* [BLOG](https://www.graphql-java.com/blog/)
* [TUTORIALS](https://www.graphql-java.com/tutorials/)
* [DOCUMENTATION](https://www.graphql-java.com/documentation/latest/)
* [CONTACT](https://www.graphql-java.com/contact/)
* [ABOUT](https://www.graphql-java.com/about/)

Getting started with GraphQL Java and Spring Boot

* [GraphQL in 3 minutes](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#graphql-in-3-minutes)
* [GraphQL Java Overview](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#graphql-java-overview)
* [Our example API: getting book details](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#our-example-api-getting-book-details)
* [Create a Spring Boot app](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#create-a-spring-boot-app)
* [Schema](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#schema)
* [DataFetchers](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#datafetchers)
  + [Source of the data](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#source-of-the-data)
  + [Book DataFetcher](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#book-datafetcher)
  + [Author DataFetcher](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#author-datafetcher)
  + [Default DataFetchers](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#default-datafetchers)
* [Try out the API](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#try-out-the-api)
* [Complete example source code and more information](https://www.graphql-java.com/tutorials/getting-started-with-spring-boot/#complete-example-source-code-and-more-information)

This is a tutorial for people who want to create a GraphQL server in Java. It requires some Spring Boot and Java knowledge and while we give a brief introduction into GraphQL, the focus of this tutorial is on developing a GraphQL server in Java.

GraphQL in 3 minutes

GraphQL is a query language to retrieve data from a server. It is an alternative to REST, SOAP or gRPC in some way.

Let’s suppose we want to query the details for a specific book from a online store backend.

With GraphQL you send the following query to server to get the details for the book with the id “book-1”:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {  bookById(id: "book-1"){  id  name  pageCount  author {  firstName  lastName  }  }  } |

This is not JSON (even though it looks deliberately similar), it is a GraphQL query. It basically says:

* query a book with a specific id
* get me the id, name, pageCount and author from that book
* for the author I want to know the firstName and lastName

The response is normal JSON:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | {  "bookById":  {  "id":"book-1",  "name":"Harry Potter and the Philosopher's Stone",  "pageCount":223,  "author": {  "firstName":"Joanne",  "lastName":"Rowling"  }  }  } |

One very important property of GraphQL is that it is statically typed: the server knows exactly the shape of every object you can query and any client can actually “introspect” the server and ask for the so called “schema”. The schema describes what queries are possible and what fields you can get back. (Note: when we refer to schema here, we always refer to a “GraphQL Schema”, which is not related to other schemas like “JSON Schema” or “Database Schema”)

The schema for the above query looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | type Query {  bookById(id: ID): Book  }  type Book {  id: ID  name: String  pageCount: Int  author: Author  }  type Author {  id: ID  firstName: String  lastName: String  } |

This tutorial will focus on how to implement a GraphQL server with exactly this schema in Java.

We’ve barely scratched the surface of what’s possible with GraphQL. Further information can be found on the official page: <https://graphql.github.io/learn/>

GraphQL Java Overview

[GraphQL Java](https://www.graphql-java.com/) is the Java (server) implementation for GraphQL. The are several repositories in the GraphQL Java Github org. The most important one is the [GraphQL Java Engine](https://github.com/graphql-java/graphql-java) which is the basis for everything else.

GraphQL Java Engine itself is only concerned with executing queries. It doesn’t deal with any HTTP or JSON related topics. For these aspects, we will use the [GraphQL Java Spring Boot](https://github.com/graphql-java/graphql-java-spring) adapter which takes care of exposing our API via Spring Boot over HTTP.

The main steps of creating a GraphQL Java server are:

1. Defining a GraphQL Schema.
2. Deciding on how the actual data for a query is fetched.

Our example API: getting book details

Our example app will be a simple API to get details for a specific book. This is in no way a comprehensive API, but it is enough for this tutorial.

Create a Spring Boot app

The easiest way to create a Spring Boot app is to use the “Spring Initializr” at <https://start.spring.io/>.

Select:

* Gradle Project
* Java
* Spring Boot 2.1.x

For the project metadata we use:

* Group: com.graphql-java.tutorial
* Artifact: book-details

As dependency, we just select Web.

A click on Generate Project will give you a ready to use Spring Boot app. All subsequently mentioned files and paths will be relative to this generated project.

We are adding three dependencies to our project inside the dependencies section of build.gradle:

the first two are GraphQL Java and GraphQL Java Spring and then we also add [Google Guava](https://github.com/google/guava). Guava is not strictly needed but it will make our life a little bit easier.

