` has a data fetcher, as **does each field in type Book**. we **typically don't need specialized data fetchers on each field**. As GraphQL Java ships with a smart `graphql.schema.PropertyDataFetcher` that knows how to follow POJO patterns based on the field name. In the example above there is a `name` field and hence it will try to look for a `public String getName()` POJO method to get the data.

RuntimeWiringConfigurer

You can change `UnaryOperator<TypeRuntimeWiring.Builder>` as a lambda expression. Then, the last step left is to tell Spring about GraphQL wiring by defining **RuntimeWiringConfigurer** as bean as:

```
@Configuration
public class GraphQLConfiguration {

    @Bean
    public RuntimeWiringConfigurer runtimeWiringConfigurer(BookCatalogService bookCatalogService) {

        return builder -> {
            builder.type(
                "Query",
                wiring ->
                    wiring
                        .dataFetcher("books", environment -> bookCatalogService.getBooks())
                        .dataFetcher(
                            "bookById",
                            env -> bookCatalogService.bookById(Integer.parseInt(env.getArgument("id"))));
            builder.type(
                "Book",
                wiring ->
                    wiring.dataFetcher("ratings", env -> bookCatalogService.ratings(env.getSource())));
        };
    }
}
```

Running and Testing

You can start the GraphQL service from IDE by running the main method of **GraphQLJavaSpringApplication** class.

```
@SpringBootApplication
public class GraphQLJavaSpringApplication {

    public static void main(String[] args) { SpringApplication.run(GraphQLJavaSpringApplication.class, args); }

}
```

To test, open the GraphQL editor in the browser using the link <http://localhost:8080/graphql-play=graphql>.

The screenshot shows the GraphQL editor with a query on the left and its JSON response on the right. The query is:

```

1 query {
2   bookById(id:1) {
3     id
4     name
5     ratings {
6       user
7       rating
8       comment
9     }
10  }
11 }
12 }

```

The JSON response is:

```

{
  "data": {
    "bookById": {
      "id": "1",
      "name": "Zero to One",
      "ratings": [
        {
          "user": "Konstantinos Papakonstantinou",
          "rating": 5,
          "comment": "The 4 minutes that will help you decide if thi"
        },
        {
          "user": "Konstantinos Papakonstantinou",
          "rating": 3,
          "comment": "Is Peter Thiel the next robber baron?"
        },
        {
          "user": "Todd Holscher",
          "rating": 3,
          "comment": "Simple-minded. Is it satire? Poorly-reasoned?"
        }
      ]
    }
  }
}

```



To enable GraphQL editor, you must set `spring.graphql.graphiql.enabled=true` in application.properties.

Summary

GraphQL, a **query language for API** and a **server-side runtime** for executing queries, is fast emerging as an alternative to REST API. Most significantly, it solves common problems associated with REST APIs such as over-fetching and under-fetching.

Spring Boot GraphQL provides support for Spring applications built on [GraphQL Java](#). It supports the handling of GraphQL requests over HTTP, WebSocket, and RSocket.

gRPC Interceptor: unary interceptor with code example

If you like this article, then please [follow me on LinkedIn](#) in for mo

Next Post

Spring for GraphQL : @SchemaMapping and @QueryMapping

Tags: [graphql](#)



Pankaj

Software Architect @ Schlumberger "" Cloud | Microservices | Programming | Kubernetes | Architecture | Machine Learning | Java | Python ""



Related Posts



GRAPHQL

GraphQL Directive

🕒 SEPTEMBER 29, 2022

GRAPHQL

Spring for GraphQL: Mutation

🕒 AUGUST 14, 2022

GRAPHQL

Spring for GraphQL: How to solve the N+1 Problem?