The dependencies will look like that:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | dependencies {  implementation 'com.graphql-java:graphql-java:11.0' // NEW  implementation 'com.graphql-java:graphql-java-spring-boot-starter-webmvc:1.0' // NEW  implementation 'com.google.guava:guava:26.0-jre' // NEW  implementation 'org.springframework.boot:spring-boot-starter-web'  testImplementation 'org.springframework.boot:spring-boot-starter-test'  } |

Schema

We are creating a new file schema.graphqls in src/main/resources with the following content:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | type Query {  bookById(id: ID): Book  }  type Book {  id: ID  name: String  pageCount: Int  author: Author  }  type Author {  id: ID  firstName: String  lastName: String  } |

This schema defines one top level field (in the type Query): bookById which returns the details of a specific book.

It also defines the type Book which has the fields: id, name, pageCount and author. author is of type Author, which is defined after Book.

The Domain Specific Language shown above which is used to describe a schema is called Schema Definition Language or SDL. More details about it can be found [here](https://graphql.org/learn/schema/).

Once we have this file we need to “bring it to life” by reading the file and parsing it and adding code to fetch data for it.

We create a new GraphQLProvider class in the package com.graphqljava.tutorial.bookdetails with an init method which will create a GraphQL instance:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | @Component  public class GraphQLProvider {  private GraphQL graphQL;  @Bean  public GraphQL graphQL() {  return graphQL;  }  @PostConstruct  public void init() throws IOException {  URL url = Resources.getResource("schema.graphqls");  String sdl = Resources.toString(url, Charsets.UTF\_8);  GraphQLSchema graphQLSchema = buildSchema(sdl);  this.graphQL = GraphQL.newGraphQL(graphQLSchema).build();  }  private GraphQLSchema buildSchema(String sdl) {  // TODO: we will create the schema here later  }  } |

We use Guava Resources to read the file from our classpath, then create a GraphQLSchema and GraphQL instance. This GraphQL instance is exposed as a Spring Bean via the graphQL() method annotated with @Bean. The GraphQL Java Spring adapter will use that GraphQL instance to make our schema available via HTTP on the default url /graphql.

What we still need to do is to implement the buildSchema method which creates the GraphQLSchema instance and wires in code to fetch data:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @Autowired  GraphQLDataFetchers graphQLDataFetchers;  private GraphQLSchema buildSchema(String sdl) {  TypeDefinitionRegistry typeRegistry = new SchemaParser().parse(sdl);  RuntimeWiring runtimeWiring = buildWiring();  SchemaGenerator schemaGenerator = new SchemaGenerator();  return schemaGenerator.makeExecutableSchema(typeRegistry, runtimeWiring);  }  private RuntimeWiring buildWiring() {  return RuntimeWiring.newRuntimeWiring()  .type(newTypeWiring("Query")  .dataFetcher("bookById", graphQLDataFetchers.getBookByIdDataFetcher()))  .type(newTypeWiring("Book")  .dataFetcher("author", graphQLDataFetchers.getAuthorDataFetcher()))  .build();  } |

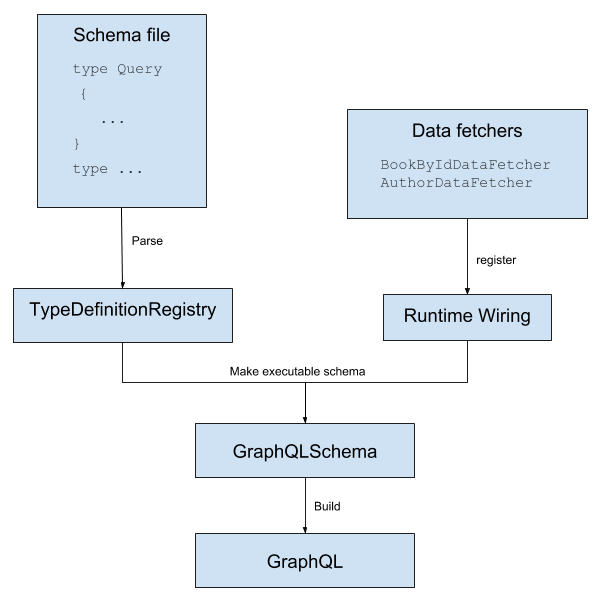
TypeDefinitionRegistry is the parsed version of our schema file. SchemaGenerator combines the TypeDefinitionRegistry with RuntimeWiring to actually make the GraphQLSchema.

buildWiring uses the graphQLDataFetchers bean to actually register two DataFetchers:

* One to retrieve a book with a specific ID
* One to get the author for a specific book.

DataFetcher and how to implement the GraphQLDataFetchers bean is explained in the next section.

Overall the process of creating a GraphQL and GraphQLSchema instance looks like this:



DataFetchers

Probably the most important concept for a GraphQL Java server is a DataFetcher: A DataFetcher fetches the Data for one field while the query is executed.

While GraphQL Java is executing a query, it calls the appropriate DataFetcher for each field it encounters in query. A DataFetcher is an Interface with a single method, taking a single argument of type DataFetcherEnvironment:

|  |  |
| --- | --- |
| 1  2  3 | public interface DataFetcher<T> {  T get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception;  } |

Important: **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. We will discuss this later in more detail.

We are creating a new class GraphQLDataFetchers which contains a sample list of books and authors.

The full implementation looks like this which we will look at it in detail soon:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | @Component  public class GraphQLDataFetchers {  private static List<Map<String, String>> books = Arrays.asList(  ImmutableMap.of("id", "book-1",  "name", "Harry Potter and the Philosopher's Stone",  "pageCount", "223",  "authorId", "author-1"),  ImmutableMap.of("id", "book-2",  "name", "Moby Dick",  "pageCount", "635",  "authorId", "author-2"),  ImmutableMap.of("id", "book-3",  "name", "Interview with the vampire",  "pageCount", "371",  "authorId", "author-3")  );  private static List<Map<String, String>> authors = Arrays.asList(  ImmutableMap.of("id", "author-1",  "firstName", "Joanne",  "lastName", "Rowling"),  ImmutableMap.of("id", "author-2",  "firstName", "Herman",  "lastName", "Melville"),  ImmutableMap.of("id", "author-3",  "firstName", "Anne",  "lastName", "Rice")  );  public DataFetcher getBookByIdDataFetcher() {  return dataFetchingEnvironment -> {  String bookId = dataFetchingEnvironment.getArgument("id");  return books  .stream()  .filter(book -> book.get("id").equals(bookId))  .findFirst()  .orElse(null);  };  }  public DataFetcher getAuthorDataFetcher() {  return dataFetchingEnvironment -> {  Map<String,String> book = dataFetchingEnvironment.getSource();  String authorId = book.get("authorId");  return authors  .stream()  .filter(author -> author.get("id").equals(authorId))  .findFirst()  .orElse(null);  };  }  } |

Source of the data

We are getting our books and authors from a static list inside the class. This is made just for this tutorial. It is very important to understand that GraphQL doesn’t dictate in anyway where the data comes from. This is the power of GraphQL: it can come from a static in memory list, from a database or an external service

Book DataFetcher

Our first method getBookByIdDataFetcher returns a DataFetcher implementation which takes a DataFetcherEnvironment and returns a book. In our case this means we need to get the id argument from the bookById field and find the book with this specific id. If we can’t find it, we just return null.

The “id” in String bookId = dataFetchingEnvironment.getArgument("id"); is the “id” from the bookById query field in the schema:

|  |  |
| --- | --- |
| 1  2  3  4 | type Query {  bookById(id: ID): Book  }  ... |

Author DataFetcher

Our second method getAuthorDataFetcher, returns a DataFetcher for getting the author for a specific book. Compared to the previously described book DataFetcher, we don’t have an argument, but we have a book instance. The result of the DataFetcher from the parent field is made available via getSource. This is an important concept to understand: the DataFetcher for each field in GraphQL are called in a top-down fashion and the parent’s result is the source property of the child DataFetcherEnvironment.

We then use the previously fetched book to get the authorId and look for that specific author in the same way we look for a specific book.

Default DataFetchers

We only implement two DataFetchers. As mentioned above, if you don’t specify one, the default PropertyDataFetcher is used. In our case it means Book.id, Book.name, Book.pageCount, Author.id, Author.firstName and Author.lastName all have a default PropertyDataFetcher associated with it.

A PropertyDataFetcher tries to lookup a property on a Java object in multiple ways. In case of a java.util.Map it simply looks up the property by key. This works perfectly fine for us because the keys of the book and author Maps are the same as the fields specified in the schema. For example in the schema we define for the Book type the field pageCount and the book DataFetcher returns a Map with a key pageCount. Because the field name is the same as the key in the Map(“pageCount”) the PropertyDateFetcher works for us.