🕒 AUGUST 7, 2022

GRAPHQL

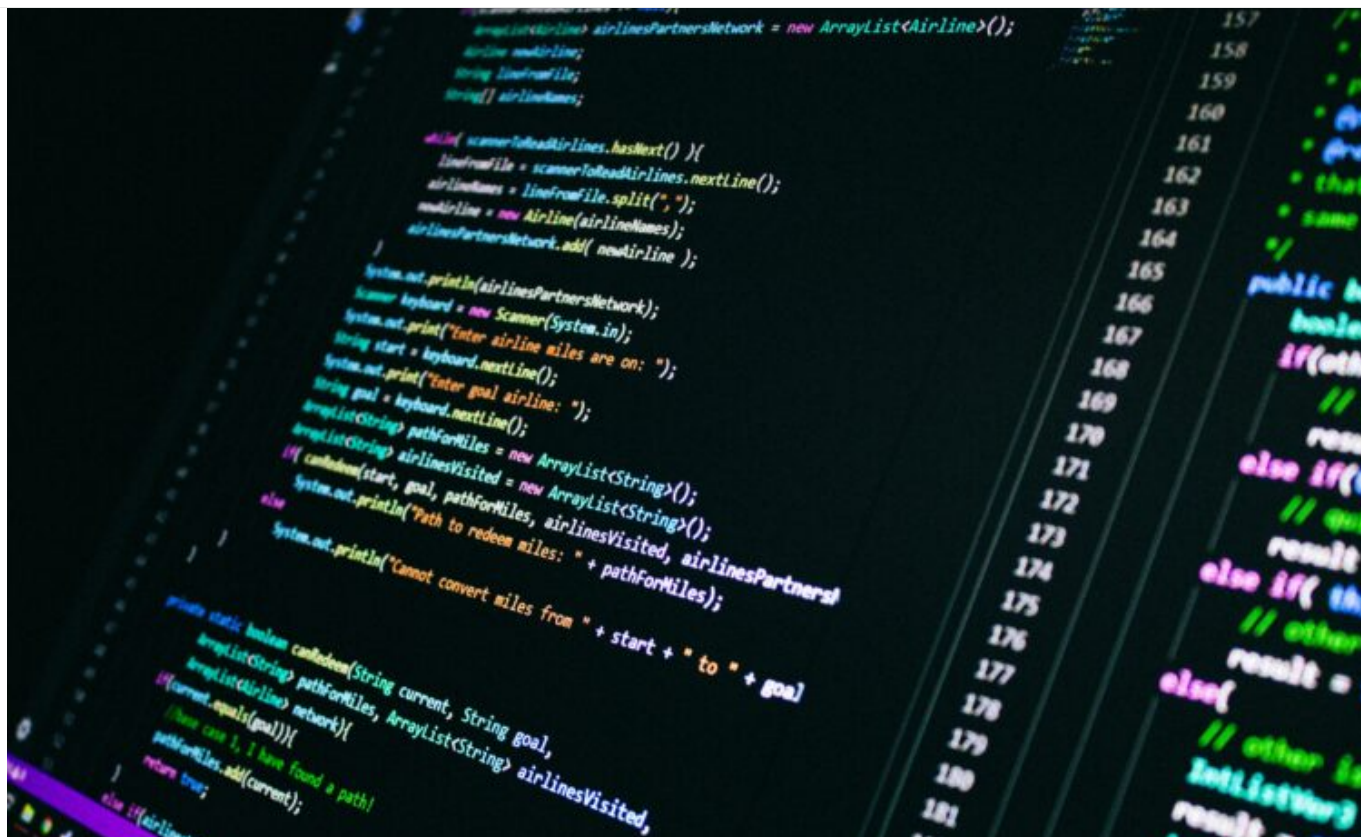
Spring for GraphQL : @SchemaMapping and @QueryMapping

🕒 JULY 29, 2022

Discussion about this post

Recent Articles





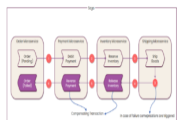
gRPC Bidirectional Streaming with Code Example

© FEBRUARY 17, 2023



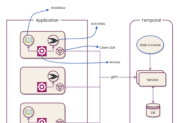
gRPC Client Streaming

© JANUARY 20, 2023



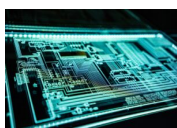
Distributed Transactions in Microservices: implementing Saga with Temporal

© NOVEMBER 8, 2022



Workflow Orchestration with Temporal and Spring Boot

© OCTOBER 29, 2022



Deploying a RESTful Spring Boot Microservice on Kubernetes

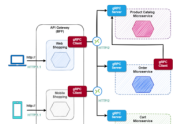
© AUGUST 18, 2021

Workflow Orchestration with Temporal and Spring Boot

© OCTOBER 29, 2022







gRPC for microservices communication

🕒 AUGUST 29, 2021



FACEBOOK



TWITTER



PINTEREST

TECHDOZO

Simplifying modern tech stack!

Browse by Category

Bitesize

Java

Spring Boot

GraphQL

Kubernetes

gRPC

Microservices

Recent Articles

gRPC Bidirectional Streaming with Code Example

🕒 FEBRUARY 17, 2023

gRPC Client Streaming

🕒 JANUARY 20, 2023

© 2023 Techdozo.