Lets assume for a second we have a mismatch and the book Map has a key totalPages instead of pageCount. This would result in a null value for pageCount for every book, because the PropertyDataFetcher can’t fetch the right value. In order to fix that you would have to register a new DataFetcher for Book.pageCount which looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | // In the GraphQLProvider class  private RuntimeWiring buildWiring() {  return RuntimeWiring.newRuntimeWiring()  .type(newTypeWiring("Query")  .dataFetcher("bookById", graphQLDataFetchers.getBookByIdDataFetcher()))  .type(newTypeWiring("Book")  .dataFetcher("author", graphQLDataFetchers.getAuthorDataFetcher())  // This line is new: we need to register the additional DataFetcher  .dataFetcher("pageCount", graphQLDataFetchers.getPageCountDataFetcher()))  .build();  }  // In the GraphQLDataFetchers class  // Implement the DataFetcher  public DataFetcher getPageCountDataFetcher() {  return dataFetchingEnvironment -> {  Map<String,String> book = dataFetchingEnvironment.getSource();  return book.get("totalPages");  };  }  ... |

This DataFetcher would fix that problem by looking up the right key in the book Map. (Again: we don’t need that for our example, because we don’t have a naming mismatch)

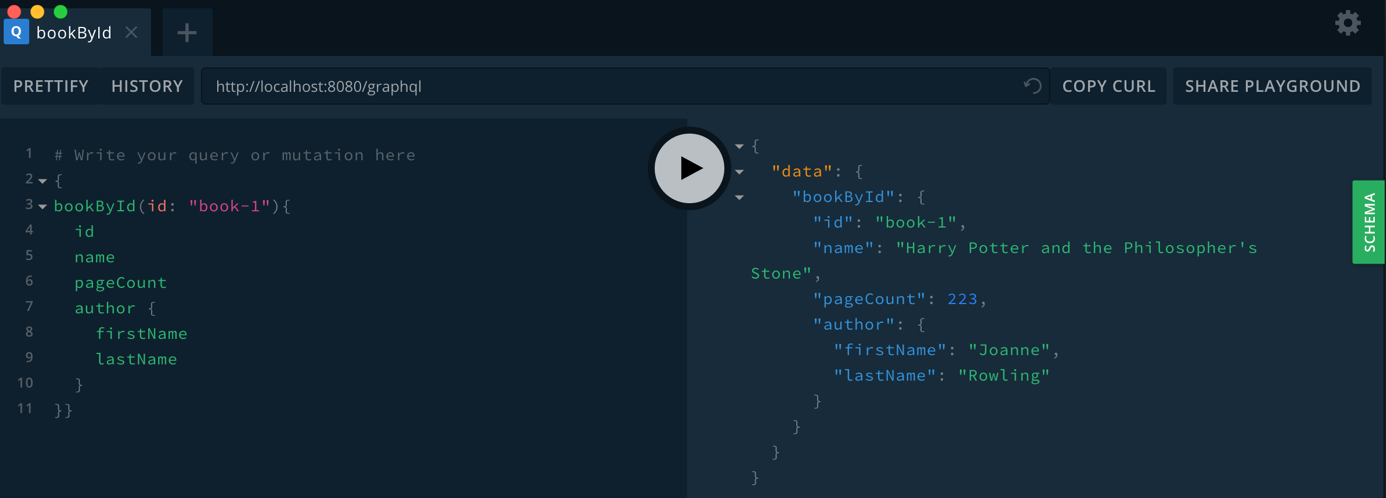
Try out the API

This is all you actually need to build a working GraphQL API. After the starting the Spring Boot application the API is available on http://localhost:8080/graphql.

The easiest way to try out and explore a GraphQL API is to use a tool like [GraphQL Playground](https://github.com/prisma/graphql-playground). Download it and run it.

After starting it you will be asked for a URL, enter “[http://localhost:8080/graphql"](http://localhost:8080/graphql%22).

After that, you can query our example API and you should get back the result we mentioned above in the beginning. It should look something like this:



Complete example source code and more information

The complete project with the full source code can be found here: <https://github.com/graphql-java/tutorials/tree/master/book-details>

More information about GraphQL Java can be found in the [documentation](https://www.graphql-java.com/documentation/).

We also have [spectrum chat](https://spectrum.chat/graphql-java) for any question or problems.

For direct feedback you can also ping us on our [GraphQL Java Twitter account](https://twitter.com/graphql_java).

© Andreas Marek. Powered by [Hugo](https://gohugo.io/) and [sustain](http://www.github.com/nurlansu/hugo-sustain/) with ♥